Modeling and Programming with Gecode

Christian Schulte

Guido Tack

Mikael Z. Lagerkvist

This document was published on November 5, 2014. It corresponds to Gecode 4.3.2.

© Christian Schulte, Guido Tack, Mikael Z. Lagerkvist, 2008–2014.

License information

This document is provided under the terms of the

Creative Commons License Attribution-Noncommercial-No Derivative Works 3.0

The complete license text can be found at the end of this document.

We kindly ask you to not make this document available on the Internet such that it might be indexed by a search engine. We insist that search engines should point to a single and up-to-date version of this document.

The location of this document is:

`http://www.gecode.org/doc/4.3.2/MPG.pdf`

The location of this document corresponding to the latest Gecode version is:

`http://www.gecode.org/doc-latest/MPG.pdf`

Acknowledgments

We thank the following people for helpful comments: Vincent Barichard, Léonard Benedetti, Pavel Bochman, Felix Brandt, Markus Böhm, Roberto Castañeda Lozano, Gregory Crosswhite, Pierre Flener, Gustavo Gutierrez, Gabriel Hjort Blindell, Sverker Janson, Andreas Karlsson, Håkan Kjellerstrand, Chris Mears, Benjamin Negrevergne, Flutra Osmani, Max Ostrowski, David Rijsman, Dan Scott, Kish Shen.

Contents

1 Introduction **1**
- 1.1 What is Gecode? . . . 1
- 1.2 What is this document? . . . 2
- 1.3 How to read this document? . . . 4
- 1.4 Do I need to be a C++ wizard? . . . 5
- 1.5 Can you help me? . . . 6
- 1.6 Does Gecode have bugs? . . . 6
- 1.7 How to refer to this document? . . . 7
- 1.8 Do you have comments? . . . 7

M Modeling **9**

2 Getting started **13**
- 2.1 A first Gecode model . . . 13
- 2.2 Searching for solutions . . . 18
- 2.3 Compiling, linking, and executing . . . 20
  - 2.3.1 Microsoft Visual Studio . . . 20
  - 2.3.2 Apple Mac OS . . . 22
  - 2.3.3 Linux and relatives . . . 23
- 2.4 Using Gist . . . 24
- 2.5 Best solution search . . . 27
- 2.6 Obtaining Gecode . . . 29
  - 2.6.1 Installing Gecode . . . 29
  - 2.6.2 Compiling Gecode . . . 30
  - 2.6.3 Advanced configuration and compilation . . . 32

3 Getting comfortable **35**
- 3.1 Posting linear constraints de-mystified . . . 35
- 3.2 Using a cost function . . . 35
- 3.3 Using the script commandline driver . . . 38

4 Integer and Boolean variables and constraints **43**
- 4.1 Integer and Boolean variables . . . 44
  - 4.1.1 Creating integer variables . . . 44

4.1.2 Limits for integer values . . . 45
4.1.3 Variable domains are never empty . . . 45
4.1.4 Creating Boolean variables . . . 46
4.1.5 Variable access functions . . . 46
4.1.6 Iterating over integer variable domains . . . 46
4.1.7 When to inspect a variable . . . 47
4.1.8 Updating variables . . . 47
4.2 Variable and argument arrays . . . 48
4.2.1 Integer and Boolean variable arrays . . . 48
4.2.2 Argument arrays . . . 49
4.2.3 STL-style iterators . . . 51
4.3 Posting constraints . . . 52
4.3.1 Post functions are clever . . . 52
4.3.2 Everything is copied . . . 52
4.3.3 Reified constraints . . . 52
4.3.4 Half reification . . . 53
4.3.5 Selecting the consistency level . . . 54
4.3.6 Exceptions . . . 55
4.3.7 Unsharing arguments . . . 55
4.4 Constraint overview . . . 56
4.4.1 Domain constraints . . . 56
4.4.2 Membership constraints . . . 57
4.4.3 Simple relation constraints over integer variables . . . 57
4.4.4 Simple relation constraints over Boolean variables . . . 59
4.4.5 Arithmetic constraints . . . 61
4.4.6 Linear constraints . . . 61
4.4.7 Distinct constraints . . . 62
4.4.8 Counting constraints . . . 63
4.4.9 Number of values constraints . . . 65
4.4.10 Sequence constraints . . . 65
4.4.11 Channel constraints . . . 66
4.4.12 Element constraints . . . 67
4.4.13 Extensional constraints . . . 68
4.4.14 Sorted constraints . . . 70
4.4.15 Bin-packing constraints . . . 70
4.4.16 Geometrical packing constraints . . . 72
4.4.17 Circuit and Hamiltonian path constraints . . . 73
4.4.18 Scheduling constraints . . . 75
4.4.19 Value precedence constraints . . . 78
4.5 Synchronized execution . . . 79

5 Set variables and constraints **81**
5.1 Set variables . . . 81
5.2 Constraint overview . . . 84
5.2.1 Domain constraints . . . 84
5.2.2 Relation constraints . . . 85
5.2.3 Set operations . . . 86
5.2.4 Element constraints . . . 87
5.2.5 Constraints connecting set and integer variables . . . 87
5.2.6 Set channeling constraints . . . 88
5.2.7 Convexity constraints . . . 89
5.2.8 Sequence constraints . . . 89
5.2.9 Value precedence constraints . . . 89
5.3 Synchronized execution . . . 90

6 Float variables and constraints **91**
6.1 Float values and numbers . . . 91
6.2 Float variables . . . 92
6.3 Constraint overview . . . 95
6.3.1 Domain constraints . . . 95
6.3.2 Simple relation constraints . . . 96
6.3.3 Arithmetic constraints . . . 97
6.3.4 Linear constraints . . . 97
6.3.5 Channel constraints . . . 98
6.4 Synchronized execution . . . 98

7 Modeling convenience: MiniModel **99**
7.1 Expressions and relations . . . 99
7.1.1 Integer expressions and relations . . . 100
7.1.2 Boolean expressions and relations . . . 102
7.1.3 Set expressions and relations . . . 104
7.1.4 Float expressions and relations . . . 105
7.2 Matrix interface for arrays . . . 107
7.3 Support for cost-based optimization . . . 108
7.4 Regular expressions for extensional constraints . . . 109
7.5 Channeling functions . . . 110
7.6 Aliases for integer constraints . . . 110
7.7 Aliases for set constraints . . . 110

8 Branching **113**
8.1 Branching basics . . . 113
8.2 Branching on integer and Boolean variables . . . 115
8.3 Branching on set variables . . . 118
8.4 Branching on float variables . . . 120

8.5 Local versus shared variable selection criteria . . . 121
8.5.1 Local variable selection criteria . . . 121
8.5.2 Selection using accumulated failure count . . . 122
8.5.3 Selection using activity . . . 123
8.6 Random variable and value selection . . . 124
8.7 User-defined variable selection . . . 125
8.8 User-defined value selection . . . 126
8.9 Tie-breaking . . . 128
8.10 Lightweight Dynamic Symmetry Breaking . . . 130
8.10.1 Specifying Symmetry . . . 132
8.10.2 Notes . . . 132
8.11 Using branch filter functions . . . 133
8.12 Using variable-value print functions . . . 134
8.13 Assigning integer, Boolean, set, and float variables . . . 135
8.14 Executing code between branchers . . . 135

9 Search **139**
9.1 Hybrid recomputation . . . 139
9.1.1 Cloning . . . 140
9.1.2 Recomputation . . . 140
9.1.3 Hybrid recomputation . . . 141
9.1.4 Why recomputation is almost for free . . . 142
9.1.5 Adaptive recomputation . . . 142
9.1.6 Controlling recomputation . . . 143
9.2 Parallel search . . . 143
9.3 Search engines . . . 147
9.3.1 Search options . . . 148
9.3.2 Stop objects . . . 149
9.4 Restart-based search . . . 150
9.4.1 Restart-based search as a meta search engine . . . 151
9.4.2 Cutoff generators . . . 151
9.4.3 Computing a next solution . . . 154
9.4.4 Master and slave configuration . . . 154
9.4.5 Large Neighborhood Search . . . 156
9.5 No-goods from restarts . . . 158

10 Gist **163**
10.1 The search tree . . . 163
10.2 Invoking Gist . . . 164
10.2.1 Standalone use . . . 164
10.2.2 Use as a Qt widget . . . 165
10.3 Using Gist . . . 165
10.3.1 Automatic search . . . 165

10.3.2 Interactive search . . . 166
10.3.3 Branch-and-bound search . . . 167
10.3.4 Inspecting and comparing nodes . . . 167
10.3.5 Zooming, centering, exporting, printing . . . 170
10.3.6 Options and preferences . . . 172

11 Script commandline driver **175**
11.1 Commandline options . . . 175
11.2 Scripts . . . 177

C Case studies 179

12 Golomb rulers **181**
12.1 Problem . . . 181
12.2 Model . . . 182
12.3 More information . . . 185

13 Magic sequence **187**
13.1 Problem . . . 187
13.2 Model . . . 187
13.3 More information . . . 189

14 Photo alignment **191**
14.1 Problem . . . 191
14.2 Model . . . 191
14.3 More information . . . 194

15 Locating warehouses **195**
15.1 Problem . . . 195
15.2 Model . . . 197
15.3 More information . . . 199

16 Nonogram **201**
16.1 Problem . . . 201
16.2 Model . . . 201
16.3 More information . . . 205

17 Social golfers **207**
17.1 Problem . . . 207
17.2 Model . . . 207
17.3 More information . . . 210

18 Knight's tour **211**
18.1 Problem . . . 211
18.2 Model . . . 211
18.3 Branching . . . 214
18.4 More information . . . 216

19 Bin packing **219**
19.1 Problem . . . 219
19.2 A naive model . . . 220
19.3 Improving propagation . . . 225
19.4 Improving branching . . . 226
19.5 More information . . . 233

20 Kakuro **235**
20.1 Problem . . . 235
20.2 A naive model . . . 235
20.3 A working model . . . 240
20.4 More information . . . 243

21 Crossword puzzle **245**
21.1 Problem . . . 245
21.2 Model . . . 245
21.3 An optimized model . . . 251
21.4 More information . . . 253

P Programming propagators **259**

22 Getting started **261**
22.1 Constraint propagation in a nutshell . . . 261
22.2 Background reading . . . 264
22.3 What to implement? . . . 265
22.4 Implementing the `less` constraint . . . 268
22.5 Improving the `Less` propagator . . . 274
22.6 Propagation conditions . . . 277
22.7 Using propagator patterns . . . 281
22.8 Propagator obligations . . . 282
22.9 Waiving obligations . . . 284

23 Avoiding execution **287**
23.1 Fixpoint reasoning reconsidered . . . 287
23.2 A Boolean disjunction propagator . . . 291
23.3 Dynamic subscriptions . . . 295

24 Reification and rewriting **299**
24.1 Reification . . . 299
24.2 A fully reified less or equal propagator . . . 300
24.3 Supporting both full and half reification . . . 302
24.4 Rewriting during propagation . . . 305
24.5 Rewriting during cloning . . . 307

25 Domain propagation **311**
25.1 Why domain operations are needed . . . 311
25.2 Iterator-based modification operations . . . 313
25.3 Taking advantage of iterators . . . 316
25.4 Modification event deltas . . . 318
25.5 Staging . . . 320

26 Advisors **325**
26.1 Advisors for incremental propagation . . . 325
26.2 The `samedom` constraint . . . 327
26.3 General Boolean disjunction . . . 332
26.4 Forced propagator rescheduling . . . 336

27 Views **337**
27.1 Integer views . . . 337
27.1.1 Minus views . . . 337
27.1.2 Offset views . . . 339
27.1.3 Constant and scale views . . . 340
27.2 Boolean views . . . 340
27.3 Integer propagators on Boolean views . . . 342

28 Propagators for set constraints **343**
28.1 A simple example . . . 343
28.2 Modification events, propagation conditions, views, and advisors . . . 347

29 Propagators for float constraints **351**
29.1 A simple example . . . 351
29.2 Modification events, propagation conditions, views, and advisors . . . 354

30 Managing memory **357**
30.1 Memory areas . . . 357
30.2 Managing propagator state . . . 360
30.3 Shared objects and handles . . . 361
30.4 Local objects and handles . . . 363

B Programming branchers **367**

31 Getting started **369**
31.1 What to implement? . . . 369
31.2 Implementing a `nonemin` branching . . . 373
31.2.1 A naive brancher . . . 374
31.2.2 Improving status and choice . . . 377
31.3 Implementing a `sizemin` branching . . . 377

32 Advanced topics **381**
32.1 Assignment branchers . . . 381
32.2 Supporting no-goods . . . 381
32.2.1 Returning no-good literals . . . 383
32.2.2 Implementing no-good literals . . . 385
32.3 Using variable views . . . 387

V Programming variables **389**

33 Getting started **391**
33.1 Overview . . . 391
33.2 Structure . . . 393

34 Variable implementations **397**
34.1 Design decisions . . . 397
34.2 Base definitions . . . 400
34.3 Variable implementation . . . 403
34.4 Additional specification options . . . 408

35 Variables and variable arrays **411**
35.1 Variables . . . 411
35.2 Variable arrays and variable argument arrays . . . 413

36 Views **417**
36.1 View types . . . 417
36.2 Variable implementation views: integer view . . . 419
36.3 Constant views: constant integer view . . . 420
36.4 Derived views . . . 422
36.4.1 Minus views . . . 422
36.4.2 Offset views . . . 425

37 Variable-value branchings **429**
37.1 Type, traits, and activity definitions . . . 429
37.2 Variable and value selection . . . 432

37.3 View selection creation . . . 435
37.4 Value selection and commit creation . . . 438
37.5 Branchings . . . 441

38 Putting everything together **445**
38.1 Golomb rulers à la integer interval variables . . . 445
38.2 Configuring and compiling Gecode . . . 445

S Programming search engines **449**

39 Getting started **451**
39.1 Space-based search . . . 451
39.2 Binary depth-first search . . . 455
39.3 Depth-first search . . . 457
39.4 Branch-and-bound search . . . 459

40 Recomputation **463**
40.1 Full recomputation . . . 463
40.2 Recomputation invariants . . . 467
40.2.1 Choice compatibility . . . 467
40.2.2 Recomputation is not deterministic . . . 468
40.3 Branch-and-bound search . . . 469
40.4 Last alternative optimization . . . 471
40.5 Hybrid recomputation . . . 473
40.6 Adaptive recomputation . . . 476

41 An example engine **479**
41.1 Engine design . . . 479
41.2 Engine implementation . . . 479
41.3 Exploration . . . 482
41.4 Recomputation . . . 485

Bibliography **489**

Changelog **495**

License **501**

Figures

1.1 Gecode architecture . . . 3
1.2 Dependencies among different parts of this document . . . 4

2.1 A Gecode model for Send More Money . . . 14
2.2 Using Gist for Send More Money . . . 25
2.3 Gist screen shots . . . 25
2.4 Using Gist for Send More Money with node inspection . . . 26
2.5 A Gecode model for Send Most Money finding a best solution . . . 27

3.1 A Gecode model for Send More Money using modeling support . . . 36
3.2 A Gecode model for Send Most Money using a cost function . . . 37
3.3 A Gecode model for Send Most Money using the script commandline driver 39

4.1 Integer relation types . . . 57
4.2 Boolean operation types . . . 59
4.3 Arithmetic constraints . . . 61
4.4 A DFA for the Swedish drinking protocol . . . 69
4.5 Representing edges and propagating `circuit` . . . 73
4.6 Representing edges and propagating `circuit` with cost . . . 74

5.1 Set relation types . . . 85
5.2 Set operation types . . . 86

6.1 Functions on float values . . . 93
6.2 Float relation types . . . 96
6.3 Arithmetic constraints . . . 97

7.1 Integer expressions and relations . . . 101
7.2 Boolean expressions . . . 102
7.3 Set expressions and relations . . . 105
7.4 Float expressions and relations . . . 106
7.5 Constructing regular expressions . . . 109
7.6 Aliases for integer constraints . . . 111

8.1 Integer and Boolean variable selection . . . 115
8.2 Integer and Boolean value selection . . . 116
8.3 Set variable selection . . . 119

8.4 Set value selection . . . 119
8.5 Float variable selection . . . 120
8.6 Float value selection . . . 121
8.7 Branch value functions . . . 127
8.8 Branch commit functions . . . 128
8.9 A Gecode model for Latin Squares with LDSB . . . 130
8.10 Symmetric solutions of the Latin Square problem . . . 131
8.11 Model sketch for branch filter function . . . 133
8.12 Value selection for assigning variables . . . 136

9.1 Example search tree . . . 140
9.2 Hybrid recomputation . . . 141
9.3 Output for Golomb rulers with eight workers . . . 145
9.4 Output for Golomb rulers with one worker . . . 146
9.5 Search statistics . . . 147
9.6 Available search engines . . . 148
9.7 Search options . . . 149
9.8 Predefined stop objects . . . 150
9.9 Current restart information . . . 155
9.10 Default `master()` and `slave()` functions . . . 156
9.11 Model sketch for LNS . . . 157
9.12 Search tree after cutoff 3 has been reached . . . 158

10.1 A search tree . . . 163
10.2 Gist, solving the Send More Money problem . . . 164
10.3 A hidden subtree in Gist . . . 166
10.4 The *Node* menu . . . 167
10.5 Branch information in Gist . . . 168
10.6 The *Tools* menu . . . 169
10.7 Inspecting a solution in Gist . . . 170
10.8 Using Gist for Send More Money with node comparison . . . 171
10.9 Node statistics . . . 172
10.10 Gist preferences . . . 173
10.11 Displaying where Gist stores spaces in the tree . . . 174

11.1 Predefined commandline options . . . 176
11.2 User-definable commandline options . . . 177

12.1 An optimal Golomb ruler with 6 marks . . . 181
12.2 A constructed Golomb ruler with 6 marks . . . 181
12.3 A script for computing Golomb rulers . . . 182

13.1 A script for solving magic sequence puzzles . . . 188
13.2 Magic sequence puzzles with a global counting constraint . . . 189

14.1 A script for the photo alignment problem . . . 192

15.1 A script for locating warehouses . . . 196

16.1 Example nonogram puzzle . . . 202
16.2 Solution to the example puzzle . . . 202
16.3 A script for solving nonogram puzzles . . . 203

17.1 A script for the social golfers' problem . . . 208

18.1 8 × 8-knight's tour . . . 212
18.2 A script for the knight's tour problem . . . 213
18.3 A brancher for Warnsdorff's heuristic . . . 217

19.1 An example optimal bin packing . . . 219
19.2 Instance data for a bin packing problem . . . 220
19.3 Computing a lower bound for the number of bins . . . 220
19.4 Computing an upper bound for the number of bins . . . 221
19.5 An non-optimal bin packing found during upper bound computation . . . 222
19.6 A naive script for solving a bin packing problem . . . 223
19.7 A script with improved propagation for solving a bin packing problem . . . 226
19.8 A script with improved branching for solving a bin packing problem . . . 227
19.9 CDBF brancher and branching . . . 228
19.10 CDBF choice . . . 232

20.1 A Kakuro puzzle . . . 236
20.2 Solution for Kakuro puzzle from Figure 20.1 . . . 236
20.3 A naive script and board specification for solving Kakuro puzzles . . . 237
20.4 Propagation for the Kakuro puzzle . . . 240
20.5 A working script for solving Kakuro puzzles . . . 241

21.1 A crossword puzzle grid . . . 246
21.2 Solution for crossword puzzle grid from Figure 21.1 . . . 246
21.3 Crossword script . . . 247
21.4 Grid and words specification . . . 248
21.5 An optimized crossword script . . . 252
21.6 Comparison of Gecode model with COMBUS . . . 254
21.7 Comparison of Gecode model using restarts with COMBUS . . . 256
21.8 Results for some hard `words` dictionary instances . . . 256
21.9 Solution for instance `words-21×21-10` . . . 257
21.10 Solution for instance `words-23×23-06` . . . 257

22.1 Scheduling and executing propagators . . . 263
22.2 Propagators, views, and variable implementations . . . 266
22.3 A constraint and propagator for `less` . . . 269

22.4 Summary of propagation cost functions . . . 272
22.5 Value-based modification functions for integer variable views . . . 273
22.6 A better constraint and propagator for `less` . . . 274
22.7 Check and fail macros . . . 277
22.8 An even better constraint and propagator for `less` . . . 279
22.9 A propagator for disequality . . . 280
22.10 Propagator patterns . . . 281
22.11 A concise constraint and propagator for `less` . . . 283

23.1 A naive equality bounds propagator . . . 288
23.2 An equality bounds propagator with fixpoint reasoning . . . 289
23.3 An idempotent equality bounds propagator . . . 289
23.4 An idempotent equality bounds propagator using modification events . . . 290
23.5 Naive Boolean disjunction . . . 292
23.6 Propagation for naive Boolean disjunction . . . 294
23.7 Naive Boolean disjunction using a propagator pattern . . . 295
23.8 Boolean disjunction with dynamic subscriptions . . . 296
23.9 Resubscribing for Boolean disjunction with dynamic subscriptions . . . 298

24.1 A constraint and propagator for fully reified less or equal . . . 301
24.2 A constraint and propagator for full and half reified less or equal . . . 303
24.3 Propagate function for full and half reified less or equal . . . 304
24.4 A maximum propagator using rewriting . . . 306
24.5 A Boolean disjunction propagator using rewriting . . . 308

25.1 An incorrect propagator for domain equal . . . 312
25.2 A naive propagator for domain equal . . . 313
25.3 A propagator for domain equal without sharing . . . 315
25.4 A propagator for domain equal with offset . . . 317
25.5 A propagator for domain equal using bounds propagation . . . 319
25.6 Stage transitions for the equality propagator . . . 321
25.7 A propagator for domain equal using staging . . . 322

26.1 A `samedom` propagator using advisors . . . 329
26.2 A `samedom` propagator using predefined view advisors . . . 333
26.3 A Boolean disjunction propagator using advisors . . . 334

27.1 Minimum and maximum constraints implemented by a `Max` propagator . . . 338
27.2 Domain equality with and without offset . . . 339
27.3 Disjunction and conjunction from same propagator . . . 341
27.4 Less constraints for both integer and Boolean variables . . . 342

28.1 A constraint and propagator for set intersection . . . 344
28.2 Set view operations . . . 346

28.3 Set modification events and propagation conditions . . . 349

29.1 A constraint and propagator for ternary linear . . . 352
29.2 Most important float view operations . . . 353
29.3 Rounding operations on float numbers . . . 353
29.4 Float modification events and propagation conditions . . . 355

30.1 A simple shared object and handle . . . 362
30.2 A simple local object and handle . . . 364
30.3 A local object and handle with external resources . . . 365

31.1 A branching and brancher for `nonemin` . . . 374
31.2 An improved brancher for `nonemin` . . . 378
31.3 A brancher for `sizemin` . . . 379

32.1 A brancher for `assignmin` . . . 382
32.2 Branching for `nonemin` with no-good support . . . 384
32.3 Branchings for `nonemin` and `nonemax` . . . 387

33.1 The header file for integer interval variables . . . 394

34.1 Variable implementation specification . . . 401
34.2 Modification event section . . . 402
34.3 Propagation condition section . . . 403
34.4 Variable implementation . . . 404
34.5 Summary of member functions predefined by variable implementations . . . 408

35.1 Variable programmed from a variable implementation . . . 412
35.2 Summary of member functions predefined by variables . . . 413
35.3 Array traits for variable arrays . . . 414
35.4 Variable arrays . . . 415

36.1 Summary of member functions predefined by views . . . 418
36.2 Integer view . . . 419
36.3 Constant integer view . . . 421
36.4 Minus view . . . 423
36.5 Negation of modification events and propagation conditions . . . 424
36.6 Offset view . . . 426

37.1 Part of header file concerned with branching . . . 430
37.2 Variable selection class . . . 434
37.3 View selection creation function . . . 436
37.4 Size merit class . . . 437
37.5 Value selection and commit creation function . . . 438
37.6 Branch function . . . 441

37.7 Branch function with tie-breaking 443

38.1 Golomb rulers à la integer interval variables 446

39.1 Depth-first search for binary choices 456
39.2 Depth-first search 458
39.3 Branch-and-bound search 460

40.1 Depth-first search using full recomputation 464
40.2 `Edge` class for depth-first search using full recomputation 465
40.3 Example situations during recomputation 468
40.4 Branch-and-bound search using full recomputation 470
40.5 Last alternative optimization (LAO) 472
40.6 Depth-first search using full recomputation and LAO 472
40.7 Depth-first search using hybrid recomputation 474
40.8 Depth-first search using adaptive recomputation 477

41.1 Depth-first search engine 480
41.2 Implementation of depth-first search engine 481
41.3 Implementation of exploration 483
41.4 Implementation of edges 484
41.5 Implementation of path of edges 485
41.6 Implementation of recomputation 486

Tips

2.1 `Space&` versus `Home` . . . 15
2.2 Propagation is explicit . . . 19
2.3 Catching Gecode exceptions . . . 20
2.4 Cygwin with Microsoft Visual Studio . . . 22
2.5 Eclipse on Windows and Mac OS . . . 24
2.6 Gist scales . . . 27
2.7 Linking against Gist . . . 27
2.8 Compiling on Windows for `x86` versus `x64` . . . 29
2.9 Do not forget the library path . . . 31
2.10 Compatible compilers and installations for Gecode and Qt . . . 32

3.1 Linking against the driver . . . 40
3.2 Aborting execution . . . 42
3.3 How Gecode has been configured . . . 42
3.4 Which version of Gecode are we using? . . . 42

4.1 Do not use views for modeling . . . 44
4.2 Initializing variables . . . 44
4.3 Small variable domains are beautiful . . . 45
4.4 Reversing argument arrays . . . 50
4.5 Dynamically constructing models . . . 51
4.6 Different consistency levels have different costs . . . 55
4.7 Unsharing is expensive . . . 56
4.8 Boolean negation . . . 59
4.9 Shared integer arrays . . . 67
4.10 Shared arrays also provide STL-style iterators . . . 68
4.11 Failing a space . . . 79

5.1 Still do not use views for modeling . . . 81
5.2 Small variable domains are still beautiful . . . 83
5.3 Reification by decomposition . . . 84

6.1 Transcendental and trigonometric functions and constraints . . . 91
6.2 Still do not use views for modeling . . . 92
6.3 Small variable domains are still beautiful . . . 94
6.4 Weak propagation for strict inequalities ($<$, $>$) and disequality ($\neq$) . . . 96

7.1 Boolean precedences . . . 103
7.2 Reification of non-functional constraints . . . 104
7.3 Element for matrix can compromise propagation . . . 108
7.4 Cost must be assigned for solutions . . . 109
7.5 Creating a DFA only once . . . 110

8.1 Variables are re-selected during branching . . . 117
8.2 Do not try all values . . . 117
8.3 Using a member function as merit function . . . 126
8.4 Propagation is still explicit . . . 137

9.1 Search is indeterministic . . . 141
9.2 Values for c_d and a_d . . . 143
9.3 Be optimistic about parallel search . . . 144
9.4 Do not optimize by branching alone . . . 145
9.5 Number of threads for stop objects . . . 150
9.6 Controlling restart-based search with the commandline driver . . . 151
9.7 Controlling no-goods with the commandline driver . . . 160

12.1 Small variable domains are still beautiful . . . 183

15.1 Choose variables to avoid constraints . . . 197
15.2 Small variable domains are still beautiful . . . 198

22.1 Variables and views are passed by value . . . 265
22.2 Immediately return after subsumption . . . 274

23.1 Understanding `ES_NOFIX` . . . 290
23.2 View arrays also provide STL-style iterators . . . 293
23.3 View arrays have non-copying copy constructors . . . 294
23.4 `drop_fst()` and `drop_lst()` are efficient . . . 297

25.1 Narrow is dangerous . . . 314
25.2 Iterators must be increasing . . . 314

26.1 Different types of advisors for the same propagator . . . 329
26.2 Getting a propagator started with advisors . . . 330
26.3 Advisors and propagator obligations . . . 332
26.4 Advisor space requirements . . . 334

27.1 Using **`using`** clauses . . . 338
27.2 Boolean variables are not integer variables . . . 342

30.1 Keep the scope of a region small . . . 359

31.1 Never execute . . . 376

34.1 Correctness matters 398
34.2 Variable implementations must always be consistent 406

39.1 Printing information about alternatives. 453

1 Introduction

This document provides an introduction to modeling and programming with Gecode, an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications.

The hands-on, tutorial-style approach will get you started very quickly. The focus is on giving an overview of the key concepts and ideas required to model and program with Gecode. Each concept is introduced using concrete C++ code examples that are developed and explained step by step. This document is complemented by the complete Gecode reference documentation, as well as pointers to introductory and more advanced material throughout the text.

The first part of this document (Part M) is about *modeling* with Gecode. It explains modeling and solving constraint problems, and how to program, compile, link, and execute these models. This is complemented by a collection of interesting case studies of how to model with Gecode (Part C). The remaining, more advanced parts are about *programming* with Gecode: they explain how to use Gecode for implementing constraints (Part P), branchings (Part B), new variable types (Part V), and search engines (Part S).

1.1 What is Gecode?

Gecode is an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications. Gecode is:

open Gecode is radically open for programming: it can be easily interfaced to other systems. It supports the programming of new propagators (as implementation of constraints), branching strategies, and search engines. New variables can be programmed at the same level of efficiency as integer, set, and float variables that ship with Gecode.

free Gecode is distributed under the MIT license and is listed as free software by the FSF. All of its parts — including reference documentation, implementations of global constraints, and examples — are available as source code for download.

portable Gecode is implemented in C++ that rigidly follows the C++ standard. It can be compiled with modern C++ compilers and runs on a wide range of platforms.

accessible Gecode comes with complete tutorial and reference documentation that allows users to focus on different programming tasks with Gecode.

efficient Gecode offers excellent performance with respect to runtime, memory usage, and scalability. For example, Gecode won all gold medals in the MiniZinc Challenge in 2012, 2011, 2010, 2009, and 2008.

parallel Gecode complies with reality in that it exploits the multiple cores of today's commodity hardware for parallel search, giving an already efficient base system an additional edge.

alive Gecode has a sizeable user community and is being actively developed and maintained. In order to give you an idea: there has been a release every two to three month since the first release in December 2005.

1.2 What is this document?

We do not want to disappoint our readers, so let us get this out of the way as early as possible – here is *what this document is not*. This document is very definitely neither

- an introduction to constraint programming or modeling techniques, nor
- a collection of interesting implementations of constraints, nor
- an introduction to the architecture of constraint solvers, nor
- a reference documentation of the Gecode API.

The reader is therefore expected to have some background knowledge in constraint programming.

Furthermore, the document describes the C++ interface to Gecode, it is not about modeling with any of the available interfaces to Gecode.

Keeping it simple. Throughout this document, we will use simple examples to explain the concepts you have to understand in order to model and program with Gecode. However, these simple examples demonstrate the *complete array of techniques* that are sufficient to implement complex models, constraints, branchings, variables, and search engines. In fact, Gecode itself is based on the very same techniques – you will learn how to develop code that is just as good as (or maybe better than?) what Gecode itself provides.

Gecode architecture. This document follows the general architecure of Gecode, containing one part for each major component. Figure 1.1 gives an overview of the Gecode architecture. The kernel provides common functionality, upon which the modules for integer, set, and float constraints as well as the search engines are built. The colored boxes refer to the topics covered in this document.

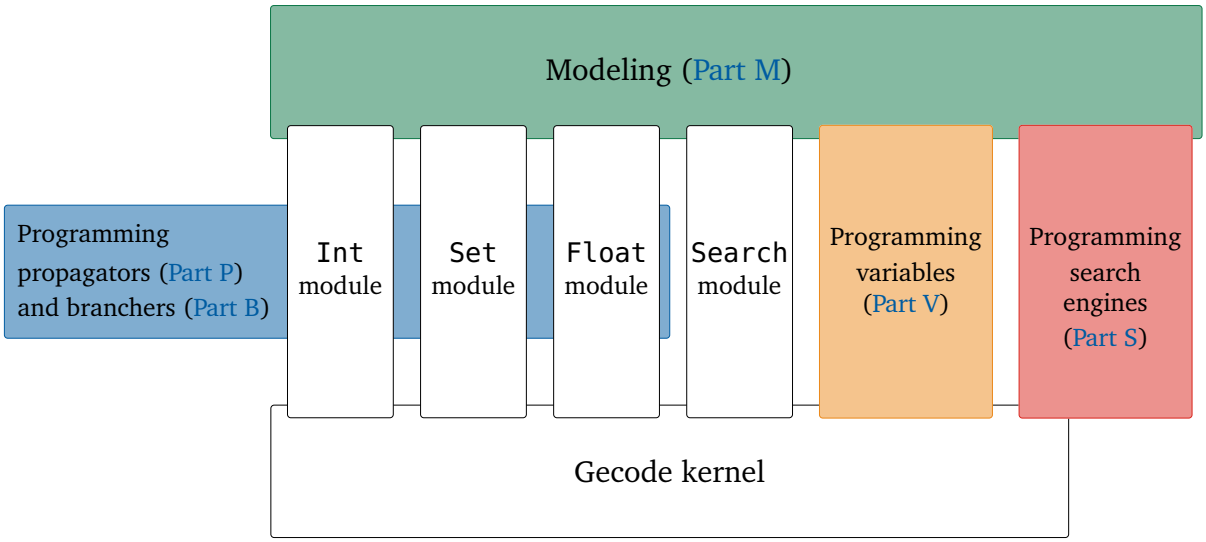


Figure 1.1: Gecode architecture

Modeling. The modeling part (Part M) of this document assumes some basic knowledge of modeling and solving constraint problems, as well as some basic C++ skills. The document restricts itself to simple and well known problems as examples. A constraint programming novice should have no difficulty to concentrate on the how-to-model with Gecode in particular, rather than the how-to-model with constraint programming in general.

The modeling part starts with a very simple constraint model that already touches on all the important concepts used in Gecode. There are detailed instructions how to compile and link the example code, so that you can get started right away. After that, the different variable types, the most important classes of constraints (including pointers to the Global Constraint Catalog [5], referred to by GCCAT) , and the infrastructure provided by Gecode (such as search engines) are presented.

Case studies. This document includes a collection of case studies in Part C. The case studies mix modeling and programming with Gecode. Some case studies are just interesting constraint models. Other case studies are constraint models that include the programming of new constraints and/or branchings.

Programming. The programming parts of this document require the same knowledge as the modeling part, plus some additional basic knowledge of how constraint propagation is organized. Section 22.2 provides pointers to recommended background reading.

The programming parts of this document cover the following topics:

programming propagators Part P describes in detail how new propagators (as implementations of constraints) can be programmed using Gecode. The part gives a fairly complete account of concepts and techniques for programming propagators.

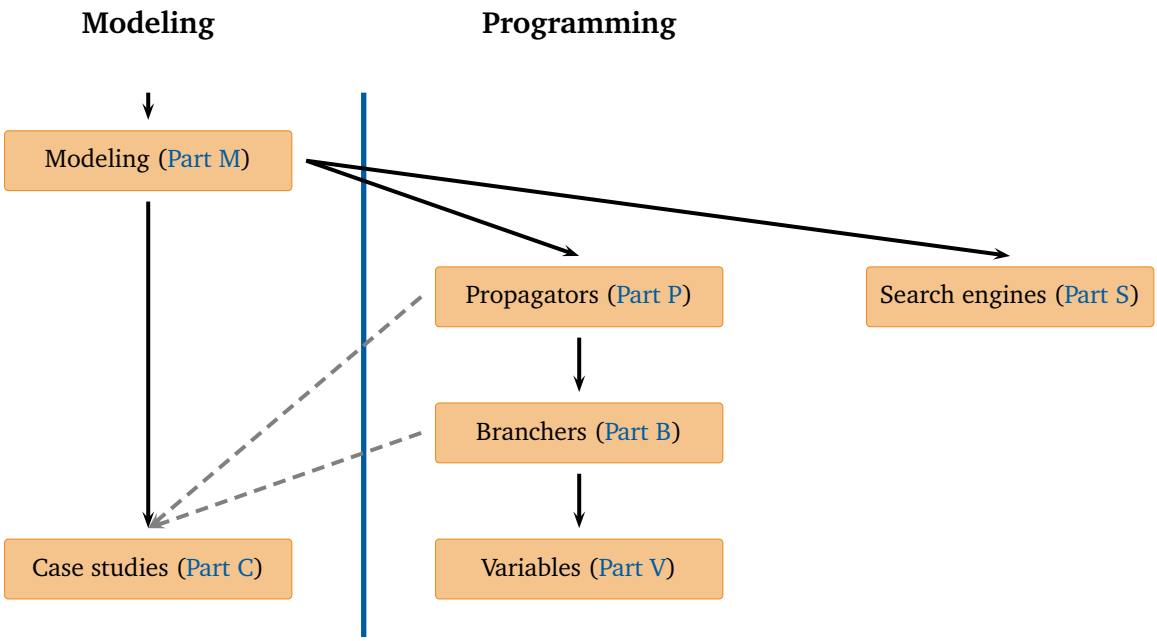


Figure 1.2: Dependencies among different parts of this document

programming branchers Part B describes how new branchers (as implementations of branchings for search) can be programmed using Gecode. This part is short and straightforward.

programming variables Gecode supports the addition of new variables types: the modules for integer, Boolean, set, and float variables use exactly the same programming interface as is available to any user. This interface is described in Part V.

programming search Part S describes how to program new search engines (Gecode comes with the most important search engines). The programming interface is simple yet very powerful as it is based on concurrency-enabled techniques such as recomputation and cloning.

1.3 How to read this document?

The dependencies among the different parts of this document are sketched in Figure 1.2. Every part starts with a short overview section and sketches what constitutes the basic material of a part. A part only requires the basic material of the parts it depends on.

The dashed arrows from programming propagators (Part P) and programming branchers (Part B) capture that some but not all case studies require knowledge on how to program

propagators and branchers. The individual case studies provide information on the required prerequisites.

Downloading example programs. All example program code used in this document is available for download, just click the download link in the upper right corner of an example.

Note that the code available for download is licensed under the same license as Gecode and not under the same license as this document. By this, you can use an example program as a starting point for your own programs.

If you prefer to download all example programs at once, you can do so here:

- example programs as gzipped tar archive
- example programs as 7z archive

1.4 Do I need to be a C++ wizard?

You very definitely do not have to be a C++ wizard to program Gecode models, some basic C++ knowledge will do. Modeling constraint problems typically involves little programming and tends to follow a common and simple structure that is presented in this document. It should be sufficient to have a general idea of programming and object-oriented programming.

Even programming with Gecode requires only basic C++ knowledge. The implementation of propagators, branchings, variables, and search engines follows simple and predictable recipes. However, this estimate refers to the aspects of using the concepts and techniques provided by Gecode. Of course, implementing truly advanced propagation algorithms inside a propagator will be challenging!

If you want to brush up your C++ knowledge, then brushing up your knowledge about the following topics might be most rewarding when using Gecode:

- Classes and objects, inheritance, virtual member functions: models are typically implemented by inheritance from a Gecode-provided base class.
- Overloading and operator overloading: post functions for constraints and support for posting symbolic expressions and relations rely on overloading (several functions with different argument types share the same function name).
- Exceptions: Gecode throws exceptions if post functions are used erroneously, for example, when numerical overflow could occur or when parameters are used inconsistently.
- Templates: while the modeling layer uses only few generic classes implemented as templates, programming with Gecode requires some basic knowledge about how to program with templates in C++.

Any textbook covering these topics should be well suited, for example, [26] for a general introduction to C++, and [67] for an introduction to C++ for Java programmers.

1.5 Can you help me?

Gecode has a lively and sizeable user community that can be tapped for help. You can ask questions about Gecode on the mailing list `users@gecode.org`. But, please make sure to not waste your time and the time of others:

- Please check this document and the Gecode reference documentation before asking a question.
- Please check the archive of the Gecode users mailing list as to whether a similar question has been asked before (note that the archive can be searched) before asking a question.
- Please focus on questions specific to Gecode. For general questions about constraint programming more suitable forums exist.
- Please provide sufficient detail: describe your platform (operating system, compiler, Gecode version) and your problem (what does not work, what do you want to do, what is the problem you observe) as accurately as you can.
- Please do not contact the developers for general Gecode questions, we will not answer. First, we insist on the benefit to the entire user community to see questions and answers (and the contribution to the mailing list archive). Second, more importantly, our users are known to have very good answers indeed. Remember, they – in contrast to the developers – might be more familiar with your user perspective on Gecode.
- Never ask for solutions to homework. The only more offensive thing you could do is to provide a solution on the mailing list if someone has violated the no homework policy!

1.6 Does Gecode have bugs?

Yes, of course! But, Gecode is very thoroughly tested (tests cover almost 100%) and extensively used (several thousand users). If something does not work, we regret to inform you that this is most likely due to your program and not Gecode. Again, this does not mean that Gecode has no bugs. But it does mean that it might be worth searching for errors in *your* program first.

Likewise, all major program fragments in this document (those that can be downloaded) have been carefully tested as well.

And, yes. Please take our apologies in advance if that somewhat bold claim does turn out to be false... If you have accepted our apologies, you can submit your bug report here.

1.7 How to refer to this document?

We kindly ask you to refer to the individual parts of this document with their respective authors (each part has a dedicated set of authors). BibTeX entries for the individual parts are available here.

If you refer to concepts introduced in Gecode, we kindly ask you to refer to the relevant academic publications.

1.8 Do you have comments?

If you have comments, suggestions, bug reports, wishes, or any other feedback for this document, please send a mail with your feedback to `mpg@gecode.org`.

Modeling

Christian Schulte, Guido Tack, Mikael Z. Lagerkvist

This part explains modeling and solving constraint problems, and how to program, compile, link, and execute constraint models.

Basic material. The basic material needed for modeling with Gecode is as follows:

- Chapter 2 (Getting started) provides an overview of how to program, compile, link, and execute a constraint model in Gecode.
- Chapter 3 (Getting comfortable) discusses functionality in Gecode that makes modeling and execution of models more convenient.
- The three first sections of Chapter 4 (Integer and Boolean variables and constraints) explain integer and Boolean variables (Section 4.1), variable and argument arrays (Section 4.2), and how constraints are posted (Section 4.3).
- The first section of Chapter 5 (Set variables and constraints) gives an overview of set variables (Section 5.1).
- The first section of Chapter 6 (Float variables and constraints) gives an overview of float variables (Section 6.2).
- The first sections of Chapter 8 (Branching) explain branching: basics (Section 8.1), branchings for integer and Boolean variables (Section 8.2), branchings for set variables (Section 8.3), and branchings for float variables (Section 8.4).
- Even though not strictly necessary for modeling, it is recommended to also read Section 9.1 and Section 9.2 that explain how search (and in particular parallel search) works in Gecode.

Part C features a collection of example models for Gecode as further reading.

Overview material. The remaining chapters and sections provide an overview of the available functionality for modeling and solving:

- Constraints on integer and Boolean variables are summarized in Section 4.4 and Section 4.5 of Chapter 4 (Integer and Boolean variables and constraints).
- Section 5.2 and Section 5.3 of Chapter 5 (Set variables and constraints) summarize constraints on set variables.
- Section 6.3 and Section 6.4 of Chapter 6 (Float variables and constraints) summarize constraints on float variables.
- Chapter 7 (Modeling convenience: MiniModel) provides an overview of modeling convenience implemented by MiniModel.
- The remaining sections of Chapter 8 (Branching) discuss more advanced topics for branchings: local versus shared variable selection (Section 8.5), random selection (Section 8.6), user-defined variable (Section 8.7)and value (Section 8.8)selection, tie-breaking (Section 8.9), branch filter functions (Section 8.11), assigning variables (Section 8.13), and executing code between branchers (Section 8.14).
- Section 9.3 of Chapter 9 (Search) summarizes how to use search engines.
- Chapter 10 (Gist) summarizes how to use Gist as a graphical and interactive search tool for developing constraint models.
- Chapter 11 (Script commandline driver) summarizes the commandline driver for Gecode models.

2 Getting started

This chapter provides a basic overview of how to program, compile, link, and execute a constraint model in Gecode. The chapter restricts itself to the fundamental concepts available in Gecode, the following chapter presents functionality that makes programming models more comfortable.

Overview. Section 2.1 explains the basics of how a model is programmed in Gecode. This is followed in Section 2.2 by a discussion of how search is used to find solutions of a model. How a model is compiled, linked, and executed is explained for several different operating systems in Section 2.3. Section 2.4 shows how Gist as a graphical and interactive search tool can be used for developing constraint models. Search for a best solution of a model is explained in Section 2.5.

The chapter also includes an explanation of how to obtain Gecode in Section 2.6 which covers both installation of binary packages available for some platforms and compilation of source packages. There is no need to say, that this section is very important reading!

2.1 A first Gecode model

Models in Gecode are implemented using *spaces*. A space is *home* to the *variables*, *propagators* (implementations of constraints), *branchers* (implementations of branchings, describing the search tree's shape, also known as labelings), and – possibly – an *order* determining a best solution during search.

Not surprisingly in an object-oriented language such as C++, an elegant approach to programming a model is by inheritance: a model inherits from the class `Space` (implementing spaces) and the subclass constructor implements the model. In addition to the constructor, a model must implement a copy constructor and a copy function such that search for that model works (to be discussed later).

Send More Money. The model we choose as an example is Send More Money: find distinct digits for the letters S, E, N, D, M, O, R, and Y such that the well-formed equation (no leading zeros) $SEND + MORE = MONEY$ holds.

The program (with some parts yet to be presented) is shown in Figure 2.1. Note that clicking a blue line starting with ► jumps to the corresponding code. Clicking [DOWNLOAD] in the upper right corner of the program provides access to the complete program text.

SEND MORE MONEY ≡ [DOWNLOAD]

```
#include <gecode/int.hh>
#include <gecode/search.hh>

using namespace Gecode;

class SendMoreMoney : public Space {
protected:
  IntVarArray l;
public:
  SendMoreMoney(void) : l(*this, 8, 0, 9) {
    IntVar s(l[0]), e(l[1]), n(l[2]), d(l[3]),
           m(l[4]), o(l[5]), r(l[6]), y(l[7]);
    ▶ NO LEADING ZEROS
    ▶ ALL LETTERS DISTINCT
    ▶ LINEAR EQUATION
    ▶ POST BRANCHING
  }
  ▶ SEARCH SUPPORT
  ▶ PRINT SOLUTION
};

▶ MAIN FUNCTION
```

Figure 2.1: A Gecode model for Send More Money

The program starts by including the relevant Gecode headers. To use integer variables and constraints, it includes `<gecode/int.hh>` and to access search engines it includes `<gecode/search.hh>`. All Gecode functionality is in the scope of the namespace `Gecode`, for convenience the program makes all functionality of the Gecode namespace visible by `using namespace Gecode`.

As discussed, the model is implemented as the class `SendMoreMoney` inheriting from the class `Space`. It declares an array `l` of integer variables and initializes this array to have `8` newly created integer variables as elements, where each variable in the array can take values from `0` to `9`. Note that the constructor for the variable array `l` takes the current space (that is, `*this`) as first argument. This is very common: any function that depends on a space takes the current space as argument (called *home space*) Examples are constructors for variables and variable arrays, functions that post constraints, and functions that post branchings.

To simplify the posting of constraints, the constructor defines a variable of type `IntVar` for each letter. Note the difference between creating a new integer variable (as done with creating the array of integer variables together with creating a new integer variable for each array element) and referring to the same integer variable through different C++ variables of type `IntVar`. This difference is discussed in more detail in Section 4.1.

Posting constraints. For each constraint there is a *constraint post function* that creates *propagators* implementing the constraint (in the home space that is passed as argument).

Tip 2.1 (`Space&` versus `Home`)**.** Actually, when you check the reference documentation, you will see that these functions do not take an argument of type `Space&` but of type `Home` instead. An object of type `Home` actually stores a reference to a space of type `Space&` (and a reference of type `Space&` is automatically coerced to an object of type `Home`). Additionally, a `Home` object might store other information that is useful for posting propagators and branchers. However, this is nothing you need to be concerned with when modeling with Gecode. Just think that `Home` reads as `Space&`. Using `Home` is important when programming propagators and branchers, see Section 22.5. ◀

The first constraints to be posted enforce that the equation is well formed in that it has no leading zeros:

```
NO LEADING ZEROS ≡
  rel(*this, s, IRT_NQ, 0);
  rel(*this, m, IRT_NQ, 0);
```

The family of `rel` post functions (functions with name `rel` overloaded with different argument types) implements simple relation constraints such as equality, inequalities, and disequality (see Section 4.4.3 and Simple relation constraints over integer variables). The constant `IRT_NQ` requests a disequality constraint.

All letters are constrained to take pairwise distinct values by posting a `distinct` constraint (also known as `alldifferent` constraint):

```
ALL LETTERS DISTINCT ≡
  distinct(*this, l);
```

See Section 4.4.7 and Distinct constraints for more information on the `distinct` constraint.

The constraint that $SEND + MORE = MONEY$ is posted as a linear equation where the individual letters are scaled to their appropriate decimal positions:

```
LINEAR EQUATION ≡
  IntArgs c(4+4+5); IntVarArgs x(4+4+5);
  c[0]=1000; c[1]=100; c[2]=10; c[3]=1;
  x[0]=s;    x[1]=e;   x[2]=n;  x[3]=d;
  c[4]=1000; c[5]=100; c[6]=10; c[7]=1;
  x[4]=m;    x[5]=o;   x[6]=r;  x[7]=e;
  c[8]=-10000; c[9]=-1000; c[10]=-100; c[11]=-10; c[12]=-1;
  x[8]=m;      x[9]=o;     x[10]=n;    x[11]=e;   x[12]=y;
  linear(*this, c, x, IRT_EQ, 0);
```

The `linear` constraint (which, again, exists in many overloaded variants) posts the linear equation (as instructed by `IRT_EQ`)

$$\sum_{i=0}^{|c|-1} c_i \cdot x_i = 0$$

with coefficients `c`, integer variables `x`, and right-hand side constant 0 (see Section 4.4.6 and Linear constraints over integer variables). Here, $|c|$ denotes the size (the number of elements) of the array `c` (which can be computed by `c.size()`). Post functions are designed to be as general as possible, hence the variant of `linear` that takes an array of coefficients and an array of integer variables as arguments. Other variants of `linear` exist that do not take coefficients (all coefficients are one) or accept an integer variable as the right-hand side instead of an integer constant.

Section 3.1 demonstrates additional support for posting linear expressions constructed from the usual arithmetic operators such as `+`, `-`, and `*`.

Posting branchings. Branchings determine the shape of the search tree. Common branchings take a variable array of the variables to be assigned values during search, a variable selection strategy, and a value selection strategy.

Here, we select the variable with a smallest domain size first (`INT_VAR_SIZE_MIN()`) and assign the smallest value of the selected variable first (`INT_VAL_MIN()`):

```
POST BRANCHING ≡
  branch(*this, l, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

A *branching* is implemented by a *brancher* (like a constraint is implemented by a propagator). A brancher creates a number of *choices* where each choice is defined by a number of *alternatives*. For example, the brancher posted above will create as many choices as needed

to assign all variables in the integer variable array `l`. Each of the choices is based on the variable selected by the brancher, say x, and the value selected by the brancher, say n. Then the alternatives of a choice are $x = n$ and $x \neq n$ and are tried by search in that order.

A space can have several branchers, where the brancher that is posted first is also used first for search. More information on branchings can be found in Chapter 8.

Search support. As mentioned before, a space must implement an additional `copy()` function that is capable of returning a fresh copy during search. Search in Gecode is based on a hybrid of *recomputation* and *cloning* (see Chapter 9). Cloning during search relies on the capability of a space to create a copy of itself.

To avoid confusion, by *cloning* we refer to the entire process of creating a clone of a space. By *copying*, we refer to the creation of a copy of a particular object during cloning, for example, a variable or a space.

SEARCH SUPPORT ≡
```
SendMoreMoney(bool share, SendMoreMoney& s) : Space(share, s) {
  l.update(*this, share, s.l);
}
virtual Space* copy(bool share) {
  return new SendMoreMoney(share,*this);
}
```

The actual `copy()` function is straightforward and uses an additional copy constructor. The `copy()` function is virtual such that cloning (used on behalf of a search engine) can create a copy of a space even though the space's exact subclass is not known to cloning. The Boolean argument `share` defines whether a shared copy is constructed and is of no further interest here.[1]

The obligation of the copy constructor is to invoke the copy constructor of the parent class, and to copy all data structures that contain variables. For `SendMoreMoney` this amounts to invoking `Space(share,s)` and updating the variable array. An exception of type `SpaceNotCloned` is thrown if the copy constructor of the `Space` class is not invoked. Please keep in mind that the copy constructor is run on the copy being created and is passed the space that needs to be copied as argument. Hence, updating the variable array `l` in the copy copies the array `s.l` from the space `s` being cloned (including all variables contained in the array). More on updating variables and variable arrays can be found in Section 4.1.8.

Printing solutions. Finally, the following prints the variable array `l`:

[1] For the curious reader: a shared (`share` is **`true`**) clone can only be used within the same thread. A clone without sharing (`share` is **`false`**) creates a clone where no data structures are shared between original and clone. Hence both original and cloned space can be used in different threads without problems. The appropriate value for `share` is passed to the `clone()` member function of a space by a search engine, see also Section 39.1.

PRINT SOLUTION ≡
```
void print(void) const {
  std::cout << l << std::endl;
}
```

In a real application, one would use the solution in some other parts of the program. The point is that the space acts as a *closure* for the solution variables: the space maps member names to objects. The space for an actual solution is typically different from the space created initially. This is due to the fact that search for a solution returns a space that has been obtained by constraint propagation and cloning. The space members that refer to the solution variables (the member `l` in our example) provide the means to access a solution independent of a particular space.

2.2 Searching for solutions

Let us assume that we want to search for all solutions and that search is controlled by the main function of our program. Search consists of two parts:

- create a model and a search engine for that model; and
- use the search engine to find all solutions.

Hence, our main function looks as follows:

MAIN FUNCTION ≡
```
int main(int argc, char* argv[]) {
  ▶ CREATE MODEL AND SEARCH ENGINE
  ▶ SEARCH AND PRINT ALL SOLUTIONS
  return 0;
}
```

Creating a model is almost obvious: create an object of the subclass of `Space` that implements the model. Then, create a search engine (we will be using a search engine `DFS` for depth-first search) and initialize it with a model. Search engines are generic with respect to the type of model, implemented as a template in C++. Hence, we use a search engine of type `DFS<SendMoreMoney>` for the model `SendMoreMoney`.

When the engine is initialized, it takes a clone of the model passed to it (`m` in our example). As the engine takes a clone, several engines can be used without recreating the model. As we are interested in a single engine, we immediately delete the model `m` after the search engine has been initialized.

CREATE MODEL AND SEARCH ENGINE ≡
```
SendMoreMoney* m = new SendMoreMoney;
DFS<SendMoreMoney> e(m);
delete m;
```

Tip 2.2 (Propagation is explicit)**.** A common misconception is that constraint propagation is performed as soon as a space is created or as soon as a constraint is posted. Executing the following code

```
SendMoreMoney* m = new SendMoreMoney;
m->print();
```

prints

```
{[1..9], [0..9], [0..9], [0..9], [1..9], [0..9], [0..9], [0..9]}
```

That is, only very simple and cheap propagation (nothing but modifying the domain of some variables) has been performed.

Constraint propagation is *explicit* and must be requested by the `status()` member function of a space (the function also returns information about the result of propagation but this is of no concern here). Requesting propagation by

```
(void) m->status();
m->print();
```

prints

```
{9, [4..7], [5..8], [2..8], 1, 0, [2..8], [2..8]}
```

◀

A search engine first performs constraint propagation as only spaces that have been propagated can be cloned (so as to not duplicate propagation for the original and for the clone).

The `DFS<SendMoreMoney>` search engine has a simple interface: the engine features a `next()` function that returns the next solution or `NULL` if no more solutions exist. As we are interested in all solutions, a while loop iterates over all solutions that are found by the search engine:

SEARCH AND PRINT ALL SOLUTIONS ☰

```
while (SendMoreMoney* s = e.next()) {
  s->print(); delete s;
}
```

As you can see, a solution is nothing but a model again. A search engine ensures that constraint propagation is performed and that all variables are assigned as described by the branching(s) of the model passed to the search engine. When a search engine returns a model, the responsibility to delete the solution model is with the client of the search engine.

It is straightforward to see how one would search for a single solution instead: replace **`while`** by **`if`**. `DFS` is but one search engine and the behavior of a search engine can be configured (for example: how cloning or recomputation is used; how search can be interrupted) and it can be queried for statistical information. Search engines are discussed in more detail in Chapter 9.

Tip 2.3 (Catching Gecode exceptions)**.** Posting constraints, posting branchings, creating variables, and so on with Gecode might throw exceptions (for example, potential numerical overflow, illegal use of arguments). It is good practice to construct your programs right from the start to catch all these exceptions.

That is, you should wrap the entire body of the main function but the **`return`** statement into a **`try`** statement as follows:

```
try {
  ...
} catch (Exception e) {
  std::cerr << "Gecode exception: " << e.what() << std::endl;
  return 1;
}
return 0;
```

Even though this is good practice, the example programs in this document do not follow this advice, so as to keep the programs more readable. ◀

2.3 Compiling, linking, and executing

This section assumes that you have Gecode with the right version (this document uses Gecode 4.3.2) already installed on your computer. If that is not the case, you should visit the download section on Gecode's website and read Section 2.6 on how to install and/or compile Gecode.

Naturally, the following sections are platform-specific.

2.3.1 Microsoft Visual Studio

Gecode uses a technique called auto-linking with Visual Studio: by including Gecode header files, the compiler/linker knows which library and DLL must be linked against. Hence, it is sufficient to provide information to the compiler where the libraries reside.

The library and DLL names encode the Gecode version, the platform (`x86`, `x64`, or `ia64`), and whether Gecode has been built as release (optimized) or debug. That has the advantage that a single set of header files can be shared among different platforms and builds and the appropriate libraries and DLLs are selected automatically.

In the following we assume that you are using one of the pre-compiled Windows packages we provide (see Section 2.6.1). The packages define the environment variable `%GECODEDIR%` where Gecode has been installed, and update the `Path` environment variable so that Gecode DLLs are found for execution.

Commandline. In the following we assume that you use the Visual Studio Command Prompt. When compiling and linking with `cl`, you have to take the following into account:

- As Gecode uses exceptions, you have to add `/EHsc` as option on the commandline.
- You have to link dynamically against multithreaded libraries. That is, you have to add to the commandline either `/MD` (release build) or `/MDd` (debug build).
- If you want a release build, you need to switch off assertions by defining `/DNDEBUG`.
- You should instruct the compiler `cl` to search for the Gecode header files by adding `/I"%GECODEDIR%\include"` as an option.
- When using `cl` for linking, you should add at the very end of the commandline: `/link /LIBPATH:"%GECODEDIR%\lib"`.
- By default, `cl` warns if **`this`** is used in an initializer list (Gecode uses this for the initialization of variables and variable arrays). You can suppress the warning by passing `/wd4355`.

The full command for compiling `send-more-money.cpp` as a release build (including optimization with `/Ox`) is

```
cl /DNDEBUG /EHsc /MD /Ox /wd4355 -I"%GECODEDIR%\include" \
  -c -Fosend-more-money.obj -Tpsend-more-money.cpp
```

where the \ at the end of a line means that the line actually continues on the next line. The following command links the program:

```
cl /DNDEBUG /EHsc /MD /Ox /wd4355 -I"%GECODEDIR%\include" \
  -Fesend-more-money.exe send-more-money.obj \
  /link /LIBPATH:"%GECODEDIR%\lib"
```

Integrated development environment. When your Microsoft Visual Studio solution uses Gecode, all necessary settings can be configured in the properties dialog of your solution. We assume that Gecode is installed in the default location `"<GECODEDIR>"`.[2]

- You must use dynamic linking against a multithreaded library. That is, either `/MD` (release build) or `/MDd` (debug build). Depending on whether `/MD` or `/MDd` is used, release or debug libraries and DLLs will be used automatically.
- As Gecode uses exceptions, you have to enable `/EHsc` as option (this is true by default).
- If you want a release build, you have to switch off assertions by defining `/DNDEBUG` (this is true by default).
- Configuration Properties, C++, General: set the "Additional Include Directories" to include `"<GECODEDIR>\include"` as the directory containing the Gecode header files.

[2]Unfortunately, the development environment does not resolve `%GECODEDIR%` automatically. So you have to replace `"<GECODEDIR>"` in the following by the path where Gecode has been installed.

- Configuration Properties, Linker, General: set the "Additional Library Directories" to `"<GECODEDIR>\lib"` as the path containing the libraries.

Tip 2.4 (Cygwin with Microsoft Visual Studio)**.** A setup that works well for us is to install Cygwin and Microsoft Visual Studio (Microsoft distributes a free Express Edition).

The easiest way to use the Microsoft Visual Studio C++ compiler (`cl`) from Cygwin is to add a line that loads the file `vcvarsall.bat` to the `Cygwin.bat` file that starts Cygwin. The file `vcvarsall.bat` comes with Microsoft Visual Studio (for example, it is used to start the Visual Studio Command Prompt). On my machine using Visual C++ 2008 Express Edition, the added line is

```
call "c:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"
```

Or, you start the Visual Studio Command Prompt first, and then start the bash shell by hand with

```
chdir c:\Cygwin\bin
bash --login -i
```

provided that Cygwin is installed in `c:\Cygwin`. ◀

2.3.2 Apple Mac OS

These compilation instructions assume that you use the Gecode binary package for Mac OS (see Section 2.6.1). If you compiled and installed Gecode from the source package, please read Section 2.3.3.

Commandline. When compiling your code using the `gcc` compiler (invoking it as `g++`), the Gecode header files are found automatically, they are on the default header search path. For linking against the Gecode framework, just add `-framework gecode` as option. Note that only versions 4.2 or better of `gcc` are supported.

The following command compiles and links `send-more-money.cpp` as a release build (including optimization):

```
g++ -O3 -c send-more-money.cpp
g++ -framework gecode -o send-more-money send-more-money.cpp
```

XCode. You can easily use Gecode within the XCode development environment by choosing *Add* ▷ *Existing Frameworks...* from the context menu on your target. Then pick the `gecode` framework from the list. You may have to edit your project settings to choose *Mac OS 10.6* as the base SDK.

2.3.3 Linux and relatives

On Linux (and similar operating systems), Gecode is installed as a set of shared libraries. The default installation directory is `"/usr/local"`, which means that the header files can be found in `"/usr/local/include"` and the libraries in `"/usr/local/lib"`. Depending on your Linux distribution, a binary package may have been installed under the `"/usr"` path instead. If you installed Gecode from the source package, you may have chosen a different installation directory. For now, assume that Gecode is installed in `"<dir>"`.

Commandline. To compile your code using the `gcc` compiler, you have to add the option `-I<dir>/include` so that `gcc` can find the header files. Note that only versions 4.2 or better of `gcc` are supported.

For linking, the path has to be given as `-L<dir>/lib`, and in addition the individual Gecode libraries must be linked. You always have to link against the support and kernel libraries, using `-lgecodesupport -lgecodekernel`. For the remaining libraries, the rule of thumb is that if you include a header file `<gecode/FOO.hh>`, then `-lgecodeFOO` must be given as a linker option. For instance, if you use integer variables and include `gecode/int.hh`, you have to link using `-lgecodeint`.

Some linkers require the list of libraries to be sorted such that libraries appear before all libraries they depend on. In this case, use the following order (and omit libraries you don't use):

1. `-lgecodeflatzinc`
2. `-lgecodedriver`
3. `-lgecodegist`
4. `-lgecodesearch`,
5. `-lgecodeminimodel`
6. `-lgecodeset`
7. `-lgecodefloat`
8. `-lgecodeint`
9. `-lgecodekernel`
10. `-lgecodesupport`

A complete example for compiling and linking the file `send-more-money.cpp` is as follows.

```
g++ -I<dir>/include -c send-more-money.cpp
g++ -o send-more-money -L<dir>/lib send-more-money.o \
  -lgecodesearch -lgecodeint -lgecodekernel -lgecodesupport
```

The \ at the end of a line means that the line actually continues on the next line.

In order to run programs that are linked against Gecode, the Gecode libraries must be found on the library path. They either have to be installed in one of the default locations (such as `/usr/lib`), or the environment variable `LD_LIBRARY_PATH` has to be set to include `<dir>/lib`.

Eclipse development environment. If you use the Eclipse IDE with the CDT (C/C++ development tools), you have to configure the paths to the Gecode header files and libraries.

In the *Project* menu, select the *Properties* dialog. Under *GCC C++ Compiler*, add `<dir>/include` to the *Directories*. Under *GCC C++ Linker*, add `<dir>/lib` to the *Library search path*, and the Gecode libraries you have to link against to the *Libraries* field.

In order to run programs that link against Gecode from within the Eclipse CDT, select *Open Run Dialog* from the *Run* menu. Either add a new launch configuration, or modify your existing launch configuration. In the *Environment* tab, add the environment variable `LD_LIBRARY_PATH=<dir>/lib`.

Tip 2.5 (Eclipse on Windows and Mac OS)**.** If you use Eclipse on Windows or Mac OS, the procedure should be similar, except that you do not have to add the environment variable to the launch configuration, and on Windows you do not need to specify the libraries to link against. ◀

2.4 Using Gist

When developing a constraint model, the usual outcome of a first modeling attempt is that the model has no solutions or searching for a solution takes too much time to be feasible. What one really needs in these situations is additional insight as to: why does the model have no solutions, why is propagation not sufficient, or why is the branching not appropriate for the problem?

Gecode offers Gist as a graphical and interactive search tool with which you can explore any part of the search tree of a model step by step or automatically and inspect the nodes of the search tree.

Using Gist is absolutely straightforward. Figure 2.2 shows how Gist is used for the Send More Money problem. As before, a space `m` for the model is created. This space is passed to Gist, where Gist is instructed to work in `dfs` (depth-first search) mode. The call to `Gecode::dfs` terminates only after Gist's window is closed.

Figure 2.3 shows two screenshots of Gist. The left-hand side shows how Gist starts (with no node of the tree yet explored). The right-hand side shows the fully explored search tree of Send More Money.

Gist is so intuitive that our recommendation is to just play a little with it. If you want to know more about Gist, consult Chapter 10.

One additional feature of Gist that comes in handy when developing constraint models is to inspect nodes of the search tree by double-clicking them. Figure 2.4 shows a modified

SEND MORE MONEY WITH GIST ≡ [DOWNLOAD]

```
#include <gecode/int.hh>
#include <gecode/gist.hh>

using namespace Gecode;

class SendMoreMoney : public Space {
  ...
};

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  Gist::dfs(m);
  delete m;
  return 0;
}
```

Figure 2.2: Using Gist for Send More Money

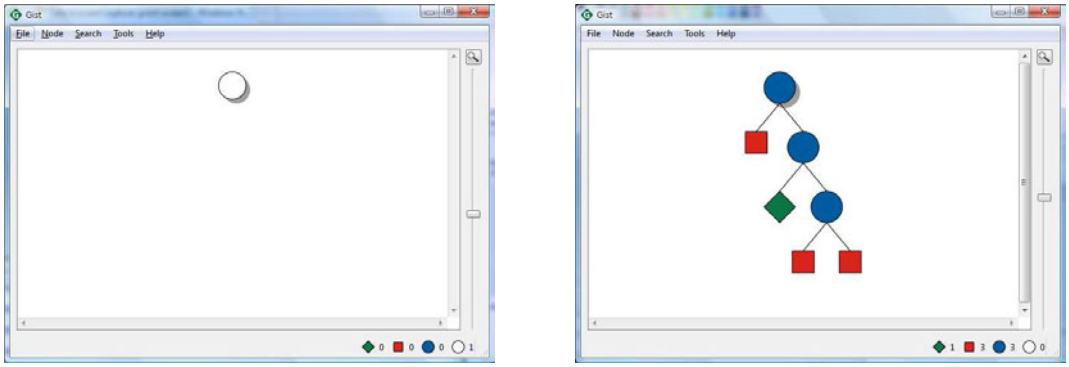


Figure 2.3: Gist screen shots

SEND MORE MONEY WITH GIST INSPECTION ≡ [DOWNLOAD]

```
#include <gecode/int.hh>
#include <gecode/gist.hh>
...
class SendMoreMoney : public Space {
  ...
  void print(std::ostream& os) const {
    os << l << std::endl;
  }
};

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  Gist::Print<SendMoreMoney> p("Print solution");
  Gist::Options o;
  o.inspect.click(&p);
  Gist::dfs(m,o);
  delete m;
  return 0;
}
```

Figure 2.4: Using Gist for Send More Money with node inspection

```
SEND MOST MONEY ≡                                                [DOWNLOAD]
  ...
  class SendMostMoney : public Space {
    ...
    ▶ CONSTRAIN FUNCTION
  };

  ▶ MAIN FUNCTION
```

Figure 2.5: A Gecode model for Send Most Money finding a best solution

program that instructs Gist to use the `print()` function of `SendMoreMoney` whenever a node is double-clicked. Note that the print function has been changed to take a standard outstream to print on as argument.

Tip 2.6 (Gist scales)**.** Do not be afraid to use Gist even on large problems. You can expect that per Gigabyte of main memory, Gist can maintain around eight to ten million nodes. And the runtime overhead is low (in our experiments, around 15% compared to the commandline search engine using one thread). Just be sure to increase the display refresh rate for larger trees (see Section 10.3.6). ◀

Section 3.3 explains how to use a commandline driver that supports to execute the same constraint model with different search engines (for example, `DFS` or `Gist`) by passing options on the commandline.

Tip 2.7 (Linking against Gist)**.** As discussed in Section 2.3, when you use Gist on a platform (Linux and relatives) that requires to state all libraries to link against, you also have to link against the library for Gist (that is, for Linux and relatives by adding `-lgecodegist` to the compiler options on the commandline). ◀

2.5 Best solution search

The last aspect to be discussed in this chapter is how to search for a best solution. We are using a model for Send Most Money as an example: find distinct digits for the letters S, E, N, D, M, O, T, and Y such that the well-formed equation (no leading zeros) $SEND + MOST = MONEY$ holds and that $MONEY$ is maximal.

Searching for a best solution requires a best solution search engine and a function that constrains a space to yield a better solution. A Gecode model for Send Most Money is shown in Figure 2.5. The model differs from Send More Money only by using a different linear equation and the additional `constrain()` function.

Assume a new solution, say `b`, is found during best solution search: on the current search node `s` (a space) the member function `constrain()` is called and the so-far best solution

b is passed as argument (that is, s.constrain(b) is executed). The constrain() member function must add a constraint to s such that s can only yield a better solution than b during search. For Send Most Money, the constrain() member function is as follows:

CONSTRAIN FUNCTION ≡
```
virtual void constrain(const Space& _b) {
  const SendMostMoney& b = static_cast<const SendMostMoney&>(_b);
  IntVar e(l[1]), n(l[2]), m(l[4]), o(l[5]), y(l[7]);
  IntVar b_e(b.l[1]), b_n(b.l[2]), b_m(b.l[4]),
         b_o(b.l[5]), b_y(b.l[7]);
  int money = (10000*b_m.val()+1000*b_o.val()+100*b_n.val()+
               10*b_e.val()+b_y.val());
  IntArgs c(5); IntVarArgs x(5);
  c[0]=10000; c[1]=1000; c[2]=100; c[3]=10; c[4]=1;
  x[0]=m;     x[1]=o;    x[2]=n;   x[3]=e;  x[4]=y;
  linear(*this, c, x, IRT_GR, money);
}
```

First, the integer value of money in the so-far best solution is computed from the values of the variables. Note that the search engine does not know what model it searches a solution for. The search engine passes a space _b that the constrain member function must cast into a SendMostMoney space. Then the constraint is added that a better solution must yield more money.

Using a best solution search engine. The main function now uses a branch-and-bound search engine rather than a plain depth-first engine:

MAIN FUNCTION ≡
```
int main(int argc, char* argv[]) {
  SendMostMoney* m = new SendMostMoney;
  BAB<SendMostMoney> e(m);
  ...
}
```

The loop that iterates over all solutions found by the branch-and-bound search engine is exactly the same as before. That means that solutions are found and printed with an increasing value of $MONEY$. The best solution is printed last.

The branch-and-bound engine BAB (see also Section 9.3) calls the constrain() member function defined by the model. Note that every space defines a default constrain() member function (to keep the design of models simple). If a model does not re-define the constrain() member function (either directly or indirectly bu inheriting a constrain() function), the default function will do nothing.

Using Gist for best solution search is straightforward. Instead of using Gist::dfs, one uses Gist::bab to put Gist into branch-and-bound mode.

In Section 3.2 it is discussed how a simple cost() function can be used for best solution search instead of a more general constrain() function.

2.6 Obtaining Gecode

This section explains how to obtain Gecode. There are two basic options: install Gecode as a binary package (explained in Section 2.6.1) or compile Gecode from its source (explained in Section 2.6.2 and Section 2.6.3).

We recommend to use the pre-compiled binaries we provide, unless you like to tinker. The advantage of the packages we provide is that required and additional software for those platforms not having automatic package management (for example, Microsoft Windows) and reference documentation in platform-specific format are already included.

2.6.1 Installing Gecode

Naturally the following sections are platform specific.

Installing Gecode on Windows

The pre-compiled Gecode binaries available for download require Microsoft Visual Studio. Note that Visual Express Editions are available free of charge from Microsoft.

Pre-compiled binaries are available for several versions of Microsoft Visual C++. The binary packages include everything needed (in particular, Qt for Gist).

Tip 2.8 (Compiling on Windows for `x86` versus `x64`)**.** The download section on Gecode's website offers Windows packages for two different platforms: one for `x86` (32 bit) and one for `x64` (64 bit). When downloading and installing one of these packages you should make sure that the package's platform matches the compiler you are using (the Windows platform does not matter). Some freely available Express Editions of Visual Studio only support `x86`. ◀

Installing Gecode on Apple Mac OS

The pre-compiled Gecode binaries for Mac OS require the XCode developer tools, version 3.1 or higher. XCode is available from the Mac OS install DVD, or from the Apple Mac Dev Center.

The binary packages include the Qt library (necessary for Gist).

After installing these prerequisites, simply download and open the Gecode disk image for Mac OS, double-click the installer package, and follow the instructions.

Installing Gecode on Linux and relatives

The Debian and Ubuntu Linux distributions come with pre-compiled packages for Gecode. These packages (and all the packages they depend on) can be installed with the usual package management tools (for example, `apt-get`). Note that we do not maintain these packages, and that the repositories do not always provide the most up-to-date version.

If the Linux distribution of your choice does not provide a binary package for Gecode, or does not contain the latest version, please refer to the next section for instructions how to compile Gecode from the source.

2.6.2 Compiling Gecode

Gecode can be built on all recent versions of Windows, Linux, and MacOS X. Porting to other Unix flavors should be easy, if any change is necessary at all. The Gecode source code is available from the download section of the Gecode web site.

Prerequisites. In order to compile Gecode, you need a standard Unix toolchain including the following programs: a bash-compatible shell, GNU make, sed, cp, diff, tar, perl, grep.

These are available in all standard installations of Linux. On MacOS X, you need to install the XCode developer tools, version 3.1 or higher. XCode is available from the Mac OS install DVD, or from the Apple Mac Dev Center. For Windows, we require the Cygwin environment that provides all necessary tools (see also Tip 2.4).

We currently support:

- Microsoft Visual C++ compilers for Windows. The Microsoft Visual C++ Express Edition is available free of charge from Microsoft.
- GNU Compiler Collection (gcc) for Unix flavors such as Linux and MacOS X. The GNU gcc is open source software and available from the GCC home page. It is included in all Linux distributions and the Apple MacOS X developer tools.

  Important: Gecode requires at least version 4.2 of `gcc`.
- Intel C++ compiler, although we do not test the binaries produced by it.

Configuring the sources. After unpacking the sources, you need to run the `configure` script in the toplevel directory. This script (which uses GNU autoconf) acquires information about the system you try to compile Gecode on.

To setup Gecode for your particular system, you may need to add one or more of the following options to `configure`:

- To install Gecode somewhere else than the default `/usr/local`, please use the commandline `--prefix=[···]` switch.
- You can enable and disable the individual modules Gecode consists of by using `--enable-[MODULE]` and `--disable-[MODULE]`.

You can get a list of all supported configuration options by calling `configure` with the `--help` switch. Section 2.6.3 explains the configuration options in more detail, if the defaults do not work for you.

Compiling the sources. After successful configuration, simply invoking

```
make
```

in the toplevel Gecode directory will compile the whole library and the examples.

Installation. After a successful compilation, you can install the Gecode library and all header files necessary for compiling against it by invoking

```
make install
```

in the build directory.

Tip 2.9 (Do not forget the library path)**.** In order to run programs that are linked against Gecode (such as the Gecode examples), the libraries must be found on the library path. See Section 2.3.3 for details. ◀

Running the examples. After compiling the examples, they can be run directly from the command line. For instance, try the Golomb Rulers Problem:

```
./examples/golomb
```

or (when running Windows):

```
./examples/golomb.exe
```

On some platforms, you may need to set environment variables like `LD_LIBRARY_PATH` (Linux) or `DYLD_LIBRARY_PATH` (Mac OS) to the toplevel compile directory or the installation directory (where the dynamic libraries are placed after compilation).

Compilation with Gist. The Gecode Interactive Search Tool (Gist) is a graphical search engine for Gecode, built on top of the Qt GUI toolkit.

In order to compile Gecode with Gist, you need an installation of the Qt library including the development header files. The Qt binary packages for Windows and Mac OS (available from the Nokia Qt web pages) as well as the Qt packages in the usual Linux distributions contain everything that is necessary. You can however also compile Qt from its sources, available from Nokia under both free and commercial licenses.

Please note that if you develop closed-source software with Gecode and Gist, you will have to either comply with the LGPL, or obtain a commercial license for Qt from Nokia!

If you are developing on Windows using the Microsoft Visual C++ compiler, make sure to compile the Qt library with the same compiler.

After installing Qt, make sure that the `qmake` tool is in your shell path. The Gecode `configure` script will then detect the presence of Qt and automatically compile with support for Gist.

Tip 2.10 (Compatible compilers and installations for Gecode and Qt)**.** Please make sure that the compiler with which Qt has been compiled is compatible with the compiler you intend to use for Gecode (most likely, the requirement is that both packages must be compiled with the very same compiler).

In particular, make sure that this is true when you install a Qt binary package. Watch out for pre-installed Qt packages! For example: the Qt packages on Windows available through Cygwin should be disabled or deinstalled when you want to use Qt and Gecode with the Microsoft Visual C++ compiler. ◀

Compilation with support for trigonometric and transcendental float constraints. By default, trigonometric and transcendental float constraints are *disabled* (see also Tip 6.1). To enable them, Gecode must be configured to use the GMP (or MPIR instead) and MPFR libraries. After having installed the libraries, the following commandline options instruct `configure` to use these packages:

- `-with-gmp-include=···`: the directory where the header files of GMP or MPIR are installed.
- `-with-gmp-lib=···`: the directory where the library files of GMP or MPIR are installed.
- `-with-mpfr-include=···`: the directory where the header files of MPFR are installed.
- `-with-mpfr-lib=···`: the directory where the library files of MPFR are installed.

2.6.3 Advanced configuration and compilation

If the instructions from the previous section do not work for your system, please have a look at the following example configurations and advanced options to configure Gecode to your needs.

Example configurations

To compile only the Gecode library **without examples** on a Unix machine, use

```
./configure --disable-examples
```

To compile on a **Unix machine** using a different than the default `gcc` compiler, and install under `/opt/gecode`, use

```
./configure --prefix=/opt/gecode CC=gcc-4.2 CXX=g++-4.2
```

To compile a **debug build** on Unix, turning on all assertions and not inlining anything, use

```
./configure --enable-debug
```

To compile on a system using a different than the default compiler, and a `/bin/sh` that is not bash compatible (for example, on a **Solaris machine**), use

```
./configure --with-host-os=linux \
      CC="gcc-4.2" CXX="g++-4.2" \
      SHELL="/bin/bash"
make SHELL="/bin/bash"
```

You can compile as **universal binary** on a Mac OS machine. Configure with

```
./configure --with-architectures=i386,ppc
```

For building universal binaries on a PowerPC machine, you have to supply the path to the universal SDK (which is the default on Intel based Macs):

```
./configure --with-architectures=i386,ppc \
      --with-sdk=/Developer/SDKs/MacOSX10.4u.sdk
```

Passing options for compilation. Additional options for compilation can be passed to the compiler from the make commandline via the variable `CXXUSR`. For example, to pass to `gcc` the additional option `-mtune=i686` the following can be used:

```
make CXXUSR="-mtune=i686"
```

Compiling in a separate directory. The Gecode library can be built in a separate directory. This is useful if you do not want to clutter the source tree with all the object files and libraries.

Configuring Gecode in a separate directory is easy. Assume that the sources can be found in directory `$GSOURCEDIR`, change to the directory where you want to compile Gecode and call

```
$GSOURCEDIR/configure [options]
```

This will generate all necessary files in the new build directory.

Dependency management. The dependencies between source files are not handled automatically. If you are using a Gecode version from our subversion repository or if you modified any of the source files, you will have to call `make depend` before compilation in order to determine the source dependencies.

Dependency management is only needed for recompiling Gecode after changing something. In an unmodified version (or after a `make clean`) all files are compiled anyway.

Compiling for unsupported platforms. If you want to try compiling Gecode on a platform that we do not mention, you can override the platform tests during `configure`. There are two options to specify the type of platform:

- `--with-host-os=[linux|darwin|windows]`
- `--with-compiler-vendor=[gnu|microsoft]`

Using the first option, you can state that your platform should behave like Linux, Darwin (which is actually BSD), or Windows. This affects mainly the filenames and the tools used to generate shared and static libraries.

The second option says that your compiler can be used very much like the gnu compiler `gcc`, or the Microsoft compiler `cl`. Please let us know of any successful attempt at compiling Gecode on other platforms.

Useful Makefile targets

The main Gecode Makefile supports the following useful targets:

- `all` compiles all parts of the library that were enabled during `configure`, and the examples if enabled.
- `install` installs library, headers and examples (if enabled) into the `prefix` given at `configure`.
- `clean` removes object files.
- `veryclean` removes object files, libraries, and all files generated during `make`.
- `distclean` removes object files, libraries, and all generated files.
- `depend` generates dependencies between source files.
- `test` compiles the test suite.
- `doc` generates the reference documentation using doxygen.
- `installdoc` installs the documentation.
- `distdoc` creates tgz and zip archives of the documentation.

3 Getting comfortable

This chapter provides an overview of some functionality in Gecode that makes modeling and execution of models more convenient.

Overview. Expressions constructed from standard arithmetic operators for posting linear constraints are discussed in Section 3.1, cost functions for best solution search are discussed in Section 3.2, and a script commandline driver that supports the most common options for running models from the commandline is discussed in Section 3.3.

3.1 Posting linear constraints de-mystified

As mentioned in the previous chapter, Gecode comes with simple modeling support for posting constraints defined by linear expressions and relations. The parts of the program for Send More Money from Figure 2.1 that change are shown in Figure 3.1. In order to use the modeling support, we have to include the MiniModel header.

The MiniModel module also supports Boolean expressions and relations, and much more, see Chapter 7 for more information. The module in itself does not implement any constraints. The function `rel` takes the description of the linear constraint, analyzes it, and posts a linear constraint by using the same `linear` function we have been using in Section 2.1.

3.2 Using a cost function

Figure 3.2 uses the class `IntMaximizeSpace` for cost-based optimization for Send Most Money. The class `IntMaximizeSpace` is also included in Gecode's MiniModel module (see Section 7.3 and Support for cost-based optimization).

The `IntMaximizeSpace` class is a sub-class of `Space` that defines a `constrain()` member function based on the cost of a space. Our model must implement a virtual `cost()` function that returns an integere variable defining the cost (the function must be **`const`**). In our example, we extend the model to maintain the cost (the amount of money) in a dedicated variable `money` (note that this variable must also be updated during cloning).

SEND MORE MONEY DE-MYSTIFIED ≡ [DOWNLOAD]

```
#include <gecode/int.hh>
#include <gecode/minimodel.hh>
#include <gecode/search.hh>
...
class SendMoreMoney : public Space {
protected:
  IntVarArray l;
public:
  SendMoreMoney(void) : l(*this, 8, 0, 9) {
    ...
    rel(*this,             1000*s + 100*e + 10*n + d
                         + 1000*m + 100*o + 10*r + e
              == 10000*m + 1000*o + 100*n + 10*e + y);
    ...
  }
  ...
};
...
```

Figure 3.1: A Gecode model for Send More Money using modeling support

SEND MOST MONEY WITH COST ≡ [DOWNLOAD]

```
...
class SendMostMoney : public IntMaximizeSpace {
protected:
  IntVarArray l;
  IntVar money;
public:
  SendMostMoney(void)
    : l(*this, 8, 0, 9), money(*this,0,100000) {
    ...
    rel(*this, money == 10000*m + 1000*o + 100*n + 10*e + y);
    ...
  }
  SendMostMoney(bool share, SendMostMoney& s)
    : IntMaximizeSpace(share, s) {
    l.update(*this, share, s.l);
    money.update(*this, share, s.money);
  }
  ...
```

▶ COST FUNCTION

```
};
...
```

Figure 3.2: A Gecode model for Send Most Money using a cost function

The cost function then just returns the amount of money as follows:

COST FUNCTION ≡
```
virtual IntVar cost(void) const {
  return money;
}
```

3.3 Using the script commandline driver

In order to experiment from the commandline with different model variants, different search engines, and so on, it is convenient to have support for passing different option values on the commandline that then can be used by a model. Gecode comes with a simple commandline driver that defines `Script` as a subclass of `Space` for modeling and support for commandline options.[1]

Defining a script class. Suppose that we want to experiment with two different variants of Send Most Money with a cost function: the first variant uses the model from Section 3.2 and the second variant models the equation $SEND+MOST=MONEY$ by using carry variables.

Using carry variables with several linear equations instead of a single linear equation is straightforward. A carry variable is an integer variable with value `0` or `1` and each column in the equation $SEND+MOST=MONEY$ is modeled by a linear equation involving the appropriate carry variable as follows:

USING CARRIES ≡
```
{
  IntVar c0(*this, 0, 1), c1(*this, 0, 1),
         c2(*this, 0, 1), c3(*this, 0, 1);
  rel(*this,      d + t == y + 10 * c0);
  rel(*this, c0 + n + s == e + 10 * c1);
  rel(*this, c1 + e + o == n + 10 * c2);
  rel(*this, c2 + s + m == o + 10 * c3);
  rel(*this, c3         == m);
}
```

Figure 3.3 shows a model for Send Most Money that uses the `IntMaximizeScript` class as base class (see Script classes) rather than `IntMaximizeSpace` (likewise, the driver module also offers a `Script` class to be used instead of `Space`). There are two main differences between `IntMaximizeScript` and `IntMaximizeSpace` (`Script` and `Space`). First, the constructor must accept a constant argument of type `Options` (actually, it must accept a constant argument of the type that is specified for the `run` member function to be explained below)

[1] One might wonder why a commandline driver should be part of Gecode but this is due to popular demand. The driver functionality has been there for the examples that ship with Gecode and everybody has been using it and got into trouble as it was not available as a module that could be easily reused. Here we are!

SEND MOST MONEY WITH DRIVER ≡ [DOWNLOAD]

```
#include <gecode/driver.hh>
#include <gecode/int.hh>
#include <gecode/minimodel.hh>

using namespace Gecode;

class SendMostMoney : public IntMaximizeScript {
  ...
public:
  enum {
    MODEL_SINGLE, MODEL_CARRY
  };
  SendMostMoney(const Options& opt)
    : l(*this, 8, 0, 9), money(*this,0,100000) {
    ...
    switch (opt.model()) {
    case MODEL_SINGLE:
      ...
      break;
    case MODEL_CARRY:
      ► USING CARRIES
      break;
    }
    ...
  }
  ...
  virtual void print(std::ostream& os) const {
    os << l << std::endl;
  }
};

int main(int argc, char* argv[]) {
  ► COMMANDLINE OPTIONS
  ► RUN SCRIPT
  return 0;
}
```

Figure 3.3: A Gecode model for Send Most Money using the script commandline driver

that is used to pass values computed from options passed on the commandline. Second, a subclass of `IntMaximizeScript` must define a virtual print function that accepts a standard output stream as argument.

Note that one has to include `<gecode/driver.hh>` for a model that uses the script commandline driver. However, neither `<gecode/search.hh>` nor `<gecode/gist.hh>` need to be included as search is handled by the commandline driver.

Tip 3.1 (Linking against the driver)**.** As discussed in Section 2.3, when you use the commandline driver on a platform (Linux and relatives) that requires to state all libraries to link against, you also have to link against the library for the commandline driver (that is, for Linux and relatives by adding `-lgecodedriver` to the compiler options on the commandline). ◀

The class `SendMostMoney` defines an enumeration type with values `MODEL_SINGLE` (for a model where a single linear equation is posted for $SEND + MOST = MONEY$) and `MODEL_CARRY` (for a model where several linear equations using carry variables are posted for $SEND + MOST = MONEY$). The options object `opt` provides a member function `model` that returns an enumeration value and posts the constraints accordingly.

Defining commandline options. The mapping between strings passed on the commandline and the values `MODEL_SINGLE` and `MODEL_CARRY` is established by configuring an object of class `Options` accordingly as follows:

COMMANDLINE OPTIONS ≡
```
Options opt("SEND + MOST = MONEY");
opt.model(SendMostMoney::MODEL_SINGLE,
          "single", "use single linear equation");
opt.model(SendMostMoney::MODEL_CARRY,
          "carry", "use carry");
opt.model(SendMostMoney::MODEL_SINGLE);
opt.solutions(0);
opt.parse(argc,argv);
```

This code creates a new `Option` object `opt` where the string `"SEND + MOST = MONEY"` serves as identification (such as when requesting to print help about the available commandline options).

The first call to `opt.model()` defines that `single` is a legal value for the option switch `-model`, that the string `use single linear equation` is the help text for the option value, and that the corresponding value (that is, the value returned by `opt.model()`) is `SendMostMoney::MODEL_SINGLE`. The last call to `opt.model` defines the default value.

As we are performing best solution search, we want to compute all possible solutions. This is done by setting `opt.solutions` to `0` (the default value is `1` for searching for the first solution). Note that this default value can be changed on the commandline by passing an integer as value for the `-solutions` commandline option. Parsing the commandline by `parse`

now takes the configured values for the commandline options `-model` and `-solutions` into account.

The `Options` class supports most options that are useful for propagation, search, and so on similar to the `model` option. The full details are explained in Chapter 11.

The last piece of our model is calling the static `run` method of the script class by passing as template arguments our script class type `SendMostMoney`, `BAB` as the search engine we would like to use, and the type of options `Options` (as mentioned before, the constructor of the script must accept a single argument of this type).

```
RUN SCRIPT ≡
  Script::run<SendMostMoney,BAB,Options>(opt);
```

Running the script from the commandline. Suppose that we have compiled our script example as the file `send-most-money-with-driver.exe` (for some platforms, just drop `.exe`). Then

```
  send-most-money-with-driver.exe
```

prints something along the lines

```
  SEND + MOST = MONEY
  {9, 3, 4, 2, 1, 0, 5, 7}
  {9, 3, 4, 2, 1, 0, 6, 8}
  {9, 4, 5, 2, 1, 0, 6, 8}
  {9, 5, 6, 3, 1, 0, 4, 7}
  {9, 6, 7, 2, 1, 0, 3, 5}
  {9, 6, 7, 3, 1, 0, 5, 8}
  {9, 7, 8, 2, 1, 0, 3, 5}
  {9, 7, 8, 2, 1, 0, 4, 6}

  Initial
          propagators: 3
          branchers:   1

  Summary
          runtime:      0.024 (24.000000 ms)
          solutions:    8
          propagations: 109
          nodes:        33
          failures:     9
          restarts:     0
          no-goods:     0
          peak depth:   8
```

If we want to try the model with carry variables we can do that by running

```
send-most-money-with-driver.exe -model carry
```

We can also use Gist by giving a different mode of execution:

```
send-most-money-with-driver.exe -mode gist
```

As we call `run` with `BAB` as type of search, Gist will automatically start in branch-and-bound mode. Other supported modes are `solution` (the default mode shown above), `time` for printing average runtimes, and `stat` for just printing an execution statistics but no solutions.

Another important commandline option is `-help` which prints the options supported by the script together with some configuration information. The full details are explained in Chapter 11.

Tip 3.2 (Aborting execution)**.** In the `solution` and `stat` modes, the driver aborts the search gracefully if you send it a `SIGINT` signal, for example by pressing Ctrl-C on the command line. So if your model runs a long time without returning solutions, you can press Ctrl-C and still see the statistics that tell you how deep the search tree was up to that point, or how many nodes the search has explored.

When you press Ctrl-C twice, the process is interrupted immediately. This can be useful when debugging programs, e.g. if the process is stuck in an infinite loop. Alternatively, you can make the driver ignore Ctrl-C altogether, so that it immediately interrupts your program, using the commandline option `-interrupt false`. ◀

Tip 3.3 (How Gecode has been configured)**.** Depending on the hardware and software platform on which Gecode has been compiled, some features (such as Gist, thread support for parallel search, trigonometric and transcendental constraints for floats) might not be available. For example, if Gist is not supported, the request for `-mode gist` will be silently ignored and the normal mode (`-mode solution`) is used instead.

To find out which features are available, just invoke a program using the commandline driver with the `-help` option. At the beginning of the printed text, you will find a short configuration summary.

Note that all features are enabled for the precompiled binaries that are available from the Gecode webpages. ◀

Tip 3.4 (Which version of Gecode are we using?)**.** Some programs might have to deal with incompatible changes between different versions of Gecode. The Gecode-defined macro `GECODE_VERSION_NUMBER` can be used to find out which version of Gecode is used during compilation. The macro's value is defined as $100000 \times x + 100 \times y + z$ for Gecode version $x.y.z$. ◀

4 Integer and Boolean variables and constraints

This chapter gives an overview of integer and Boolean variables and the constraints available for them in Gecode. The chapter focuses on variables and constraints, a discussion of branching for integer and Boolean variables can be found in Section 8.2.

The chapter does not make an attempt to duplicate the reference documentation (see Using integer variables and constraints). It is concerned with the most important ideas and principles underlying integer and Boolean variables and constraints. In particular, the chapter provides entry points into the reference documentation and points to illustrating examples.

Overview. Section 4.1 details how integer and Boolean variables (and variables in general) can be used for modeling. Variable arrays and argument arrays are discussed in Section 4.2. Important aspects of how constraints are posted in Gecode are explained in Section 4.3. These sections belong to the basic reading material of Part M.

The remaining sections Section 4.4 and Section 4.5 provide an overview of the constraints that are available for integer and Boolean variables in Gecode.

Important. Do not forget to add

```
#include <gecode/int.hh>
```

to your program when you want to use integer or Boolean variables and constraints.

Convention. All program fragments and references to classes, namespaces, and other entities assume that declarations and definitions from the `Gecode` namespace are visible: for example, by adding

```
using namespace Gecode;
```

to your program.

The variable `home` refers to a space reference (of type `Space&`) and defines the home space in which new variables, propagators, and branchers are posted. Often (as in Chapter 2 and Chapter 3) `home` will be `*this`, referring to the current space.

4.1 Integer and Boolean variables

Variables in Gecode are for modeling. They provide operations for creation, access, and update during cloning. By design, the only way to modify (constrain) a variable is by post functions for constraints and branchers.

Integer variables are instances of the class `IntVar` while Boolean variables are instances of the class `BoolVar`. Integer variables are *not* related to Boolean variables. A Boolean variable is *not* an integer variable with a domain that is included in $\{0,1\}$. The only way to get an integer variable that is equal to a Boolean variable is by posting a channeling constraint between them (see Section 4.4.11).

Tip 4.1 (Do not use views for modeling)**.** If you — after some browsing of the reference documentation — should come across integer views such as `IntView`, you might notice that views have a richer interface than integer variables. You might feel that this interface looks too powerful to be ignored. Now, you really should put some trust in this document: views are *not* for modeling.

The more powerful interface only works within propagators and branchers, see Part P and Part B. ◀

4.1.1 Creating integer variables

A variable provides a read-only interface to a *variable implementation* where the same variable implementation can be referred to by arbitrarily many variables.

New integer variables are created by using a constructor. A new integer variable `x` is created by

```
IntVar x(home, -4, 20);
```

This declares a variable `x` of type `IntVar` in the space `home`, creates a new integer *variable implementation* with domain $\{-4,\ldots,20\}$, and points `x` to the newly created integer variable implementation.

The domain of a variable can also be specified by an integer set `IntSet`, for example by

```
IntVar x(home, IntSet(-4, 20));
```

which creates a new variable with domain $\{-4,\ldots,20\}$. An attempt to create an integer variable with an empty domain throws an exception of type `Int::VariableEmptyDomain`.

Tip 4.2 (Initializing variables)**.** Variables do not have an initialization function. However, new variables can be also created by

```
IntVar x;
x = IntVar(home, -4, 20);
```

◀

The default or copy constructor of a variable does not create a new variable (that is, a new variable implementation). Instead, the variable does not refer to any variable implementation (default constructor) or to the same variable implementation (copy constructor). For example, in

```
IntVar x(home, 1, 4);
IntVar y(x);
```

both `x` and `y` refer to the same integer variable implementation. Using a default constructor and an assignment operator is equivalent:

```
IntVar x(home, 1, 4);
IntVar y;
y=x;
```

4.1.2 Limits for integer values

The set of values for an integer variable is a subset of the values of the type **`int`**. The set of values is symmetric: $-$`Int::Limits::min` $=$ `Int::Limits::max` for the smallest possible integer variable value `Int::Limits::min` and the largest possible integer variable value `Int::Limits::max`. Moreover, `Int::Limits::max` is strictly smaller than the largest possible integer value `INT_MAX` and `Int::Limits::min` is strictly larger than the smallest possible integer value `INT_MIN`. These limits are defined in the namespace `Int::Limits`.

Any attempt to create a variable with values outside the defined limits throws an exception of type `Int::OutOfLimits`. The same holds true for any attempt to use an integer value outside the defined limits when posting a constraint or brancher.

4.1.3 Variable domains are never empty

An important invariant in Gecode is that the domain of a variable is never empty. When a variable domain should become empty during propagation, the space is failed but the variable's domain is kept. In fact, this is the very reason why an attempt to create a variable with an empty domain, for example by

```
IntVar x(home, 1, 0);
```

throws an exception of type `Int::VariableEmptyDomain`.

Tip 4.3 (Small variable domains are beautiful)**.** It is not an omission that an integer variable has no constructor that creates a variable with the largest possible domain. One could argue that a constructor like that would come in handy for creating temporary variables. After all, one would not have to worry about the exact domain!

Sorry, but one has to worry. The apparent omission is deliberate to make you worry indeed. For many propagators posted for a constraint a small domain is essential. For example, when posting a `linear` constraint (as in Section 2.1), variable domains that are

too large might result in an exception of type `Int::OutOfLimits` as during propagation numerical overflow might occur (even if Gecode resorts to a number type supporting larger numbers than **`int`** for propagating `linear`). Moreover, the runtime of other propagators (for example, many domain propagators such as domain consistent `distinct`) depend critically on the size of a domain. Again, Gecode tries to be clever in most of the cases. But, it is better to make it a habit to think about initial variable domains carefully (please remember: better safe than sorry).

For examples where small variable domains matter, see Tip 12.1 and Tip 15.2. ◀

4.1.4 Creating Boolean variables

The only difference between integer and Boolean variables is that Boolean variables can only take the values `0` or `1`. Any attempt to create a Boolean variable with values different from `0` or `1` throws an exception of type `Int::NotZeroOne`.

Convention. If Boolean variables are not explicitly mentioned in the following, the same functionality for integer variables is also available for Boolean variables and has the same behavior.

4.1.5 Variable access functions

Variables provide member functions for access, such as `x.min()` for the minimum value of the current domain for an integer or Boolean variable `x`. In particular, the member function `x.val()` accesses the integer value of an already assigned variable (if the variable is not yet assigned, an exception of type `Int::ValOfUnassignedVar` is thrown). In addition, variables can be printed by the standard output operator `<<`.

4.1.6 Iterating over integer variable domains

The entire domain of an integer variable can be accessed by a *value iterator* `IntVarValues` or a *range iterator* `IntVarRanges`. For example, the loop

```
for (IntVarValues i(x); i(); ++i)
  std::cout << i.val() << ' ';
```

uses the value iterator `i` to print all values of the domain of the integer variable `x`. The call operator `i()` tests whether there are more values to iterate for `i`, the prefix increment operator `++i` moves the iterator `i` to the next value, and `i.val()` returns the current value of the iterator `i`. The values are iterated in strictly increasing order.

Similarly, the following loop

```
for (IntVarRanges i(x); i(); ++i)
  std::cout << i.min() << ".." << i.max() << ' ';
```

uses the range iterator `i` to print all *ranges* of the integer variable `x`. Given a finite set of integers d, the *range sequence* of d is the shortest (and unique) sequence of ranges (intervals)

$$\langle [n_0 \,..\, m_0], \ldots, [n_k \,..\, m_k] \rangle$$

such that the sequence is ordered and non-adjacent ($m_i + 1 < n_{i+1}$ for $0 \leq i < k$). A range iterator iterates over the ranges in the range sequence of a variable's domain. Like a value iterator, a range iterator implements the call operator `i()` to test whether there are more ranges to iterate for `i` and the prefix increment operator `++i` to move `i` to the next range. As a range iterator `i` iterates over ranges, it implements the member functions `i.min()` and `i.max()` for the minimal, respectively maximal, value of the current range.

Iteration of values and ranges for Boolean variables is not available (as it is not needed).

4.1.7 When to inspect a variable

Note that one must not change the domain of a variable (for example, by posting a constraint on that variable) while an iterator for that variable is still in use. This is the same as for most iterators, for example, for iterators in the C++ Standard Template Library (STL).

Otherwise, a variable can always be inspected: at any place (that is, not only in member functions of the variable's home) and at any time (regardless of the status of a space). If the variable's home is failed, the variable can still be inspected. However, it might be the case that the variable domain has more values than expected. For example, after creating a variable `x` with the singleton domain {0} and posting the constraint that `x` must be different from `0` by (read Tip 2.2 about `status()`):

```
IntVar x(home, 0, 0);
rel(home, x, IRT_NQ, 0);
(void) home.status();
```

the space `home` is failed but the variable `x` still contains the value `0` in its domain.

4.1.8 Updating variables

As discussed in Section 2.1, a variable must be updated during cloning in the copy constructor used by a space's `copy()` member function. For example, a variable `x` is updated by

```
x.update(home, share, y);
```

where `share` is the Boolean argument passed as argument to `copy()` and `y` is the variable from which `x` is to be updated. While `x` belongs to `home`, `y` belongs to the space being cloned.

A space only needs to update the variables that are part of the solution, so that their values can be accessed after a solution space has been found. Temporary variables do not need to be copied.

Assume that we want to constrain the integer variable `p` to be the product of `x`, `y`, and `z`. Gecode only offers multiplication of two variables, hence a temporary variable `t` is created (assume also that we know that the values for `t` are between `0` and `1000`):

```
IntVar t(home, 0, 1000);
mult(home, x, y, t);
mult(home, t, z, p);
```

Here, `t` does not require updating. The multiplication propagators created by `mult` take care of updating the variable implementation of `t`.

4.2 Variable and argument arrays

Gecode has very few *proper* data structures. The proper data structures for integer and Boolean variables are the variables themselves, integer sets, and arrays of variables. Proper means that these data structures can be updated and hence be stored in a space. Other proper data structures are DFAs (deterministic finite automata) and tuple sets used for extensional constraints. These data structures are discussed in Section 4.4.13.

Of course, data structures that themselves do not contain proper data structures can be stored in a space, such as integers, pointers, and strings.

Gecode supports the programming of new proper data structures, this is discussed in Section 30.3.

4.2.1 Integer and Boolean variable arrays

Integer variable arrays of type `IntVarArray` can be used like variables. For example,

```
IntVarArray x(home, 4, -10, 10);
```

creates a new integer variable array with four variables containing newly created variables with domain $\{-10,\ldots,10\}$. Boolean variable arrays of type `BoolVarArray` are analogous.

Creation of a variable array allocates memory from the home space. The memory is freed when the space is deleted (not when the destructor of the variable array is called). Variable arrays can be created without creating new variables by just passing the size. That is,

```
IntVarArray x(home, 4);
for (int i=0; i<4; i++)
  x[i] = IntVar(home, -10, 10);
```

is equivalent to the previous example.

The other operations on variable arrays are as one would expect. For example, one can check whether all variables are assigned using the `assigned()` function. More importantly, variable arrays like variables have an update function and variable arrays must be updated during cloning. In the following, we will refer to the size of a variable array `x` by |x| (which can be computed by `x.size()`).

Matrix interface. Many models are naturally expressed by using matrices. Gecode offers support that superimposes a matrix interface for modeling on an array, see Section 7.2.

4.2.2 Argument arrays

As mentioned above, the memory allocated for a variable array is freed only when its home space is deleted. That makes variable arrays *unsuited* for temporary variable arrays, in particular for arrays that are built dynamically or used as arguments for post functions.

For this reason, Gecode provides argument arrays: `IntVarArgs` for integer variables, `BoolVarArgs` for Boolean variables, `IntArgs` for integers, and `IntSetArgs` for integer sets (see Argument arrays). Internally, they allocate space from the heap[1] and the memory is freed when their destructor is executed. They are *not* proper data structures as they cannot be updated.

Argument arrays can be created empty:

```
IntVarArgs x;
```

with a certain size but without initializing the elements:

```
IntVarArgs x(5);
```

or fully initialized:

```
IntVarArgs x(home,5,0,10);
```

For a typical example, consider Section 2.1 where an integer argument array and an integer variable argument array are used to pass coefficients and variables to the `linear` post function.

Dynamic argument arrays. In contrast to variable arrays, argument arrays can grow dynamically by adding elements or whole arrays using **`operator`**`<<`:

```
IntVarArgs x;
x << IntVar(home,0,10);
IntVarArgs y;
y << IntVar(home,10,20);
y << x;
linear(home, IntVarArgs()<<x[0]<<x[1], IRT_EQ, 0);
```

Furthermore, argument arrays can be concatenated using **`operator`**`+`:

```
IntVarArgs z = x+y;
```

[1] Actually, if an argument array has few fields it uses some space that is part of the object implementing the array rather than allocating memory from the heap. Hence, small argument arrays reside entirely on the stack.

Slices. It is sometimes necessary to post constraints on a subsequence of the variables in an array. This is made possible by the `slice(start,inc,n)` method of variable and argument arrays. The `start` parameter gives the starting index of the subsequence. The `inc` optional parameter gives the increment, i.e., how to get from one element to the next (its default is 1). The `n` parameter gives the maximal length of the resulting array (its default is -1, meaning as long as possible).

The following examples should make this clearer. Assume that the integer variable argument array `x` is initialized as follows:

```
IntVarArgs x(home, 10, 0, 10);
```

Then the following calls of `slice()` return:

- `x.slice(5)` returns an array with elements `x[5],x[6],···,x[9]`.
- `x.slice(5,1,3)` returns `x[5],x[6],x[7]`.
- `x.slice(5,-1)` returns `x[5],x[4],···,x[0]`.
- `x.slice(3,3)` returns `x[3],x[6],x[9]`.
- `x.slice(8,-2)` returns `x[8],x[6],x[4],x[2],x[0]`.
- `x.slice(8,-2,3)` returns `x[8],x[6],x[4]`.

Tip 4.4 (Reversing argument arrays)**.** The `slice()` method can be used to compute an array with the elements of `x` in reverse order like this:

```
x.slice(x.size()-1,-1)
```

◀

Creating integer argument arrays. Integer argument arrays with simple sequences of integers can be generated using the static method `IntArgs::create(n,start,inc)`. The `n` parameter gives the length of the generated array. The `start` parameter is the starting value, and `inc` determines the increment from one value to the next.

Here are a few examples:

- `IntArgs::create(5,0)` creates an array with elements `0,1,2,3,4`.
- `IntArgs::create(5,4,-1)` creates `4,3,2,1,0`.
- `IntArgs::create(3,2,0)` creates `2,2,2`.
- `IntArgs::create(6,2,2)` creates `2,4,6,8,10,12`.

Tip 4.5 (Dynamically constructing models)**.** Sometimes the number of variables cannot be determined easily, for example when it depends on data read from a file.

Suppose the following script with a variable array x:

```
DYNAMIC SCRIPT ≡
  class Script : public Space {
    IntVarArray x;
  public:
    Script(void) {
      ▶ READ DATA
      ...
      ▶ INITIALIZE VARIABLE ARRAY
    }
    ...
  }
```

It is easy to use a variable argument array _x for collecting variables as follows:

```
READ DATA ≡
  IntVarArgs _x;
  while (···) {
    ...
    _x << IntVar(*this,···);
  }
```

and then initialize the variable array x using the argument array:

```
INITIALIZE VARIABLE ARRAY ≡
  x = IntVarArray(*this,_x);
```

◀

In the following we do not distinguish between arrays and argument arrays unless operations require a certain type of array. In fact, all post functions for constraints and branchers only accept variable argument arrays. A variable array is automatically casted to a variable argument array if needed.

4.2.3 STL-style iterators

All arrays in Gecode (including variable arrays and argument arrays) also support STL-style (Standard Template Library) iterators. For example, assume that a is an integer variable argument array. Then

```
  for (IntVarArgs::iterator i = a.begin(); i != a.end(); ++i) {
    ...
  }
```

creates an iterator `i` for the elements of `a` and iterates from the first to the last element in `a`.

More powerfully, iterators give you the ability to work with STL algorithms. Suppose that `f()` is a function that takes an integer variable by reference such as in

```
void f(IntVar& x) { ··· }
```

and `a` is an integer variable argument array. Then

```
#include <algorithm>

std::for_each(a.begin(), a.end(), f);
```

applies the function `f()` to each integer variable in `a`.

4.3 Posting constraints

This section provides information about general principles for posting constraints over integer and Boolean variables.

4.3.1 Post functions are clever

A constraint post function carefully analyzes its arguments. Based on this analysis, the constraint post function chooses the best possible propagator for the constraint.

For example, when posting a `distinct` constraint (see Section 4.4.7) for the variable array `x` by

```
distinct(home, x);
```

where `x` has two elements, the much more efficient propagator for disequality $x_0 \neq x_1$ is created.

4.3.2 Everything is copied

When passing arguments to a post function, all data structures that are needed for creating a propagator (or several propagators) implementing a constraint are copied. That is, none of the data structures that are passed as arguments are needed after a constraint has been posted.

4.3.3 Reified constraints

Many constraints also exist as *reified* variants: the validity of a constraint is reflected to a Boolean control variable (reified constraints are also known as meta-constraints). In addition to full reification also half reification [14] is supported for reified constraints. Whether a reified version exists for a given constraint can be found in the reference documentation.

If a reified version does exist, the Boolean control variable (and possibly information about the reification mode, to be discussed in Section 4.3.4) is passed as the last non-optional argument.

For example, posting

```
rel(home, x, IRT_EQ, y, b);
```

for integer variables `x` and `y` and a Boolean control variable `b` creates a propagator for the reified constraint $b = 1 \Leftrightarrow x = y$ that propagates according to the following rules:

- If `b` is assigned to `1`, the constraint $x = y$ is propagated.
- If `b` is assigned to `0`, the constraint $x \neq y$ is propagated.
- If the constraint $x = y$ holds, then $b = 1$ is propagated.
- If the constraint $x \neq y$ holds, then $b = 0$ is propagated.

4.3.4 Half reification

Reification as discussed in the previous paragraph is also known as *full* reification as it propagates a full equivalence between the constraint c and the constraint that a Boolean control variable is equal to `1`. *Half reification* propagates only one direction of the equivalence [14]. Half reification can be used by passing an object of class `Reify` that combines a Boolean control variable and a *reification mode* of type `ReifyMode` (see Using integer variables and constraints).

For example, the half reified constraint $b = 1 \Rightarrow x = y$ for integer variables `x` and `y` and a Boolean control variable `b` can be posted by

```
Reify r(b, RM_IMP);
rel(home, x, IRT_EQ, y, r);
```

and is propagated as follows (`RM_IMP` suggests *implication* $\Rightarrow$):

- If `b` is assigned to `1`, the constraint $x = y$ is propagated.
- If the constraint $x \neq y$ holds, then $b = 0$ is propagated.

Likewise, the half reified constraint $b = 1 \Leftarrow x = y$ for integer variables `x` and `y` and a Boolean control variable `b` can be posted by

```
Reify r(b, RM_PMI);
rel(home, x, IRT_EQ, y, r);
```

and is propagated as follows (`RM_PMI` suggests *inverse implication* $\Leftarrow$):

- If `b` is assigned to `0`, the constraint $x \neq y$ is propagated.

- If the constraint `x = y` holds, then `b = 1` is propagated.

Full reification can be requested by the reification mode `RM_EQV` (for equivalence $\Leftrightarrow$) as follows:

```
Reify r(b, RM_EQV);
rel(home, x, IRT_EQ, y, r);
```

As the constructor for `Reify` has `RM_EQV` as default value for its second argument, this can be written shorter as:

```
Reify r(b);
rel(home, x, IRT_EQ, y, r);
```

or even shorter as:

```
rel(home, x, IRT_EQ, y, b);
```

For convenience, three functions `eqv()`, `imp()`, and `pmi()` exist that take a Boolean variable and return a corresponding object of class `Reify`. For example, instead of writing:

```
Reify r(b, RM_IMP);
rel(home, x, IRT_EQ, y, r);
```

one can write more concisely:

```
rel(home, x, IRT_EQ, y, imp(b));
```

4.3.5 Selecting the consistency level

For many constraints, Gecode provides different propagators with different levels of consistency. All constraint post functions take an optional argument of type `IntConLevel` (see Using integer variables and constraints) controlling which propagator is chosen for a particular constraint.

The different values for `IntConLevel` have the following meaning:

- `ICL_VAL`: perform value propagation. A typical example is naive `distinct`: wait until a variable becomes assigned to a value n, then prune n from all other variables.
- `ICL_BND`: perform bounds propagation or achieve bounds consistency. This captures both bounds consistency over the integers (for example, for `distinct`, see Section 4.4.7) or bounds consistency over the real numbers (for example, for `linear`, see Section 4.4.6). For more information on bounds consistency over integers or real numbers, see [12].

  Some propagators that are selected might not even achieve bounds consistency but the idea is that the propagator performs propagation by reasoning on the bounds of variable domains.

- `ICL_DOM`: perform domain propagation or achieve domain consistency. Most propagators selected by `ICL_DOM` achieve domain consistency but some just perform propagation by taking entire variable domains for propagation into account (for example, `circuit`, see Section 4.4.17).

- `ICL_DEF`: choose default consistency level for this constraint.

Whether bounds or domain consistency is achieved and the default consistency level for a constraint are mentioned in the reference documentation for each post function.

Tip 4.6 (Different consistency levels have different costs)**.** Note that propagators of different consistency level for the very same constraint can have vastly different cost. In general, propagation for `ICL_VAL` will be cheapest, while propagation for `ICL_DOM` will be most expensive.

The reference documentation for a constraint lists whether propagation for a particular consistency level might have prohibitive cost for a large number of variables or a large number of values in the variables' domains. For example, for the `linear` constraint with n variables and at most d values for each variable, the complexity to perform bounds propagation (that is, `ICL_BND`) is $O(n)$ whereas the complexity for domain propagation (that is, `ICL_DOM`) is $O(d^n)$. ◀

4.3.6 Exceptions

Many post functions check their arguments for consistency before attempting to create a propagator. For each post function, the reference documentation lists which exceptions might be thrown.

4.3.7 Unsharing arguments

Some constraints can only deal with *non-shared* variable arrays: a variable is not allowed to appear more than once in the array (more precisely: no unassigned variable implementation appears more than once in the array). An attempt to post one of these constraints with shared variable arrays will throw an exception of type `Int::ArgumentSame`.

To be able to post one of these constraints on shared variable arrays, Gecode provides a function `unshare` (see Unsharing variables) that takes a variable argument array `x` as argument as in:

```
unshare(home, x);
```

It replaces each but the first occurrence of a variable y in `x` by a new variable z, and creates a propagator $y = z$ for each new variable z.

Note that `unshare` requires a variable argument array and *not* a variable array. If `x` is a variable array, the following

```
IntVarArgs y(x);
```

creates a variable argument array y containing the same variables as x.

Tip 4.7 (Unsharing is expensive)**.** It is important to keep in mind that `unshare` creates new variables and propagators. This is also the reason why unsharing is not done implicitly by a post function for a constraint that does not accept shared variable arrays.

Consider the following example using `extensional` constraints for a variable argument array x possibly containing a variable more than once, where a and b are two different DFAs (see Section 4.4.13 for `extensional` constraints). By

```
unshare(home, x);
extensional(home, x, a);
extensional(home, x, b);
```

multiple occurrences of the same variable in x are unshared *once* and the propagators for extensional can work on the same non-shared array.

If unsharing were implicit, the following

```
extensional(home, x, a);
extensional(home, x, b);
```

would unshare x twice and create many more (useless) propagators and variables. Rather than implicitly unsharing the same array over and over again (and hence creating variables and propagators), unsharing is made explicit and should be done only once, if possible. ◀

4.4 Constraint overview

This section provides an overview of the constraints and their post functions available for integer and Boolean variables.

4.4.1 Domain constraints

Domain constraints constrain integer variables and variable arrays to values from a given domain. For example, by

```
dom(home, x, 2, 12);
```

the values of the variable x (or of all variables in a variable array x) are constrained to be between 2 and 12. Domain constraints also take integers (assigning variables to the integer value) and integer sets of type `IntSet`. For example,

```
IntArgs a(4, 1,-3,5,-7)
IntSet d(a);
dom(home, x, d);
```

IRT_EQ	equality ($=$)	IRT_NQ	disequality ($\neq$)
IRT_LE	strictly less inequality ($<$)	IRT_LQ	less or equal inequality ($\leq$)
IRT_GR	strictly greater inequality ($>$)	IRT_GQ	greater or equal inequality ($\geq$)

Figure 4.1: Integer relation types

constrains the variable x (or, the variables in x) to take values from the set $\{-7, -3, 1, 5\}$ (see also GCCAT: domain, in, in_interval, in_intervals, in_set).

Note that there are no domain constraints for Boolean variables, please use relation constraints instead, see Section 4.4.4.

The domain of an integer or Boolean variable x can be constrained according to the domain of another variable d by

```
dom(home, x, d);
```

Here, x and d can also be arrays of integer or Boolean variables.

Domain constraints for a single variable also support reification. For examples using domain constraints, see n-Knight's tour (simple model) and Packing squares into a rectangle.

4.4.2 Membership constraints

Membership constraints constrain integer or Boolean variables to be included in an array of integer or Boolean variables. That is, for an integer variable array x and an integer variable y, the constraint

```
member(home, x, y);
```

forces that y is included in x:

$$y \in \{x_0, \ldots, x_{|x|-1}\}$$

As mentioned, x and y can also be Boolean variables. Membership constraints also support reification.

4.4.3 Simple relation constraints over integer variables

Simple relation constraints over integer variables enforce relations between integer variables and between integer variables and integer values. The relation depends on an integer relation type IntRelType (see Simple relation constraints over integer variables). Figure 4.1 lists the available integer relation types and their meaning.

Binary relation constraints. Assume that x and y are integer variables. Then

```
rel(home, x, IRT_LE, y);
```

constrains x to be strictly less than y. Similarly, by

```
rel(home, x, IRT_NQ, 4);
```

x is constrained to be different from 4. Both variants of `rel` also support reification (see also GCCAT: `eq`, `neq`, `lt`, `leq`, `gt`, `geq`).

Constraints between variable arrays and a single variable. If x is an integer variable array and y is an integer variable, then

```
rel(home, x, IRT_LQ, y);
```

constrains all variables in x to be less than or equal to y. Likewise,

```
rel(home, x, IRT_GR, 7);
```

constrains all variables in x to be larger than 7 (see also GCCAT: `arith`).

Constraints between array elements. If x is an integer variable array, then

```
rel(home, x, IRT_LQ);
```

constrains the variables in x to be sorted in increasing order as follows:

$$x_0 \leq x_1 \leq \cdots \leq x_{|x|-1}$$

The integer relation type values for inequalities (that is, `IRT_LE`, `IRT_GQ`, and `IRT_GR`) are analogous. For an example, see Chapter 12 and Finding optimal Golomb rulers.

By

```
rel(home, x, IRT_EQ);
```

all variables in the integer variable array x are constrained to be equal:

$$x_0 = x_1 = \cdots = x_{|x|-1}$$

By

```
rel(home, x, IRT_NQ);
```

the variables in x are constrained to be not all equal:

$$\neg \left(x_0 = x_1 = \cdots = x_{|x|-1} \right)$$

For an example, see Schur's lemma (see also GCCAT: `all_equal`, `decreasing`, `increasing`, `not_all_equal`, `strictly_decreasing`, `strictly_increasing`).

`BOT_AND`	conjunction ($\wedge$)	`BOT_OR`	disjunction ($\vee$)
`BOT_IMP`	implication ($\rightarrow$)	`BOT_EQV`	equivalence ($\leftrightarrow$)
`BOT_XOR`	exclusive or ($\nleftrightarrow$)		

Figure 4.2: Boolean operation types

Lexicographic constraints between variable arrays. If x and y are integer variable arrays (where the sizes of x and y can be different),

```
rel(home, x, IRT_LE, y);
```

constrains x and y such that x is lexicographically strictly smaller than y (analogously for the other inequality relations). For `IRT_EQ` and $|x| = |y|$, it is propagated that $x_i = y_i$ for $0 \leq i < |x|$. For `IRT_NQ` and $|x| = |y|$, it is propagated that $x_i \neq y_i$ for at least one i such that $0 \leq i < |x|$. If $|x| \neq |y|$, for `IRT_EQ` the space `home` is failed whereas for `IRT_NQ` the constraint is ignored (see also GCCAT: `lex_greater`, `lex_greatereq`, `lex_less`, `lex_lesseq`).

See Balanced incomplete block design (BIBD) for an example (albeit over Boolean variables).

4.4.4 Simple relation constraints over Boolean variables

Simple relation constraints over Boolean variables include the same post functions as simple relation constraints over integer variables. In addition, simple relation constraints over Boolean constraints provide support for the typical Boolean operations such as conjunction and disjunction. Boolean operations are defined by values of the type `BoolOpType` (see Simple relation constraints over integer variables). Figure 4.2 lists the available Boolean operation types.

For example, for Boolean variables x, y, and z,

```
rel(home, x, BOT_AND, y, z);
```

posts the constraint $x \wedge y = z$. Similarly,

```
rel(home, x, BOT_OR, y, 1);
```

posts that $x \vee y$ must be true (see also GCCAT: `and`, `equivalent`, `imply`, `or`, `xor`).

Note that the integer value must be either zero or one, otherwise an exception of type `Int::NotZeroOne` is thrown.

For an example, see Balanced incomplete block design (BIBD).

Tip 4.8 (Boolean negation)**.** Boolean negation can be easily obtained by using `IRT_NQ` as relation type. The constraint $x = \neg y$ for Boolean variables x and y can be posted by

```
rel(home, x, IRT_NQ, y);
```

◀

Additional constraints are available for Boolean variable arrays. For a Boolean variable array x and a Boolean variable y,

```
rel(home, BOT_OR, x, y)
```

posts the constraint

$$\bigvee_{i=0}^{|x|-1} x_i = y$$

Again, y can also be 0 or 1.

Note that Boolean implication is special in that it is not associative and Gecode follows normal notational convention. Hence for a Boolean variable array x and a Boolean variable y,

```
rel(home, BOT_IMP, x, y)
```

posts the constraint

$$x_0 \rightarrow \left(x_1 \rightarrow \left(\ldots \rightarrow \left(x_{|x|-2} \rightarrow x_{|x|-1}\right)\right)\right) = y$$

Again, y can also be 0 or 1.

Clause constraint. In order to avoid many propagators for negation, the clause constraint accepts both positive and negative Boolean variables. For Boolean variable arrays x and y and a Boolean variable z (again, z can also be 0 or 1)

```
clause(home, BOT_AND, x, y, z);
```

posts the constraint

$$\bigwedge_{i=0}^{|x|-1} x_i \wedge \bigwedge_{i=0}^{|y|-1} \neg y_i = z$$

(see also GCCAT: `clause_and`, `clause_or`, `nand`, `nor`).

Note that only `BOT_AND` and `BOT_OR` as Boolean operation types are supported, other Boolean operations types throw an exception of type `Int::IllegalOperation`.

For an example, see CNF SAT solver.

If-then-else constraint. An if-then-else constraint can be posted by

```
ite(home, b, x, y, z);
```

where b is a Boolean variable and x, y, and z are integer variables. In case b is one, then $x = z$ must hold, otherwise $y = z$ must hold.

post function	constraint posted	bnd	dom	GCCAT
`min(home, x, y, z);`	$\min(x,y)=z$	✓	✓	`minimum`
`max(home, x, y, z);`	$\max(x,y)=z$	✓	✓	`maximum`
`abs(home, x, y);`	$\lvert x\rvert = y$	✓	✓	`abs_value`
`mult(home, x, y, z);`	$x \cdot y = z$	✓	✓	
`sqr(home, x, y);`	$x^2 = y$	✓	✓	
`sqrt(home, x, y);`	$\lfloor\sqrt{x}\rfloor = y$	✓	✓	
`pow(home, x, n, y);`	$x^n = y$	✓	✓	
`nroot(home, x, n, y);`	$\lfloor\sqrt[n]{x}\rfloor = y$	✓	✓	
`div(home, x, y, z);`	$x \div y = z$	✓		
`mod(home, x, y, z);`	$x \bmod y = z$	✓		
`divmod(home, x, y, d, m);`	$x \div y = d \wedge x \bmod y = m$	✓		

Figure 4.3: Arithmetic constraints (`x`, `y`, `z`, `d`, and `m` are integer variables; `n` is an integer)

4.4.5 Arithmetic constraints

Arithmetic constraints exist only over integer variables. In addition to the constraints summarized in Figure 4.3 (bnd abbreviates bounds consistency and dom abbreviates domain consistency), the minimum and maximum constraints are also available for integer variable arrays. That is, for an integer variable array `x` and an integer variable `y`

```
min(home, x, y);
```

constrains `y` to be the minimum of the variables in `x` (`max` is analogous).

4.4.6 Linear constraints

Linear constraints over integer variables and Linear constraints over Boolean variables provide essentially the same post functions for integer and Boolean constraints (to be discussed below). The most general variant

```
linear(home, a, x, IRT_EQ, c);
```

posts the linear constraint

$$\sum_{i=0}^{|x|-1} a_i \cdot x_i = c$$

with integer coefficients `a` (of type `IntArgs`), integer variables `x`, and an integer constant `c`. Note that `a` and `x` must have the same size. Of course, all other integer relation types are supported, see Figure 4.1 for a table of integer relation types. Multiple occurrences of the same variable in `x` are explicitly allowed and common terms $a \cdot y$ and $b \cdot y$ for the same variable y are rewritten to $(a+b) \cdot y$ to increase propagation. For an example, see Section 2.1.

The array of coefficients can be omitted if all coefficients are one. That is,

```
linear(home, x, IRT_GR, c);
```

posts the linear constraint

$$\sum_{i=0}^{|x|-1} x_i > c$$

for a variable array x and an integer c.

Instead of an integer constant c as the right-hand side of the linear constraint, an integer variable can be used as well. This is true for linear constraints over both integer and Boolean variables: the right-hand side is always an integer value or an integer variable, even if the left-hand side involves Boolean variables. For example, when assuming that x is an array of Boolean variables,

```
linear(home, x, IRT_GQ, y);
```

imposes the constraint that there are at least y ones among the Boolean variables in x.

All variants of `linear` support reification and exist in variants that perform both bounds propagation (the default) and domain propagation (see also GCCAT: `scalar_product`, `sum_ctr`).

4.4.7 Distinct constraints

The `distinct` constraint (see Distinct constraints) enforces that integer variables take pairwise distinct values (also known as `alldifferent` constraint). Obviously, `distinct` does not exist for Boolean variables.

Posting

```
distinct(home, x);
```

constrains all variables in x to be pairwise different.

Posting

```
distinct(home, c, x);
```

for an array of integer values c (of type `IntArgs`) and an array of integer variables x of same size, constrains the variables in x such that

$$x_i + c_i \neq x_j + c_j \qquad \text{for } 0 \leq i, j < |x| \text{ and } i \neq j$$

Gecode offers value (the default), bounds (based on [27]), and domain propagation (based on [41]) for `distinct` (see also GCCAT: `alldifferent`, `alldifferent_cst`).

For examples, see in particular Chapter 12, n-Queens puzzle, and Crowded chessboard.

4.4.8 Counting constraints

Counting single values. Counting constraints count how often values are taken by an array of integer variables. The simplest case is

```
count(home, x, y, IRT_EQ, z);
```

which constrains z to be equal (controlled by IRT_EQ, all integer relation types are supported, see Figure 4.1) to the number of integer variables in x that are equal to y. Here y and z can be integer variables as well as integer values (see also GCCAT: atleast, atmost, count, exactly).

The count constraints also support counting how many integer variables are included in an integer set. If y is an integer set, then

```
count(home, x, y, IRT_EQ, z);
```

constrains z to be equal to the number of integer variables in x that are included in y (see also GCCAT: among, among_var, counts).

The following

```
count(home, x, c, IRT_EQ, z);
```

where x is an array of integer variables and c is an array of integers (of type IntArgs) with same size and z is an integer variable or value, constrains z to how often $x_i = c_i$, that is

$$z = \#\{i \in \{0,\ldots,|x|-1\} \mid x_i = c_i\}$$

Here, $\#s$ denotes the cardinality (number of elements) of a set s.

Counting multiple values. The count constraint also supports counting multiple values (also known as gcc, or *g*lobal *c*ardinality *c*onstraint). Suppose that x and y (the *counting variables*) are two integer variable arrays (not necessarily of the same size). Then

```
count(home, x, y);
```

posts the constraints that the number of variables in x that are equal to a value j is y_j (for $0 \leq j < |y|$):

$$\#\{i \in \{0,\ldots,|x|-1\} \mid x_i = j\} = y_j \qquad \text{for } 0 \leq j < |y|$$

and that no other values are taken by x:

$$\bigcup_{i=0}^{|x|-1} \{x_i\} = \{0,\ldots,|y|-1\}$$

Rather than using counting variables, one can also use an array of integer sets (IntSetArgs). Then the number of values taken must be included in each individual set.

A more general variant also takes into account that the values under consideration are non contiguous but are defined by an additional array of integer values. Suppose that x and y (the counting variables) are two integer variable arrays (not necessarily of the same size) and c is an array of integers with the same size as y.

Then,

```
count(home, x, y, c);
```

posts the constraints that the number of variables in x that are equal to the value c_j is y_j (for $0 \leq j < |y|$):

$$\#\{i \in \{0,\ldots,|x|-1\} \mid x_i = c_j\} = y_j \qquad \text{for } 0 \leq j < |y|$$

and that no other values but those in c are taken by x:

$$\bigcup_{i=0}^{|x|-1} \{x_i\} = \bigcup_{i=0}^{|c|-1} \{c_i\}$$

Again, y can also be an array of integer sets, where equality $=$ is replaced by set inclusion $\in$.

A slightly simpler variant replaces the cardinality variables by a single integer set. That is, for an array of integer variables x, an integer set d, and an array of integer values c

```
count(home, x, d, c);
```

posts the constraints that the number of variables in x that are equal to the value c_j is included in d (for $0 \leq j < |c|$):

$$\#\{i \in \{0,\ldots,|x|-1\} \mid x_i = c_j\} \in d \qquad \text{for } 0 \leq j < |c|$$

and that no other values but those in c are taken by x:

$$\bigcup_{i=0}^{|x|-1} \{x_i\} = \bigcup_{i=0}^{|c|-1} \{c_i\}$$

The last variant of `count` clarifies that `count` is a generalization of `distinct` (see Section 4.4.7): `distinct` constrains a value to occur at most once, whereas `count` offers more flexibility to constrain which values and how often these values can occur.

For example, if we know that the variables in the variable array x take values between `0` and `n-1`, then

```
count(home, x, IntSet(0,1), IntArgs::create(n,0,1));
```

is equivalent to

```
distinct(home, x);
```

Counting constraints only support integer variables, `linear` constraints can be used for Boolean variables, see Section 4.4.6. For examples, see the case studies in Chapter 13 and Chapter 15 or the examples Crowded chessboard and Magic sequence.

Note that Gecode implements the semantics of the original paper on the global cardinality constraint by Régin [38], where no other values except those specified may occur. This differs from the semantics in the Global Constraint Catalog [5], where values that are not mentioned can occur arbitrarily often.

Gecode offers value (the default), bounds (based on [37]), and domain propagation (based on [38]) for the global `count` constraint (see also GCCAT: `global_cardinality`).

4.4.9 Number of values constraints

Number of values constraints constrain how many values can be taken by an array of variables.

Assume that `x` is an array of integer variables and `y` is an integer variable. Then

```
nvalues(home, x, IRT_EQ, y);
```

constrains the number of distinct values in `x` to be equal to `y`, that is

$$\#\{x_0, \ldots, x_{|x|-1}\} = y$$

Instead of `IRT_EQ` any other integer relation type can be used, see Figure 4.1 for an overview. For example,

```
nvalues(home, x, IRT_LQ, y);
```

constrains the number of distinct values in `x` to be less than or equal to `y`. The array `x` can also be an array of Boolean variables and `y` can be an integer value.

The constraint is implemented by the propagators introduced in [6] (see also GCCAT: `nvalue`, `nvalues`). For an example using the `nvalues` constraint, see Dominating Queens.

4.4.10 Sequence constraints

Sequence constraints constrain how often values are taken by repeated subsequences of variables in an array of integer or Boolean variables. By

```
sequence(home, x, s, q, l, u);
```

where `x` is an array of integer or Boolean variables, `s` is an integer set, and `q`, `l`, and `u` are integers, all subsequences of length `q` in the variable array `x`, that is, the sequences

$$\begin{gathered}\langle x_0, \ldots, x_{q+0-1}\rangle \\ \langle x_1, \ldots, x_{q+1-1}\rangle \\ \cdots \\ \langle x_{|x|-q}, \ldots, x_{|x|-1}\rangle\end{gathered}$$

are constrained such that at least l and at most u variables in each subsequence are assigned to values from the integer set s.

In more mathematical notation, the constraint enforces

$$\bigwedge_{i=0}^{|x|-q} \text{among}(\langle x_i, \ldots, x_{i+q-1}\rangle, s, l, u)$$

where the among constraint for the subsequence starting at position i is defined as

$$l \leq \#\{j \in \{i, \ldots, i+q-1\} \mid x_j \in s\} \leq u$$

The constraint is implemented by the domain consistent propagator introduced in [61] (see also GCCAT: among_seq). For an example, see Car sequencing.

4.4.11 Channel constraints

Channel constraints channel Boolean to integer variables and integer variables to integer variables.

Channeling integer variables. For two integer variable arrays x and y of same size,

```
channel(home, x, y);
```

posts the constraint

$$x_i = j \iff y_j = i \qquad \text{for } 0 \leq i, j < |x|$$

(see also GCCAT: inverse). The channel constraint between two integer variable arrays also supports integer offsets. For integers n and m,

```
channel(home, x, n, y, m);
```

posts the constraint

$$x_i - n = j \iff y_j - m = i \qquad \text{for } 0 \leq i, j < |x|$$

(see also GCCAT: inverse_offset). For examples, see n-Knight's tour (simple model) and Black hole patience.

Channeling between integer and Boolean variables. As integer and Boolean variables are unrelated (see Section 4.1), the only way to express that a Boolean variable x is equal to an integer variable y is by posting either

```
channel(home, x, y);
```

or

```
channel(home, y, x);
```

The `channel` constraint can also map an integer variable `y` to an array of Boolean variables `x`. The constraint

$$x_i = 1 \iff y = i \qquad \text{for } 0 \leq i < |x|$$

is posted by

```
channel(home, x, y);
```

(see also GCCAT: `domain_constraint`). Note that an optional offset argument is supported. The constraint

$$x_i = 1 \iff y = i + n \qquad \text{for } 0 \leq i < |x|$$

for an integer value `n` is posted by

```
channel(home, x, y, n);
```

For an example, see Pentominoes.

4.4.12 Element constraints

Element constraints generalize array access to integer variables. For example,

```
IntArgs c(5, 1,4,9,16,25);
element(home, c, x, y);
```

constrains the integer variable `y` to be the element of the array `c` at index `x` (where the array starts at index `0` as is common in C++).

The index variable `x` is always an integer variable, but the array `c` can also be an array of integer variables, Boolean variables, or an array of integers between `0` and `1`. The result variable `y` must be a Boolean variable or an integer between `0` and `1` if the array is an array of Boolean variables. It can be a Boolean variable if all integer values in the array are between `0` and `1`.

Even if bounds propagation is requested for the `element` constraint, the propagators for `element` always perform domain reasoning on the index variable (see also GCCAT: `elem`, `element`). For examples, see the case study in Chapter 15 or the examples Steel-mill slab design problem and Travelling salesman problem (TSP).

Tip 4.9 (Shared integer arrays)**.** When checking the documentation for Element constraints it might come at a surprise that element constraints do not take integer argument arrays of type `IntArgs` but *shared integer arrays* of type `IntSharedArray` as argument. The reason is that the very same shared integer array can be used for several element constraints.

Consider the following example

```
IntArgs c(5, 1,4,9,16,25);
element(home, c, x, y);
element(home, c, a, b);
```

where `x`, `y`, `a`, and `b` are integer variables. Then, each time an element constraint is posted, a new shared integer array is created implicitly (that is, in the example above, two arrays are created). If the integer array is large or many element constraints are posted, it is beneficial to explicitly create a shared integer array, such as in:

```
IntArgs c(5, 1,4,9,16,25);
IntSharedArray cs(c);
element(home, cs, x, y);
element(home, cs, a, b);
```

Here only a single shared arrays is created and is used for both propagators created for posting the element constraints.

What is also obvious from the first example is that integer argument arrays of type `IntArgs` can automatically be coerced to integer shared arrays of type `IntSharedArray`. Hence, if performance is not that important, you even do not need to know that shared integer arrays exist.

For an example that uses shared integer arrays together with element constraints, see Chapter 21 and Crossword puzzle. ◀

Tip 4.10 (Shared arrays also provide STL-style iterators)**.** Shared arrays also support STL-style (Standard Template Library) iterators, similar to other arrays provided by Gecode, see Section 4.2.3. ◀

4.4.13 Extensional constraints

Extensional constraints (also known as user-defined or ad-hoc constraints) provide constraints that are specified in *extension*. The extension can be either defined by a `DFA` (deterministic finite automaton) or a tuple set `TupleSet`. DFAs can also be specified conveniently by regular expressions, see Section 7.4.

Note that both DFAs and tuple sets are proper Gecode data structures. When storing a DFA or a tuple set as a space member, the member must be updated with an `update` function during cloning exactly as described for variables in Section 4.1.8.[2]

Deterministic finite automata. Suppose we want to plan the activities of an evening that follows the Swedish drinking protocol: you may have as many drinks as you like, but now and then you sing a song after which you have to have a drink. We want to constrain an array of activities (Boolean or integer variables) such that the activities (drinking and singing) follow the protocol.

[2] In most cases, the data structure is actually shared between spaces created by cloning. Only when a non-shared clone is requested (for parallel search), a copy of the data structure is created. This is the reason why updating is required.

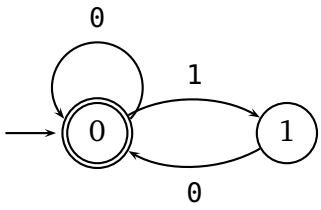


Figure 4.4: A DFA for the Swedish drinking protocol

The DFA in Figure 4.4 specifies legal sequences of activities according to the Swedish drinking protocol, where the state 0 is the start state and also the final state. The symbol `0` corresponds to drinking, whereas `1` corresponds to singing. That is, the sequence of activities must be a string of `0`s and `1`s accepted by the DFA.

The `DFA` `d` is initialized by

```
DFA::Transition t[] = {{0, 0, 0}, {0, 1, 1},
                       {1, 0, 0}, {-1, 0, 0}};
int f[] = {0,-1};
DFA d(0, t, f);
```

The array of transitions `t` is initialized by triples of integers (of type `DFA::Transition`). A triple $\{a,s,b\}$ defines a transition from state a to state b with symbol s. States are denoted by non-negative integers and symbols are integer values (as always, restricted to integer values that can be taken on by an integer variable, see Section 4.1.2). A transition where a is -1 marks the last transition in a transition array. The array of final states `f` lists all final states of the DFA, where the array is terminated by -1. The first argument of the constructor of the DFA defines the start state.

Constraining an array of variables for four activities to the Swedish drinking protocol is done by

```
BoolVarArray x(home, 4, 0, 1);
extensional(home, x, d);
```

Note that the same DFA would also work with an array of integer variables.

The propagator for the `extensional` constraint is domain consistent and is based on [36].

Examples that use regular expressions for defining DFAs can be found in Section 7.4.

Tuple sets. Constraints can also be defined by a list of tuples, where each tuple defines one solution of the extensional constraint. For example, the following defines the Swedish drinking protocol for three activities by a list of tuples:

```
TupleSet t;
t.add(IntArgs(3, 0,0,0));
t.add(IntArgs(3, 0,1,0));
t.add(IntArgs(3, 1,0,0));
t.finalize();
```

Note that before a tuple set can be used by a post function, it must be finalized as shown above. If a not-yet finalized tuple set is used for posting a constraint, an exception of type `Int::NotYetFinalized` is thrown.

The propagators for the `extensional` constraint are domain consistent and are based on [7] and [8] (see also GCCAT: `in_relation`).

Constraining a sequence of variables to the Swedish drinking protocol using a tuple set is done by

```
BoolVarArray x(home, 3, 0, 1);
extensional(home, x, t);
```

For examples of extensional constraints using tuple sets, see Chapter 20, Black hole patience, and Kakuro.

4.4.14 Sorted constraints

Sorted constraints relate an integer variable array to an array obtained by sorting the array. For example,

```
sorted(home, x, y);
```

constrains `y` to be `x` (of same size) sorted in increasing order. The more general variant features an additional integer variable array (again, of same size) `z` as in

```
sorted(home, x, y, z);
```

where `z` defines the sorting permutation, that is

$$x_i = y_{z_i} \qquad \text{for } 0 \leq i < |x|$$

The propagator for `sorted` is bounds consistent and is based on [31] (see also GCCAT: `sort`, `sort_permutation`).

4.4.15 Bin-packing constraints

Bin packing constraints constrain how items can be packed into bins.

Single-dimensional bin-packing constraints. The bin-packing constraint is posted as

```
binpacking(home, l, b, s);
```

where `l` is an array of integer variables (the *load* variables), `b` is an array of integer variables (the *bin* variables), and `s` is an array of non-negative integers (the item *sizes*).

The load variables `l` determine the load $\mathtt{l}_j$ of each bin j ($0 \leq j < |\mathtt{l}|$) and the bin variables `b` determine for each item i ($0 \leq i < |\mathtt{b}|$) into which bin $\mathtt{b}_i$ it is packed. The size of an item i ($0 \leq i < |\mathtt{b}|$) is defined by its item size $\mathtt{s}_i$. Naturally, the number of bin variables and item sizes must coincide ($|\mathtt{b}| = |\mathtt{s}|$).

The bin-packing constraint enforces that all items are packed into bins

$$\mathtt{b}_i \in \{0, \ldots, |\mathtt{l}| - 1\} \qquad \text{for } 0 \leq i < |\mathtt{b}|$$

and that the load of each bin corresponds to the items packed into it

$$\mathtt{l}_j = \sum_{\{i \in \{0,\ldots,|\mathtt{b}|-1\} \mid \mathtt{b}_j = i\}} \mathtt{s}_i \qquad \text{for } 0 \leq j < |\mathtt{l}|$$

The constraint is implemented by the propagator introduced in [53] (see also GCCAT: `bin_packing`, `bin_packing_capa`). For an example using the bin-packing constraint and CDBF (complete decreasing best fit) [17] as a specialized branching for bin-packing, see Chapter 19 and Bin packing.

Multi-dimensional bin-packing constraints. The multi-dimensional bin-packing constraint is posted as

```
binpacking(home, d, l, b, s, c);
```

where `d` is a positive integer (the *dimension*), `l` is an array of integer variables (the *load* variables), `b` is an array of integer variables (the *bin* variables), `s` is an array of non-negative integers (the item *sizes*), and `c` is an array of non-negative integers (the bin *capacities*).

In the following n refers to the number of items and m refers to the number of bins. The bin variables `b` determine for each item i ($0 \leq i < n$) into which bin $\mathtt{b}_i$ it is packed. The load variables `l` determine the load $\mathtt{l}_{j \cdot \mathtt{d} + k}$ for each bin j ($0 \leq j < m$) and dimension k ($0 \leq k < \mathtt{d}$). The size of an item i ($0 \leq i < n$) in dimension k ($0 \leq k < \mathtt{d}$) is defined by the item size $\mathtt{s}_{i \cdot \mathtt{d} + k}$. The capacity of all bins j ($0 \leq j < m$) in dimension k ($0 \leq k < \mathtt{d}$) is defined by $\mathtt{c}_k$. Naturally, the number of bin variables, load variables, item sizes, and capacities must satisfy that $|\mathtt{b}| = n$, $|\mathtt{l}| = m \cdot \mathtt{d}$, $|\mathtt{s}| = n \cdot \mathtt{d}$, and $|\mathtt{c}| = \mathtt{d}$.

The multi-dimensional bin-packing constraint enforces that all items are packed into bins

$$\mathtt{b}_i \in \{0, \ldots, m - 1\} \qquad \text{for } 0 \leq i < n$$

and that the load of each bin corresponds to the items packed into it for each dimension

$$\mathtt{l}_{j \cdot \mathtt{d} + k} = \sum_{\{i \in \{0,\ldots,n-1\} \mid \mathtt{b}_{j \cdot \mathtt{d} + k} = i\}} \mathtt{s}_{i \cdot \mathtt{d} + k} \qquad \text{for } 0 \leq j < m, 0 \leq k < \mathtt{d}$$

Furthermore, the load variables must satisfy the capacity constraints

$$\mathtt{l}_{j\cdot \mathtt{d}+k} \leq \mathtt{c}_k \qquad \text{for } 0 \leq j < m, 0 \leq k < \mathtt{d}$$

In addition to posting propagators, the post function

```
IntSet m = binpacking(home, d, l, b, s, c);
```

returns an integer set `m` of type `IntSet`. The set `m` contains a maximal number of conflicting items that must be packed into pairwise distinct bins where the items are chosen to maximize the conflict with other items. This information can be used for symmetry breaking.

Important. Posting the constraint (not propagating it) has exponential complexity in the number of items. This is due to the use of the Bron-Kerbosch algorithm [10, 11] for finding all sets of conflicting items.

The constraint is implemented by the decomposition introduced in [20] using a single-dimensional bin-packing constraint for each dimension together with derived constraints capturing capacity conflicts. For an example using the multi-dimensional bin-packing constraint see Multi-dimensional bin packing.

4.4.16 Geometrical packing constraints

Geometrical packing constraints constrain how rectangles can be packed such that no two rectangles from a collection of rectangles overlap.

If `x` and `y` are integer variable arrays and `w` and `h` are integer arrays (where all arrays must be of the same size), then

```
nooverlap(home, x, w, y, h);
```

propagates that the rectangles defined by coordinates $\langle \mathtt{x}_i, \mathtt{y}_i \rangle$, widths $\mathtt{w}_i$, and heights $\mathtt{h}_i$ for $0 \leq i < |\mathtt{x}|$ do not overlap. That is, the following constraint is enforced (see also Section 7.1.2 for a picture):

$$\left(\mathtt{x}_i + \mathtt{w}_i \leq \mathtt{x}_j\right) \vee \left(\mathtt{x}_j + \mathtt{w}_j \leq \mathtt{x}_i\right) \vee \left(\mathtt{y}_i + \mathtt{h}_i \leq \mathtt{y}_j\right) \vee \left(\mathtt{y}_j + \mathtt{h}_j \leq \mathtt{y}_i\right)$$

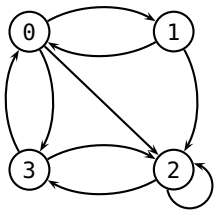
Note that the width or the height of a rectangle can be zero. In this case, the rectangle does not occupy any space. However no other rectangle is allowed to be placed where the zero-sized rectangle is placed.

Rectangles can also be modeled as optional through a Boolean variable $\mathtt{m}_i$ for each rectangle i. If the Boolean variable $\mathtt{m}_i$ is `1` the rectangle is mandatory and considered by the packing constraint, if it is `0`, the rectangle is ignored.

With an array of Boolean variables `m` the constraint taking optional rectangles into account is posted by

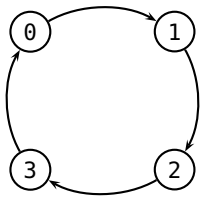
```
nooverlap(home, x, w, y, h, m);
```

The arrays defining the dimensions (that is, `w` and `h` for rectangles) can also be arrays of integer variables, where its values are constrained to be non-negative. In this case, the constraint post functions expects both a start and end coordinate. That is, by posting



$x_0 \in \{1,2,3\}, x_1 \in \{0,2\}, x_2 \in \{2,3\}, x_3 \in \{0,2\}$

(a) Before propagation



$x_0 \in \{1\}, x_1 \in \{2\}, x_2 \in \{3\}, x_3 \in \{0\}$

(b) After propagation

Figure 4.5: Representing edges and propagating `circuit`

```
nooverlap(home, x0, w, x1, y0, h, y1);
```

it is enforced that the rectangles defined by the start coordinate $\langle \mathtt{x0}_i, \mathtt{y0}_i \rangle$, the dimension $\langle \mathtt{w}_i, \mathtt{h}_i \rangle$, and the end coordinate $\langle \mathtt{x1}_i, \mathtt{y1}_i \rangle$ do not overlap. The end coordinates are *not* constrained to be the sum of the start coordinates and dimensions. That is, one has to explicitly post the linear constraints such that

$$(\mathtt{x0}_i + \mathtt{w}_i = \mathtt{x1}_i) \wedge (\mathtt{y0}_i + \mathtt{h}_i = \mathtt{y1}_i)$$

The constraints are implemented by a naive propagator (considering pairwise no-overlap between rectangles including constructive disjunction, see also GCCAT: `diffn`), this will change in the future. For an example using the no-overlap constraint, see Packing squares into a rectangle.

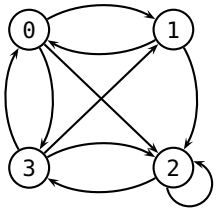
4.4.17 Circuit and Hamiltonian path constraints

The `circuit` and `path` constraints (see Graph constraints) use values of variables in an integer variable array `x` as edges: if $j \in \mathtt{x}_i$, the corresponding graph contains the edge $i \rightarrow j$ for $0 \leq i, j < |\mathtt{x}|$. Obviously, the graph has the nodes 0 to $|\mathtt{x}| - 1$.

Circuit constraints. Assume that `x` is an integer variable array (`circuit` does not support Boolean variables). Then,

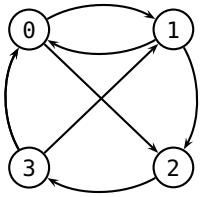
```
circuit(home, x);
```

constrains the values of `x` such that their corresponding edges form a Hamiltonian circuit (see also GCCAT: `circuit`). For an example before and after propagation of `circuit` see Figure 4.5. For an example, see Chapter 18 and n-Knights tour (model using circuit).



$$x_0 \in \{1,2,3\}, x_1 \in \{0,2\}, x_2 \in \{2,3\}, x_3 \in \{0,1,2\}$$
$$y_i \in \{-100,\ldots,100\} \quad (0 \le i < 4)$$
$$z \in \{-100,\ldots,100\},$$

(a) Before propagation



$$x_0 \in \{1,2\}, x_1 \in \{0,2\}, x_2 \in \{3\}, x_3 \in \{0,1\}$$
$$y_0 \in \{3,5\}, y_1 \in \{4,9\}, y_2 \in \{5\}, y_3 \in \{-7,8\}$$
$$z \in \{5,\ldots,27\},$$

(b) After propagation

Figure 4.6: Representing edges and propagating `circuit` with cost

Common applications of `circuit` also require costs for the edges in the graph. Assume that the cost for an edge $i \to j$ from node i to node j is defined by the following matrix:

$i \to j$	$\cdot \to 0$	$\cdot \to 1$	$\cdot \to 2$	$\cdot \to 3$
$0 \to \cdot$	0	3	5	7
$1 \to \cdot$	4	0	9	6
$2 \to \cdot$	2	1	0	5
$3 \to \cdot$	-7	8	-2	0

Then, by

```
IntArgs c(4*4, 0, 3, 5, 7,
               4, 0, 9, 6,
               2, 1, 0, 5,
              -7, 8,-2, 0);
circuit(home, c, x, y, z);
```

the integer variables `x` are constrained to the values forming the circuit as above, while the integer variables `y` define the cost of the edge for each node (these variables can be

omitted), and the integer variable z defines the total cost of the edges in the circuit. Note that the matrix interface as described in Section 7.2 might come in handy for setting up the cost matrix.

Figure 4.6 shows a simple example for propagation `circuit` with cost where the cost matrix from above is used. For an example, see Travelling salesman problem (TSP).

Hamiltonian path constraints. The `path` constraint (see Graph constraints) is similar to the `circuit` constraint and enforces that nodes in a graph from a Hamiltonian path. Assume that `x` is an integer variable array (`path` does not support Boolean variables) and `s` (for start) and `e` (for end) are integer variables. Then,

```
path(home, x, s, e);
```

constrains the values of `x`, `s`, and `e` such that their corresponding edges form a Hamiltonian path that starts at node x_s and ends at node x_e (the value of the variable x_e is always $|x|$).

As an example, assume that the integer variable array `x` has three elements (that is, $|x| = 3$) with values between 0 and 3. Then all solutions to

```
path(home, x, s, e);
```

are as follows:

x	s	e
⟨1,2,3⟩	0	2
⟨1,3,0⟩	2	1
⟨2,0,3⟩	1	2
⟨2,3,1⟩	0	1
⟨3,0,1⟩	2	0
⟨3,2,0⟩	1	0

The `path` constraint provides, similar to `circuit`, variants for costs for edges in a Hamiltonian path, see Graph constraints.

4.4.18 Scheduling constraints

This section provides an overview of scheduling constraints.

Important. The support for scheduling constraints is still experimental. It is not yet complete, as it still lacks modeling abstractions and specialized branchings. The existing code is tested and efficient, but the interfaces may change in future versions.

Unary resource constraints

A unary resource constraint models that a number of tasks to be executed on a single resource do not overlap, where each task is defined by its start time (an integer variable), its duration (an integer or integer variable), and possibly its end time (if the duration is a variable). Unary resource constraints are also known as disjunctive scheduling constraints.

For example, assume that four tasks with durations `2`, `7`, `4`, and `11` are to be executed on the same resource where the start times are specified by an array of integer variables (of course, with four variables). Then, posting

```
IntArgs d(4, 2,7,4,11);
unary(home, s, d);
```

constrains the start times in `s` such that the execution of none of the tasks overlaps in time (see Scheduling constraints).

All propagators implementing `unary` use overload-checking, detectable precedences, not-first-not-last, and edge-finding, following [64] (see also GCCAT: `disjunctive`).

A common variant for unary resource constraints is where tasks can be optional: each task t has a Boolean variable b attached to it. If $b = 1$ then the task is *mandatory* and is scheduled on the resource. If $b = 0$ then the task is *excluded* and is not scheduled. Otherwise, the task is said to be *optional*. Assume that `b` refers to an array of Boolean variables also of size 4, then

```
IntArgs d(4, 2,7,4,11);
unary(home, s, d, b);
```

posts a propagator that constrains the start times `s` (of course, only if a task is mandatory) as well the Boolean variables in `b` (if a task becomes excluded as otherwise no feasible schedule would exist).

Tasks with flexible duration. The duration of a task can also be given as an integer variable instead of a constant integer. In this case, we say that the tasks are *flexible*. In addition to the flexible duration, the `unary` constraint also requires variables for the end time of each task.

Given variable arrays `s`, `d`, and `e` for the start times, durations, and end times, a unary resource constraint is posted as

```
unary(home, s, d, e);
```

However, the additional constraint for each task i that $\texttt{s}[i] + \texttt{d}[i] = \texttt{e}[i]$ is not enforced automatically. Therefore, a model typically must contain additional constraints like

```
for (int i=0; i<s.size(); i++)
  post(home, s[i]+d[i]==e[i]);
```

The `unary` post function also exists in a variant with flexible, optional tasks.

Cumulative scheduling constraints

Gecode provides two generalizations of the unary resource scheduling constraint. The first one, called `cumulative`, models a resource where tasks can overlap. The resource has a limited *capacity*, and each task requires a certain *resource usage*. At each point in time, the sum of the resource usages of all tasks running at that point must not exceed the capacity. The second generalization, called `cumulatives`, deals with several cumulative resources at once.

Cumulative single-resource constraint. The single-resource constraint `cumulative` has nearly the same interface as `unary`. The only difference is a parameter `c` specifying the resource capacity, and an additional integer array `u` for the resource usage of each task. Assuming that `s` and `d` give the start times and durations as before, the following models a resource where two tasks can overlap, and the first three tasks require one unit of the resource, while the last task requires two:

```
IntArgs u(4, 1,1,1,2);
cumulative(home, 2, s, d, u);
```

The capacity can be an integer variable or a nonnegative integer. The resource usage must be stricly positive. As for unary, there exist versions with optional, flexible, and flexible and optional tasks.

The propagators implementing the `cumulative` constraint use time-tabling, overload checking [63], and edge-finding [62] (see also GCCAT: `cumulative`).

Cumulative multi-resource constraint. Given a *set* of resources that have some specified usage limit and a set of tasks that must be placed on these resources according to start-times, durations, resource usage, and resource compatibility, the `cumulatives` constraint (see Scheduling constraints) can be used for the placement of these tasks. The limit for the resources can be either a maximum or a minimum, and the resource usage of a task can be positive, negative, or zero. The limit is only valid over the intervals where there is at least one task assigned on that particular resource.

Consider the following code.

```
cumulatives(home, resource, start, duration, end, height,
            limit, atmost);
```

This code posts a constraint over a set of tasks T, where each task T_i is defined by $\langle \texttt{resource}_i, \texttt{start}_i, \texttt{duration}_i, \texttt{end}_i, \texttt{height}_i \rangle$. The `resource` component indicates the potential resources that the task can use; `start`, `duration`, and `end` indicate when the task can occur; and finally the `height` component indicates the amount of resource the task uses (or "provides" in the case of a negative value). The resource R_i is defined by the limit $\texttt{limit}_i$ and the parameter `atmost`. The latter is common for all resources, and indicates whether the limits are maximum limits (`atmost` is true) or minimum limits (`atmost` is false).

As for flexible tasks in Section 4.4.18, the `cumulatives` constraint does not enforce that $\texttt{start}_i + \texttt{duration}_i = \texttt{end}_i$. This additional constraint must be posted manually.

The parameters `start` and `end` are always integer variable arrays; `resource`, `duration`, and `height` can be either integer variable arrays or integer arrays; and `limit` is always an array of integers.

For an example using `cumulatives` see Packing squares into a rectangle, where the `cumulatives` constraints is used to model packing a set of squares. For an insightful discussion of how to use `cumulatives` for modeling, see [4] (the propagator for `cumulative` is implemented following this paper, see also GCCAT: `cumulatives`).

4.4.19 Value precedence constraints

Value precedence constraints over integer variables enforce that a value precedes another value in an array of integer variables. By

```
precede(home, x, s, t);
```

where `x` is an array of integer variables and both `s` and `t` are integers, the following is enforced: if there exists j $(0 \leq j < |\texttt{x}|)$ such that $\texttt{x}_j = \texttt{t}$, then there must exist i with $i < j$ such that $\texttt{x}_i = \texttt{s}$. This is equivalent to:

1. $\texttt{x}_0 \neq \texttt{t}$, and

2. if $\texttt{x}_j = \texttt{t}$ then $\bigvee_{i=0}^{j-1} \texttt{x}_i = \texttt{s}$ for $1 \leq j < |\texttt{x}|$.

A generalization is available for precedences between several integer values. By

```
precede(home, x, c);
```

where `x` is an array of integer variables and `c` is an array of integers, it is enforced that $\texttt{c}_i$ precedes $\texttt{c}_{i+1}$ in `x` for $0 \leq i < |\texttt{c}| - 1$. That is

1. $\texttt{x}_0 \neq \texttt{c}_{k+1}$ for $0 \leq k < |\texttt{c}| - 1$, and

2. if $\texttt{x}_j = \texttt{c}_{k+1}$ then $\bigvee_{i=0}^{j-1} \texttt{x}_i = \texttt{c}_k$ for $1 \leq j < |\texttt{x}|$ and $0 \leq k < |\texttt{c}| - 1$.

The constraint is implemented by the domain consistent propagator introduced in [24] (see also GCCAT: `int_value_precede`, `int_value_precede_chain`), the paper also explains how to use the `precede` constraint for breaking value symmetries. For an example, see Schur's lemma.

4.5 Synchronized execution

Gecode offers support in Synchronized execution for executing a function (either a function or a static member function but not a member function) when integer or Boolean variables become assigned.

Assume that a function `c` (for continuation) is defined by

```
void c(Space& home) { ··· }
```

Then

```
wait(home, x, &c);
```

posts a propagator that waits until the integer or Boolean variable `x` (or, if `x` is an array of variables: all variables in `x`) is assigned. If `x` becomes assigned, the function `c` is executed with the current home space passed as argument.

Tip 4.11 (Failing a space)**.** If you want to fail a space `home` (for example when executing a continuation function as discussed above), you can do that by

```
home.fail();
```

◀

Assume that two functions `t` (for then) and `e` (for else) are defined by

```
void t(Space& home) { ··· }
void e(Space& home) { ··· }
```

Then

```
when(home, x, &t, &e);
```

creates a propagator that will be run exactly once when the Boolean variable `x` becomes assigned. If `x` becomes assigned to `1`, the function `t` is executed. If `x` becomes assigned to `0`, the function `e` is executed. Both functions get the current home space of the propagator passed as argument. The else-function is optional and can be omitted.

5 Set variables and constraints

This chapter gives an overview over set variables and set constraints in Gecode and serves as a starting point for using set variables. For the reference documentation, see Using integer set variables and constraints.

Overview. Section 5.1 details how set variables can be used for modeling. The sections Section 5.2 and Section 5.3 provide an overview of the constraints that are available for set variables in Gecode.

Important. Do not forget to add

```
#include <gecode/set.hh>
```

to your program when you want to use set variables. Note that the same conventions hold as in Chapter 4.

5.1 Set variables

Set variables in Gecode model sets of integers and are instances of the class `SetVar`.

Tip 5.1 (Still do not use views for modeling)**.** Just as for integer variables, you should not feel tempted to use views of set variables (such as `SetView`) for modeling. Views can only be used for implementing propagators and branchers, see Part P and Part B. ◀

Representing set domains as intervals. The domain of a set variable is a set of sets of integers (in contrast to a simple set of integers for an integer variable). For example, assume that the domain of the set variable x is the set of subsets of $\{1,2,3\}$:

$$\{\ \{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\ \}$$

Set variable domains can become very large – the set of subsets of $\{1,\ldots,n\}$ has 2^n elements. Gecode (like most constraint solvers) therefore approximates set variable domains by a set interval $[l\,..\,u]$ of a lower bound l and an upper bound u. The interval $[l\,..\,u]$ denotes the set of sets $\{s \mid l \subseteq s \subseteq u\}$. The lower bound l (commonly referred to as greatest lower bound or *glb*) contains all elements that are *known* to be included in the set, whereas the

upper bound u (commonly referred to as least upper bound or *lub*) contains the elements that *may be* included in the set. As only the two interval bounds are stored, this representation is space-efficient. The domain of x from the above example can be represented as $[\{\} .. \{1,2,3\}]$.

Set intervals can only approximate set variable domains. For example, the domain

$$\{\ \{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\}\ \}$$

cannot be captured exactly by an interval. The closest interval would be $[\{\} .. \{1,2,3\}]$. In order to get a closer approximation of set variable domains, Gecode additionally stores cardinality bounds. We write $\#\left[i .. j\right]$ to express that the cardinality is at least i and at most j. The set interval bounds $[\{\} .. \{1,2,3\}]$ together with cardinality bounds $\#[1 .. 2]$ represent the above example domain exactly.

Creating a set variable. New set variables are created using a constructor. A new set variable `x` is created by

```
SetVar x(home, IntSet::empty, IntSet(1, 3), 1, 2);
```

This declares a variable `x` of type `SetVar` in the space `home`, creates a new set variable implementation with domain $[\{\} .. \{1,2,3\}], \#[1 .. 2]$, and makes `x` refer to the newly created set variable implementation.

There are several overloaded versions of the constructor, you can for example omit the cardinality bounds if you do not want to restrict the cardinality. You find the full interface in the reference documentation of the class `SetVar`. An attempt to create a set variable with an empty domain throws an exception of type `Set::VariableEmptyDomain`.

As for integer and Boolean variables, the default and copy constructors do not create new variable implementations. Instead, the variable does not refer to any variable implementation (default constructor) or to the same variable implementation (copy constructor). For example in

```
SetVar x(home, IntSet::empty, IntSet(1, 3), 1, 2);
SetVar y(x);
SetVar z;
z=y;
```

the variables `x`, `y`, and `z` all refer to the same set variable implementation.

Limits for set elements. All set variable bounds are subsets of the *universe*, defined as

$$[\texttt{Set::Limits::min} .. \texttt{Set::Limits::max}]$$

The universe is symmetric: $-\texttt{Set::Limits::min} = \texttt{Set::Limits::max}$. Furthermore, the cardinality of a set is limited to the unsigned integer interval

$$\#[0 .. \texttt{Set::Limits::card}]$$

The limits have been chosen such that an integer variable can hold the cardinality. This means that the maximal element of a set variable is `Int::Limits::max`$/2-1$. The limits are defined in the namespace `Set::Limits`.

Any attempt to create a set variable with values outside the defined limits throws an exception of type `Set::OutOfLimits`.

Tip 5.2 (Small variable domains are still beautiful)**.** Just like integer variables (see Tip 4.3), set variables do not have a constructor that creates a variable with the largest possible domain. And again, one has to worry and the omission is deliberate to make you worry. So think about the initial domains carefully when modeling. ◀

Variable access functions. You can access the current domain of a set variable `x` using member functions such as `x.cardMax()`, returning the upper bound of the cardinality, or `x.glbMin()`, returning the smallest element of the lower bound. Furthermore, you can print a set variable's domain using the standard output operator `<<`.

Iterating variable domain interval bounds. For access to the interval bounds of a set variable, Gecode provides three value iterators and corresponding range iterators. For example, the following loop

```
for (SetVarGlbValues i(x); i(); ++i)
  std::cout << i.val() << ' ';
```

uses the value iterator `i` to print all values of the greatest lower bound of the domain of `x` in increasing order. If `x` is assigned, this of course corresponds to the value of `x`. Similarly, the following loop

```
for (SetVarLubRanges i(x); i(); ++i)
  std::cout << i.min() << ".." << i.max() << ' ';
```

uses the range iterator `i` to print all ranges of the least upper bound of the domain of `x`. The third kind of iterator, `SetVarUnknownValues` or `SetVarUnknownRanges`, iterate the values resp. ranges that are still unknown to be part or not part of the set, that is $u \setminus l$ for the domain $[l \mathinner{..} u]$.

When to inspect a variable. The same restrictions hold as for integer variables (see Section 4.1.7). The important restriction is that one must not change the domain of a variable (for example, by posting a constraint on that variable) while an iterator for that variable is being used.

Updating variables. Set variables behave exactly like integer variables during cloning of a space. A set variable is updated by

```
x.update(home, share, y);
```

where share is the Boolean argument passed as argument to copy() and y is the variable from which x is to be updated. While home is the space x belongs to, y belongs to the space which is being cloned.

Variable and argument arrays. Set variable arrays can be allocated using the class SetVarArray. The constructors of this class take the same arguments as the set variable constructors, preceded by the size of the array. For example,

```
SetVarArray x(home, 4, IntSet::empty, IntSet(1, 3));
```

creates an array of four set variables, each with domain $[\{\} .. \{1,2,3\}]$.

To pass temporary data structures as arguments, you can use the SetVarArgs class (see Argument arrays). Some set constraints are defined in terms of arrays of sets of integers. These can be passed using IntSetArgs (see Argument arrays). Set variable argument arrays support the same operations introduced in Section 4.2.2.

5.2 Constraint overview

This section introduces the different groups of constraints over set variables available in Gecode. The section serves only as an overview. For the details and the full list of available post functions, the section refers to the relevant reference documentation.

Reified constraints. Several set constraints also exist as a reified variant. Whether a reified version exists for a given constraint can be found in the reference documentation. If a reified version does exist, the reification information combining the Boolean control variable and an optional reification mode is passed as the last non-optional argument, see Section 4.3.4.

Tip 5.3 (Reification by decomposition)**.** If your model requires reification of a constraint for which no reified version exists in the library, you can often *decompose* the reification. For example, to reify the constraint $x \cup y = z$ to a control variable b, you can introduce an auxiliary variable z0 and post the two constraints

```
rel(home, x, SOT_UNION, y, SRT_EQ, z0);
rel(home, z0, SRT_EQ, z, b);
```

◀

5.2.1 Domain constraints

Domain constraints restrict the domain of a set variable using a set constant (given as a single integer, an interval of two integers or an IntSet), depending on set relation types of type SetRelType (see Using integer set variables and constraints). Figure 5.1 lists the available set relation types and their meaning. The relations SRT_LQ, SRT_LE, SRT_GQ, and SRT_GR

SRT_EQ	equality ($=$)	SRT_NQ	disequality ($\neq$)
SRT_LQ	lex. less than or equal	SRT_LE	lex. less than
SRT_GQ	lex. greater than or equal	SRT_GR	lex. greater than
SRT_SUB	subset ($\subseteq$)	SRT_SUP	superset ($\supseteq$)
SRT_DISJ	disjointness ($\|\|$)	SRT_CMPL	complement ($\bar{\cdot}$)

Figure 5.1: Set relation types

establish a total order based on the lexicographic order of the characteristic functions of the two sets.

For example, the constraints

```
dom(home, x, SRT_SUB, 1, 10);
dom(home, x, SRT_SUP, 1, 3);
dom(home, y, SRT_DISJ, IntSet(4, 6));
```

result in the set variable x being a subset of $\{1,\ldots,10\}$ and a superset of $\{1,2,3\}$, while 4, 5, and 6 are not elements of the set y. The domain constraints for set variables support reification. Both x and y can also be arrays of set variables where each array element is constrained accordingly (but no reification is supported).

In addition to the above constraints,

```
cardinality(home, x, 3, 5);
```

restricts the cardinality of the set variable x to be between 3 and 5. x can also be an array of set variables.

The domain of a set variable x can be constrained according to the domain of another variable set d by

```
dom(home, x, d);
```

Here, x and d can also be arrays of set variables.

For examples using domain constraints, see Airline crew allocation, as well as the redundant constraints in Golf tournament.

5.2.2 Relation constraints

Relation constraints enforce relations between set variables and between set and integer variables, depending on the set relation types introduced above.

For set variables x and y, the following constrains x to be a subset of y:

```
rel(home, x, SRT_SUB, y);
```

If x is a set variable and y is an integer variable, then

`SOT_UNION`	union ($\cup$)	`SOT_INTER`	intersection ($\cap$)
`SOT_DUNION`	disjoint union ($\uplus$)	`SOT_MINUS`	set minus ($\setminus$)

Figure 5.2: Set operation types

```
rel(home, x, SRT_SUP, y);
```

constrains `x` to be a superset of the singleton set {`y`}, which means that `y` must be an element of `x`.

The last form of set relation constraint uses an integer relation type (see Figure 4.1) instead of a set relation type. This constraint restricts *all* elements of a set variable to be in the given relation to the value of an integer variable. For example,

```
rel(home, x, IRT_GR, y);
```

constrains all elements of the set variable `x` to be strictly greater than the value of the integer variable `y` (see also GCCAT: `eq_set`, `in`, `in_set`, `not_in`).

Gecode provides reified versions of all set relation constraints. For an example, see Golf tournament and Chapter 17.

5.2.3 Set operations

Set operation/relation constraints perform set operations according to the type shown in Figure 5.2 and relate the result to a set variable. For example,

```
rel(home, x, SOT_UNION, y, SRT_EQ, z);
```

enforces the relation $x \cup y = z$ for set variables `x`, `y`, and `z`. For an array of set variables `x`,

```
rel(home, SOT_UNION, x, y);
```

enforces the relation

$$\bigcup_{i=0}^{|x|-1} x_i = y$$

Instead of set variables, the relation constraints also accept `IntSet` arguments as set constants. There are no reified versions of the set operation constraints (you can decompose using reified relation constraints on the result, see Tip 5.3).

Set operation constraints are used in most examples that contain set variables, such as Airline crew allocation or Generating Hamming codes.

5.2.4 Element constraints

Element constraints generalize array access to set variables. The simplest version of `element` for set variables is stated as

```
element(home, x, y, z);
```

for an array of set variables or constants `x`, an integer variable `y`, and a set variable `z`. It constrains `z` to be the element of array `x` at index `y` (where the index starts at `0`).

A further generalization uses a *set variable* as the index, thus selecting several sets at once. The result variable is constrained to be the union, disjoint union, or intersection of the selected set variables, depending on the set operation type argument. For example,

```
element(home, SOT_UNION, x, y, z);
```

for set variables `y` and `z` and an array of set variables `x` enforces the following relation:

$$z = \bigcup_{i \in y} x_i$$

Note that generalized element constraints follow the usual semantics of set operations if the index variable is the empty set: an empty union is the empty set, whereas an empty intersection is the full universe. Because of this semantics, the `element` constraint has an optional set constant argument so that you can specify the universe (i.e., usually the full set of elements your problem deals with) explicitly. For an example of a set element constraint, see Golf tournament and Chapter 17.

5.2.5 Constraints connecting set and integer variables

Most models that involve set variables also involve integer variables. In addition to the set relation constraints that accept integer variables (interpreting them as singleton sets), Connection constraints to integer variables provide the necessary interface for models that use both set variables and integer or Boolean variables.

The most obvious constraint connecting integer and set variables is the cardinality constraint:

```
cardinality(home, x, y);
```

It states that the integer variable `y` is equal to the cardinality of the set variable `x`.

Gecode provides constraints for the minimal and maximal elements of a set. The following code

```
min(home, x, y);
```

constrains the integer variable `y` to be the minimum of the set `x`.

For an example of constraints connecting integer and set variables, see Steiner triples.

Weighted sets. The `weights` constraint assigns a weight to each possible element of a set variable `x`, and then constrains an integer variable `y` to be the sum of the weights of the elements of `x`. The mapping is given using two integer arrays, `e` and `w`. For example,

```
IntArgs e(6,   1, 3, 4, 5, 7, 9);
IntArgs w(6,  -1, 4, 1, 1, 3, 3);
weights(home, e, w, x, y);
```

enforces that `x` is a subset of $\{1,3,4,5,7,9\}$ (the set of elements), and that `y` is the sum of the weights of the elements in `x`, where the weight of the element `1` would be `-1`, the weight of `3` would be `4` and so on. Assigning `x` to the set $\{3,7,9\}$ would therefore result in `y` being assigned to $4+3+3=10$ (see also GCCAT: `sum_set`).

5.2.6 Set channeling constraints

Channel constraints link arrays of set variables, as well as set variables with integer and Boolean variables.

For an two arrays of set variables `x` and `y`,

```
channel(home, x, y);
```

posts the constraint

$$j \in \mathsf{x}_i \Leftrightarrow i \in \mathsf{y}_j \qquad \text{for } 0 \leq i < |\mathsf{x}| \qquad \text{and } 0 \leq j < |\mathsf{y}|$$

For an array of integer variables `x` and an array of set variables `y`,

```
channel(home, x, y);
```

posts the constraint

$$\mathsf{x}_i = j \Leftrightarrow i \in \mathsf{y}_j \qquad \text{for } 0 \leq i, j < |\mathsf{x}|$$

The channel between a set variable `y` and an array of Boolean variables `x`,

```
channel(home, x, y);
```

enforces the constraint

$$\mathsf{x}_i = 1 \Leftrightarrow i \in \mathsf{y} \qquad \text{for } 0 \leq i < |\mathsf{x}|$$

An array of integer variables `x` can be channeled to a set variable `y` using

```
rel(home, SOT_UNION, x, y);
```

which constrains `y` to be the set $\{\mathsf{x}_0, \ldots, \mathsf{x}_{|\mathsf{x}|-1}\}$. An alias for this constraint is defined in the modeling convenience library, see Section 7.5 and Section 7.7.

A specialized version of the previous constraint is

```
channelSorted(home, x, y);
```

which constrains `y` to be the set $\{\mathsf{x}_0, \ldots, \mathsf{x}_{|\mathsf{x}|-1}\}$, and the integer variables in `x` are sorted in increasing order ($\mathsf{x}_i < \mathsf{x}_{i+1}$ for $0 \leq i < |\mathsf{x}|$) (see also GCCAT: `link_set_to_booleans`).

5.2.7 Convexity constraints

Convexity constraints enforce that set variables are convex, which means that the elements form an integer interval. For example, the set $\{1,2,3,4,5\}$ is convex, while $\{1,3,4,5\}$ is not, as it contains a hole. The *convex hull* of a set s is the smallest convex set containing s ($\{1,2,3,4,5\}$ is the convex hull of $\{1,3,4,5\}$).

The constraint

```
convex(home, x);
```

states that the set variable x must be convex, and

```
convex(home, x, y);
```

enforces that the set variable y is the convex hull of the set variable x.

5.2.8 Sequence constraints

Sequence constraints enforce an order among an array of set variables x. Posting the constraint

```
sequence(home, x);
```

results in the sets x being pairwise disjoint, and furthermore $\max(x_i) < \min(x_{i+1})$ for all $0 \leq i < |x| - 1$. Posting

```
sequence(home, x, y);
```

additionally constrains the set variable y to be the union of the x.

For an example of sequence constraints, see Steiner triples.

5.2.9 Value precedence constraints

Value precedence constraints over set variables enforce that a value precedes another value in an array of set variables. By

```
precede(home, x, s, t);
```

where x is an array of set variables and both s and t are integers, the following is enforced: if there exists j ($0 \leq j < |x|$) such that $s \notin x_j$ and $t \in x_j$, then there must exist i with $i < j$ such that $s \in x_i$ and $t \notin x_i$.

A generalization is available for precedences between several integer values. By

```
precede(home, x, c);
```

where x is an array of set variables and c is an array of integers, it is enforced that c_k precedes c_{k+1} in x for $0 \leq k < |c| - 1$.

The constraint is implemented by the propagator introduced in [24] (see also GCCAT: set_value_precede), the paper also explains how to use the precede constraint for breaking value symmetries. For an example, see Golf tournament and Chapter 17.

5.3 Synchronized execution

Gecode offers support in Synchronized execution for executing a function (either a function or a static member function but not a member function) when set variables become assigned.

Assume that a function `c` (for continuation) is defined by

```
void c(Space& home) { ··· }
```

Then

```
wait(home, x, &c);
```

posts a propagator that waits until the set variable `x` (or, if `x` is an array of variables: all variables in `x`) is assigned. If `x` becomes assigned, the function `c` is executed with the current home space passed as argument.

6 Float variables and constraints

This chapter gives an overview over float variables and float constraints in Gecode. Just like Chapter 4 does for integer and Boolean variables, this chapter serves as a starting point for using float variables. For the reference documentation, please consult Using float variables and constraints.

Overview. Section 6.1 explains float values whereas Section 6.2 explains float variables. The sections Section 6.3 and Section 6.4 provide an overview of the constraints that are available for float variables in Gecode.

Important. Do not forget to add

```
#include <gecode/float.hh>
```

to your program when you want to use float variables. Note that the same conventions hold as in Chapter 4.

Tip 6.1 (Transcendental and trigonometric functions and constraints)**.** When compiling Gecode, by default transcendental and trigonometric functions and constraints are disabled. In order to enable them, you have to install additional third-party libraries and provide additional options to the configuration of Gecode, see Section 2.6.2.

To find out whether the functions and constraints are enabled, consult Tip 3.3. ◀

6.1 Float values and numbers

A *floating point value* (short, *float value*, see `FloatVal`) is represented as a closed interval of two *floating point numbers* (short, *float number*, see Float variables). That is, a float value is a closed interval $[a \,..\, b]$ which includes all real numbers $n \in \mathbb{R}$ such that $a \leq n$ and $n \leq b$. The float number type `FloatNum` is defined as **`double`**.

The reason why a float value is not represented by a single floating point number is that real numbers cannot be represented exactly and that operations on floating point numbers perform rounding. All operations (see below) on float values try to be as *accurate* as possible (so the interval $[a \,..\, b]$ for a float value is as small as possible) while being *correct* (no possible real number is ever excluded due to rounding). The classical reference on interval arithmetic is [34], for more information see also the Wikipedia article on interval arithmetic.

A float value x represented by the interval $[a .. b]$ provides many member functions such as `min()` (returning a) and `max()` (returning b), see `FloatVal`. The float value x is called *tight* if a equals b or if b is the smallest representable float number larger than a. If x is tight, `x.tight()` returns **`true`**.

A float value can be initialized from a single float number such as in

```
FloatVal x(1.0);
```

or from two float numbers such as in

```
FloatVal x(0.9999,1.0001);
```

Float numbers (and other numbers) are automatically cast to float values if needed, for example in

```
FloatVal x=1.0;
```

or

```
FloatVal x=1;
```

Predefined float values. The static member functions `pi_half()`, `pi()`, and `pi_twice()` of `FloatVal` return float values for $\frac{\pi}{2}$, π, and 2π respectively.

Arithmetic operators. For float values, the standard arithmetic operators `+`, `-`, `*`, and `/` and their assignment variants `+=`, `-=`, `*=`, and `/=` are defined with the obvious meaning.

Comparison operators. The usual float value comparisons `==`, `!=`, `<=`, `<`, `>`, and `>=` are provided with *entailment* semantics (or subsumption semantics).

For example, the comparison

```
x < y
```

returns **`true`** if and only if `x.max()<y.min()` returns **`true`**. That means, `x<y` returns **`false`** if either x is larger or equal than y or it cannot yet be decided: both x and y still represent values which are both smaller and greater or equal.

Functions on float values. Figure 6.1 lists the available functions on float values. The functions marked as default are always supported, the others only if Gecode has been built accordingly, see Tip 6.1.

6.2 Float variables

Float variables in Gecode model sets of real numbers and are instances of the class `FloatVar`.

Tip 6.2 (Still do not use views for modeling)**.** Just as for integer variables, you should not feel tempted to use views of float variables (such as `FloatView`) for modeling. Views can only be used for implementing propagators and branchers, see Part P and Part B. ◀

function	meaning	default
`max(x,y)`	maximum max(x, y)	✓
`min(x,y)`	minimum max(x, y)	✓
`abs(x)`	absolute value |x|	✓
`sqrt(x)`	square root $\sqrt{x}$	✓
`sqr(x)`	square x^2	✓
`pow(x,n)`	n-th power x^n	✓
`nroot(x,n)`	n-th root $\sqrt[n]{x}$	✓
`fmod(x,y)`	remainder of x/y	
`exp(x)`	exponential exp(x)	
`log(x)`	natural logarithm log(x)	
`sin(x)`	sine sin(x)	
`cos(x)`	cosine cos(x)	
`tan(x)`	tangent tan(x)	
`asin(x)`	arcsine arcsin(x)	
`acos(x)`	arccosine arccos(x)	
`atan(x)`	arctangent arctan(x)	
`sinh(x)`	hyperbolic sine sinh(x)	
`cosh(x)`	hyperbolic cosine cosh(x)	
`tanh(x)`	hyperbolic tangent tanh(x)	
`asinh(x)`	hyperbolic arcsine arcsinh(x)	
`acosh(x)`	hyperbolic arccosine arccosh(x)	
`atanh(x)`	hyperbolic arctangent arctanh(x)	

Figure 6.1: Functions on float values (x and y are float values; n is a non-negative integer)

Representing float domains as intervals. The domain of a float variable is represented exactly as a float value: a closed interval $[a \,..\, b]$ which represents all real numbers $n \in \mathbb{R}$ such that $a \leq n$ and $n \leq b$. A float variable is *assigned* if the interval $[a \,..\, b]$ is tight (see Section 6.1).[1]

Creating a float variable. New float variables are created using a constructor. A new float variable `x` is created by

```
FloatVar x(home, -1.0, 1.0);
```

This declares a variable `x` of type `FloatVar` in the space `home`, creates a new float variable implementation with domain $[-1.0 \,..\, 1.0]$, and makes `x` refer to the newly created float variable implementation.

You find the full interface in the reference documentation of the class `FloatVar`. An attempt to create a float variable with an empty domain throws an exception of type `Float::VariableEmptyDomain`.

As for integer variables, the default and copy constructors do not create new variable implementations. Instead, the variable does not refer to any variable implementation (default constructor) or to the same variable implementation (copy constructor). For example in

```
FloatVar x(home, -1.0, 1.0);
FloatVar y(x);
FloatVar z;
z=y;
```

the variables `x`, `y`, and `z` all refer to the same float variable implementation.

Limits. Float numbers range from `Float::Limits::min` to `Float::Limits::max` which also define the numbers that can represent float values and float variables. The limits are defined in the namespace `Float::Limits`.

Tip 6.3 (Small variable domains are still beautiful)**.** Just like integer variables (see Tip 4.3), float variables do not have a constructor that creates a variable with the largest possible domain. And again, one has to worry and the omission is deliberate to make you worry. So think about the initial domains carefully when modeling. ◀

Variable access functions. You can access the current domain of a float variable `x` using member functions such as `x.min()` and `x.max()`. Furthermore, you can print a float variable's domain using the standard output operator `<<`.

[1] Note that this means that a float variable is assigned even though its domain might still denote a set with more than one element. But this cannot be avoided as real numbers cannot be represented exactly.

Updating variables. Float variables behave exactly like integer variables during cloning of a space. A float variable is updated by

```
x.update(home, share, y);
```

where `share` is the Boolean argument passed as argument to `copy()` and `y` is the variable from which `x` is to be updated. While `home` is the space `x` belongs to, `y` belongs to the space which is being cloned.

Variable and argument arrays. Float variable arrays can be allocated using the class `FloatVarArray`. The constructors of this class take the same arguments as the float variable constructors, preceded by the size of the array. For example,

```
FloatVarArray x(home, 4, -1.0, 1.2);
```

creates an array of four float variables, each with domain $[-1.0 .. 1.2]$.

To pass temporary data structures as arguments, you can use the `FloatVarArgs` class. Some float constraints are defined in terms of arrays of float values. These can be passed using the `FloatValArgs` class. Float variable and value argument arrays support the same operations introduced in Section 4.2.2.

6.3 Constraint overview

This section introduces the different groups of constraints over float variables available in Gecode. The section serves only as an overview. For the details and the full list of available post functions, the section refers to the relevant reference documentation.

Reified constraints. Some float constraints (relation constraints, see Section 6.3.2, and linear constraints, see Section 6.3.4) also exist as a reified variant. If a reified version does exist, the reification information combining the Boolean control variable and an optional reification mode is passed as the last non-optional argument, see Section 4.3.4.

6.3.1 Domain constraints

Domain constraints constrain float variables and variable arrays to values from a given domain. For example, by

```
dom(home, x, -2.0, 12.0);
```

the values of the variable `x` (or of all variables in a variable array `x`) are constrained to be between the float numbers -2.0 and 12.0. Domain constraints also take float values as argument.

The domain of a float variable `x` can be constrained according to the domain of another float variable `d` by

`FRT_EQ`	equality ($=$)	`FRT_NQ`	disequality ($\neq$)
`FRT_LE`	strictly less inequality ($<$)	`FRT_LQ`	less or equal inequality ($\leq$)
`FRT_GR`	strictly greater inequality ($>$)	`FRT_GQ`	greater or equal inequality ($\geq$)

Figure 6.2: Float relation types

```
dom(home, x, d);
```

Here, `x` and `d` can also be arrays of float variables.

Domain constraints for a single variable also support reification.

6.3.2 Simple relation constraints

Simple relation constraints over float variables enforce relations between float variables and between float variables and float values. The relation depends on a float relation type `FloatRelType` (see Simple relation constraints over float variables). Figure 6.2 lists the available float relation types and their meaning.

Binary relation constraints. Assume that `x` and `y` are float variables. Then

```
rel(home, x, FRT_LE, y);
```

constrains `x` to be strictly less than `y`. Similarly, by

```
rel(home, x, FRT_LQ, 4.0);
```

`x` is constrained to be less than `4.0`. Both variants of `rel` also support reification.

Tip 6.4 (Weak propagation for strict inequalities ($<$, $>$) and disequality ($\neq$))**.** Unfortunately, the propagation for strict inequality ($<$, $>$) and disequality ($\neq$) relations is rather weak.

Consider the constraint `x<y` for float variables `x` and `y` with domains $[a \,..\, b]$ and $[c \,..\, d]$ respectively, where $b > d$ and $c < a$. Then one would like to propagate that `x` must be less than d and `y` must be larger than a. However, this would require that the domains of `x` and `y` after propagation are the *open* intervals $[a \,..\, d)$ and $(a \,..\, d]$. But only closed intervals can be represented by float variables!

Hence, the best propagation one could get is that the new domains are represented by the closed intervals $[a \,..\, d]$ and $[a \,..\, d]$ (the same propagation one would get for the constraint $\texttt{x} \leq \texttt{y}$ in this case). ◀

Constraints between variable arrays and a single variable. If `x` is a float variable array and `y` is an float variable, then

```
rel(home, x, FRT_LQ, y);
```

constrains all variables in `x` to be less than or equal to `y`. Likewise,

```
rel(home, x, FRT_GR, 7.0);
```

constrains all variables in `x` to be larger than `7.0`.

post function	constraint posted	default
`min(home, x, y, z);`	$\min(x,y)=z$	✓
`max(home, x, y, z);`	$\max(x,y)=z$	✓
`abs(home, x, y);`	$\lvert x\rvert=y$	✓
`mult(home, x, y, z);`	$x\cdot y=z$	✓
`div(home, x, y, z);`	$x/y=z$	✓
`sqr(home, x, y);`	$x^2=y$	✓
`sqrt(home, x, y);`	$\sqrt{x}=y$	✓
`pow(home, x, n, y);`	$x^n=y$	✓
`nroot(home, x, n, y);`	$\sqrt[n]{x}=y$	✓
`exp(home, x, y)`	$\exp(x)=y$	
`pow(home, b, x, y)`	$b^x=y$	
`log(home, x, y)`	$\log(x)=y$	
`log(home, b, x, y)`	$\log_b(x)=y$	
`sin(home, x, y)`	$\sin(x)=y$	
`cos(home, x, y)`	$\cos(x)=y$	
`tan(home, x, y)`	$\tan(x)=y$	
`asin(home, x, y)`	$\arcsin(x)=y$	
`acos(home, x, y)`	$\arccos(x)=y$	
`atan(home, x, y)`	$\arctan(x)=y$	

Figure 6.3: Arithmetic constraints (`x`, `y`, and `z` are float variables; `n` is a non-negative integer; `b` is a float number)

6.3.3 Arithmetic constraints

In addition to the constraints summarized in Figure 6.3 (see also Arithmetic constraints), the minimum and maximum constraints are also available for float variable arrays. That is, for a float variable array `x` and a float variable `y`

```
min(home, x, y);
```

constrains `y` to be the minimum of the variables in `x` (`max` is analogous).

The constraints marked as default in Figure 6.3 are always supported, the others only if Gecode has been built accordingly, see Tip 6.1.

6.3.4 Linear constraints

Linear constraints over float variables provide constraint post functions for linear constraints over float variables. The most general variant

```
linear(home, a, x, FRT_EQ, c);
```

posts the linear constraint

$$\sum_{i=0}^{|x|-1} a_i \cdot x_i = c$$

with float value coefficients `a` (of type `FloatValArgs`), float variables `x`, and a float value `c`. Note that `a` and `x` must have the same size. Of course, all other float relation types are supported, see Figure 6.2 for a table of float relation types (note that, linear constraints also show poor propagation for strict inequalities and disequality as discussed in Tip 6.4). Multiple occurrences of the same variable in `x` are explicitly allowed and common terms $a \cdot y$ and $b \cdot y$ for the same variable y are rewritten to $(a+b) \cdot y$ to increase propagation.

The array of coefficients can be omitted if all coefficients are one. That is,

```
linear(home, x, FRT_GR, c);
```

posts the linear constraint

$$\sum_{i=0}^{|x|-1} x_i > c$$

for a variable array `x` and a float value `c`.

Instead of a float value `c` as the right-hand side of the linear constraint, a float variable can be used as well. All variants of `linear` support reification.

6.3.5 Channel constraints

Channel constraints channel float variables to integer variables. To express that a float variable `x` is equal to an integer variable `y` is by posting either

```
channel(home, x, y);
```

or

```
channel(home, y, x);
```

6.4 Synchronized execution

Gecode offers support in Synchronized execution for executing a function (either a function or a static member function but not a member function) when float variables become assigned.

Assume that a function `c` (for continuation) is defined by

```
void c(Space& home) { ··· }
```

Then

```
wait(home, x, &c);
```

posts a propagator that waits until the float variable `x` (or, if `x` is an array of float variables: all variables in `x`) is assigned. If `x` becomes assigned, the function `c` is executed with the current home space passed as argument.

7 Modeling convenience: MiniModel

This chapter provides an overview of modeling convenience implemented by MiniModel. MiniModel (see Direct modeling support) provides some little helpers to the constraint modeler. However, it does not offer any new constraints or branchers.

Overview. Section 7.1 surveys how constraints represented by integer, Boolean, set, and float expressions and relations can be posted. How matrix interfaces for arrays can be defined and used is discussed in Section 7.2. Support for defining cost functions for cost-based optimization is presented in Section 7.3. Regular expressions for expressing extensional constraints are discussed in Section 7.4. Section 7.5 surveys channeling functions, whereas Section 7.6 and Section 7.7 discuss aliases for some commonly used constraints.

Important. Do not forget to add

```
#include <gecode/minimodel.hh>
```

to your program when you want to use MiniModel. Note that the same conventions hold as in Chapter 4.

7.1 Expressions and relations

The main part of MiniModel consists of overloaded operators and functions that provide a more natural syntax for posting constraints. These operators can be used in two slightly different ways. You can post a relation, or create a new variable from an expression.

For example, the following code creates a fresh integer variable `z` that is constrained to be equal to the given *expression* `3*x-4*y+2`, where both `x` and `y` are integer variables:

```
IntVar z=expr(home, 3*x-4*y+2);
```

An important aspect of posting an expression is that the returned variable is initialized with a reasonably small variable domain, see Tip 4.3.

A *relation* can be posted using the `rel` function, which posts the corresponding constraints and consequently returns **`void`**. Assume that `z` is an integer variable, then

```
rel(home, z == 3*x-4*y+2);
```

posts the same constraint as in the previous example.

MiniModel provides syntax for expressions and relations over integer, Boolean, set, and float variables, which can be freely mixed. For example, the following code snippet returns a Boolean variable that is true if and only if $\{x\} \subseteq s$ and $|s| = y$, where `x` and `y` are integer variables, and `s` is a set variable:

```
BoolVar b = expr(home, (singleton(x) <= s) && (cardinality(s) == y));
```

The rest of this section presents the different ways to construct expressions and relations, grouped by the type of the expressions.

7.1.1 Integer expressions and relations

Integer expressions (that is, expressions that evaluate to an integer) are constructed according to the structure sketched in Figure 7.1. We use the standard C++ operators (for an example, see Section 3.1), as well as several functions with intuitive names such as `min` or `max`. Integer expressions and relations can be constructed over integer, Boolean, and set variables. In Gecode, integer expressions are of type `LinIntExpr`, which are constructed using Linear expressions and relations, Arithmetic functions, and some Set expressions and relations.

Even arrays of variables (possibly with integer argument arrays as coefficients) can be used for posting some expressions and relations. For example, if `x` and `y` are integer variables and `z` is an array of integer variables, then

```
rel(home, x+2*sum(z) < 4*y);
```

posts a single linear constraint that involves all variables from the array `z`.

As long as an expression is *linear* (i.e., it can be represented as $\sum a_i \cdot x_i$ where the a_i are integers and x_i are integer or Boolean variables), the constraint posted for the expression will be as few `linear` constraints as possible (see Section 4.4.6) to ensure maximal constraint propagation.[1]

Non-linear expressions, such as a multiplication of two variables, are handled by MiniModel using *decomposition*. For example, posting the constraint

```
rel(home, a+b*(c+d) == 0);
```

for integer variables `a`, `b`, `c`, and `d` is equivalent to the decomposition

```
IntVar tmp0 = expr(home, c+d);
IntVar tmp1 = expr(home, b*tmp0);
rel(home, a+tmp1 == 0);
```

[1] In case a linear expression has only integer variables or only Boolean variables, a single `linear` constraint is posted. If the expression contains both integer and Boolean variables, two `linear` constraints are posted.

⟨*IntExpr*⟩	::=	⟨n⟩	integer value
	|	⟨x⟩	integer or Boolean variable
	|	`-` ⟨*IntExpr*⟩	unary minus
	|	⟨*IntExpr*⟩ `+` ⟨*IntExpr*⟩	addition
	|	⟨*IntExpr*⟩ `-` ⟨*IntExpr*⟩	subtraction
	|	⟨*IntExpr*⟩ `*` ⟨*IntExpr*⟩	multiplication
	|	⟨*IntExpr*⟩ `/` ⟨*IntExpr*⟩	integer division
	|	⟨*IntExpr*⟩ `%` ⟨*IntExpr*⟩	modulo
	|	`sum(`⟨$\overline{x}$⟩`)`	sum of integer or Boolean variables
	|	`sum(`⟨$\overline{n}$⟩`,`⟨$\overline{x}$⟩`)`	sum of integer or Boolean variables with integer coefficients
	|	`min(`⟨*IntExpr*⟩`,` ⟨*IntExpr*⟩`)`	minimum
	|	`min(`⟨$\overline{x}$⟩`)`	minimum of integer variables
	|	`max(`⟨*IntExpr*⟩`,` ⟨*IntExpr*⟩`)`	maximum
	|	`max(`⟨$\overline{x}$⟩`)`	maximum of integer variables
	|	`abs(`⟨*IntExpr*⟩`)`	absolute value
	|	`sqr(`⟨*IntExpr*⟩`)`	square
	|	`sqrt(`⟨*IntExpr*⟩`)`	square root
	|	`pow(`⟨*IntExpr*⟩`,` ⟨n⟩`)`	power
	|	`nroot(`⟨*IntExpr*⟩`,` ⟨n⟩`)`	n-th root
	|	`element(`⟨$\overline{x}$⟩`,` ⟨*IntExpr*⟩`)`	array element of integer variables
	|	`element(`⟨$\overline{n}$⟩`,` ⟨*IntExpr*⟩`)`	array element of integers
	|	`ite(`⟨*BoolExpr*⟩`,` ⟨*IntExpr*⟩`,` ⟨*IntExpr*⟩`)`	if-then-else
	|	`min(`⟨*SetExpr*⟩`)`	minimum of a set expression
	|	`max(`⟨*SetExpr*⟩`)`	maximum of a set expression
	|	`cardinality(`⟨*SetExpr*⟩`)`	cardinality of a set expression
⟨*IntRel*⟩	::=	⟨*IntExpr*⟩ ⟨r⟩ ⟨*IntExpr*⟩	relation
⟨r⟩	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`	relation symbol
⟨$\overline{x}$⟩	::=	array of integer or Boolean variables	
⟨$\overline{n}$⟩	::=	array of integers	

Figure 7.1: Integer expressions and relations

⟨*BoolExpr*⟩	::=	⟨*x*⟩	Boolean variable
	|	!⟨*BoolExpr*⟩	negation
	|	⟨*BoolExpr*⟩ && ⟨*BoolExpr*⟩	conjunction
	|	⟨*BoolExpr*⟩ || ⟨*BoolExpr*⟩	disjunction
	|	⟨*BoolExpr*⟩ == ⟨*BoolExpr*⟩	equivalence
	|	⟨*BoolExpr*⟩ != ⟨*BoolExpr*⟩	non-equivalence
	|	⟨*BoolExpr*⟩ >> ⟨*BoolExpr*⟩	implication
	|	⟨*BoolExpr*⟩ << ⟨*BoolExpr*⟩	reverse implication
	|	⟨*BoolExpr*⟩ ^ ⟨*BoolExpr*⟩	exclusive or
	|	element(⟨$\overline{x}$⟩, ⟨*IntExpr*⟩)	array element of Boolean variables
	|	⟨*IntRel*⟩	reified integer relation
	|	⟨*SetRel*⟩	reified set relation
	|	⟨*FloatRel*⟩	reified float relation

Figure 7.2: Boolean expressions

Like the post functions for integer and Boolean constraints presented in Section 4.4, posting integer expressions and relations supports an optional argument of type `IntConLevel` to select the consistency level. For more information, see Posting of expressions and relations and Section 4.3.

Using the `expr` function, you can enforce a particular decomposition, and you can specify the consistency level for each subexpression. For example,

```
rel(home, x+expr(home,y*z,ICL_DOM) == 0);
```

will propagate domain consistency for the multiplication, but bounds consistency (the default) for the sum.

An `element` expression such as `element(x,e)`, where `x` is an array of integers or integer variables, and `e` is an integer expression, corresponds to an array access `x[e]`, implemented using an `element` constraint (see Section 4.4.12).

MiniModel provides three integer expressions whose arguments are set expressions: the minimum of a set, the maximum of a set, and a set's cardinality. We will see later how set expressions are constructed.

For examples of integer expressions, see Alpha puzzle, SEND+MORE=MONEY puzzle, Grocery puzzle, Chapter 12, Chapter 15, and Section 3.1.

7.1.2 Boolean expressions and relations

Boolean expressions are constructed using standard C++ operators according to the structure sketched in Figure 7.2.

Again, the purpose of a Boolean expression or relation is to post a corresponding constraint for it (see Posting of expressions and relations). Posting a Boolean expression returns a new Boolean variable that is constrained to the value of the expression. Several constraints

might be posted for a single expression, however as few constraints as possible are posted. For example, all negation constraints are eliminated by rewriting the Boolean expression into NNF (negation normal form) and conjunction and disjunction constraints are combined whenever possible.

For example, the Boolean expression `x && (y >> z)` (to be read as $x \wedge (y \rightarrow z)$) for Boolean variables `x`, `y`, and `z` is posted by

```
BoolVar b=expr(home, x && (y >> z));
```

Tip 7.1 (Boolean precedences)**.** Note that the precedences of the Boolean connectives are different from the usual mathematical notation. In C++, operator precedence cannot be changed, so the precedences are as follows (high to low): `!`, `<<`, `>>`, `==`, `!=`, `^`, `&&`, `||`. For instance, this means that the expression $b_0 == b_1 >> b_2$ will be interpreted as $(b_0 \leftrightarrow b_1) \rightarrow b_2$ instead of the more canonical $b_0 \leftrightarrow (b_1 \rightarrow b_2)$. If in doubt, use parentheses! ◀

Any Boolean expression e corresponds to the Boolean relation stating that e is true. Posting a Boolean relation posts the corresponding Boolean constraint. Using the Boolean expression from above,

```
rel(home, x && (y >> z));
```

posts that $x \wedge (y \rightarrow z)$ must be true, whereas

```
rel(home, !(x && (y >> z)));
```

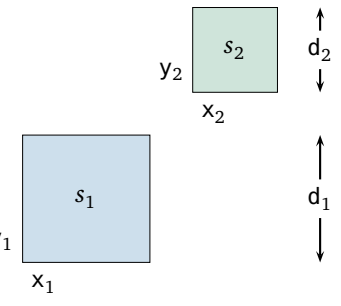
posts that $x \wedge (y \rightarrow z)$ must be false.

A Boolean `element` expression such as `element(x,e)`, where `x` is an array of Boolean variables, and `e` is an integer expression, corresponds to an array access `x[e]`, implemented using an `element` constraint (see Section 4.4.12).

Boolean expressions include reified integer relations. As an example consider the placement of two squares s_1 and s_2 such that the squares do not overlap. A well known model for this constraint is

$$\begin{array}{lll} x_1 + d_1 \leq x_2 & \vee & x_2 + d_2 \leq x_1 \quad \vee \\ y_1 + d_1 \leq y_2 & \vee & y_2 + d_2 \leq y_1 \end{array}$$

The meaning of the integer variables x_i and y_i, and the integer values d_i is sketched to the right. The squares do not overlap, if the relative position of s_1 with respect to s_2 is either left, right, above, or below. As soon as one of the relationships is established, the squares do not overlap. Please also consult Section 4.4.16 for geometrical packing constraints.



With Boolean relations using reified integer relations, the constraint that the squares s_1 and s_2 do not overlap can be posted as follows:

```
rel(home, (x1+d1 <= x2) || (x2+d2 <= x1) ||
          (y1+d1 <= y2) || (y2+d2 <= y1));
```

Like the post functions for integer and Boolean variables presented above, posting Boolean expressions and relations supports an optional argument of type `IntConLevel` to select the consistency level. For more information, see Section 4.3.

Boolean expressions also include reified set relations, which will be covered below.

Tip 7.2 (Reification of non-functional constraints)**.** Reification of integer or set relations is mostly implemented through *decomposition*. For example, given integer variables `x`, `y`, and `z`, the reified division constraint

```
rel(home, (x / y == z) == b);
```

is actually equivalent to

```
IntVar tmp = expr(home, x / y);
rel(home, (tmp == z) == b);
```

Some constraints, such as division above, are not simple functions but impose side constraints. In the case of the division above, the side constraint is that `y` is not zero. It is important to understand the subtle semantics of decomposed reification here: If `y` happens to be zero, we get failure instead of `b` being constrained to false!

There are several expressions that have non-functional semantics: division, modulo, element, and disjoint set union (introduced below). ◀

For more examples using Boolean expressions and Boolean relations including reification, see Chapter 14.

7.1.3 Set expressions and relations

Set expressions and relations are constructed using the standard C++ operators and the functions listed in Figure 7.3. Just like for integer and Boolean expressions, posting of a set expression returns a new set variable that is constrained to the value of the expression.

For example, the set expression `x & (y | z)` (to be read as $x \cap (y \cup z)$) for set variables `x`, `y`, and `z` is posted by

```
SetVar s = expr(home, x & (y | z));
```

Posting a set relation posts the corresponding constraint. Given an existing set variable `s`, the previous code fragment could therefore be written as

```
rel(home, s == (x & (y | z)));
```

As noted above, set relations can be reified, turning them into Boolen expressions. The following code posts the constraint that `b` is true if and only if `x` is the complement of `y`:

```
BoolVar b = expr(home, (x == -y));
```

⟨*SetExpr*⟩	::=	⟨x⟩	set variable
	|	⟨s⟩	set constant (`IntSet`)
	|	- ⟨*SetExpr*⟩	complement
	|	⟨*SetExpr*⟩ & ⟨*SetExpr*⟩	intersection
	|	⟨*SetExpr*⟩ | ⟨*SetExpr*⟩	union
	|	⟨*SetExpr*⟩ + ⟨*SetExpr*⟩	disjoint union
	|	⟨*SetExpr*⟩ - ⟨*SetExpr*⟩	set difference
	|	`inter`(⟨$\overline{x}$⟩)	intersection of variables
	|	`setunion`(⟨$\overline{x}$⟩)	union of variables
	|	`setdunion`(⟨$\overline{x}$⟩)	disjoint union of variables
	|	`singleton`(⟨*IntExpr*⟩)	singleton given by integer expression
⟨*SetRel*⟩	::=	⟨*SetExpr*⟩ == ⟨*SetExpr*⟩	expressions are equal
	|	⟨*SetExpr*⟩ != ⟨*SetExpr*⟩	expressions are not equal
	|	⟨*SetExpr*⟩ <= ⟨*SetExpr*⟩	first is subset of second expression
	|	⟨*SetExpr*⟩ >= ⟨*SetExpr*⟩	first is superset of second expression
	|	⟨*SetExpr*⟩ || ⟨*SetExpr*⟩	expressions are disjoint
⟨$\overline{x}$⟩	::=	array of set variables	

Figure 7.3: Set expressions and relations

Intead of a set variable, you can always use a constant `IntSet`, for example for reifying the fact that `x` is empty:

```
BoolVar b = expr(home, (x == IntSet::empty));
```

The subset relations can also be posted two-sided, such as

```
rel(home, IntSet(0,10) <= x <= IntSet(0,20));
```

7.1.4 Float expressions and relations

Linear float expressions and relations are constructed using the standard C++ operators and the functions listed in Figure 7.4 (see also Arithmetic functions, Transcendental functions, and Trigonometric functions). Posting a float expression returns a new float variable that is constrained to the value of the expression.

Instead of a float variable, you can always use a constant of type `FloatVal`.

[2]These functions are only available if Gecode has been compiled with support for MPFR, see Tip 6.1.

⟨*FloatExpr*⟩	::=	⟨x⟩	float variable
	|	⟨f⟩	float value
	|	`-`⟨*FloatExpr*⟩	unary minus
	|	⟨*FloatExpr*⟩ `+` ⟨*FloatExpr*⟩	addition
	|	⟨*FloatExpr*⟩ `-` ⟨*FloatExpr*⟩	subtraction
	|	⟨*FloatExpr*⟩ $*$ ⟨*FloatExpr*⟩	multiplication
	|	⟨*FloatExpr*⟩ `/` ⟨*FloatExpr*⟩	division
	|	`sum(`⟨$\overline{x}$⟩`)`	sum of float variables
	|	`sum(`⟨$\overline{f}$⟩`,`⟨$\overline{x}$⟩`)`	sum of float with coefficients
	|	`min(`⟨*FloatExpr*⟩`,`⟨*FloatExpr*⟩`)`	minimum
	|	`min(`⟨$\overline{x}$⟩`)`	minimum of float variables
	|	`max(`⟨*FloatExpr*⟩`,`⟨*FloatExpr*⟩`)`	maximum
	|	`max(`⟨$\overline{x}$⟩`)`	maximum of float variables
	|	`abs(`⟨*FloatExpr*⟩`)`	absolute value
	|	`sqr(`⟨*FloatExpr*⟩`)`	square
	|	`sqrt(`⟨*FloatExpr*⟩`)`	square root
	|	`pow(`⟨*FloatExpr*⟩`,`⟨n⟩`)`	power
	|	`nroot(`⟨*FloatExpr*⟩`,`⟨n⟩`)`	n-th root
	|	`exp(`⟨*FloatExpr*⟩`)`	exponential[2]
	|	`log(`⟨*FloatExpr*⟩`)`	logarithm[2]
	|	`sin(`⟨*FloatExpr*⟩`)`	sine[2]
	|	`cos(`⟨*FloatExpr*⟩`)`	cosine[2]
	|	`tan(`⟨*FloatExpr*⟩`)`	tangent[2]
	|	`asin(`⟨*FloatExpr*⟩`)`	arcsine[2]
	|	`acos(`⟨*FloatExpr*⟩`)`	arccosine[2]
	|	`atan(`⟨*FloatExpr*⟩`)`	arctangent[2]
⟨*FloatRel*⟩	::=	⟨*FloatExpr*⟩ ⟨r⟩ ⟨*FloatExpr*⟩	relation
⟨r⟩	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`	relation symbol
⟨$\overline{x}$⟩	::=	array of float variables	
⟨$\overline{f}$⟩	::=	array of float values	

Figure 7.4: Float expressions and relations

7.2 Matrix interface for arrays

MiniModel provides a `Matrix` support class for accessing an array as a two dimensional matrix. The following

```
IntVarArgs x(n*m);
...
Matrix<IntVarArgs> mat(x, n, m);
```

declares an array of integer variables `x` and superimposes a matrix interface to `x` called `mat` with width `n` and height `m`. Note that the first argument specifies the number of columns, and the second argument specifies the number of rows.

The elements of the array can now be accessed at positions $\langle i, j \rangle$ in the matrix `mat` (that is, the element in column i and row j) using

```
IntVar mij = mat(i,j);
```

Furthermore, the rows and columns of the matrix can be accessed using `mat.row(i)` and `mat.col(j)`. If a rectangular slice is required, the `slice()` member function can be used.

A matrix interface can be declared for any standard array or argument array used in Gecode, such as `IntVarArray` or `IntSetArgs`.

As an example of how the `Matrix` class can be used, consider the Sudoku problem (see Solving Sudoku puzzles using both set and integer). Given that there is a member `IntVarArray x` that contains $9 \cdot 9$ integer variables with domain $\{1,\ldots,9\}$, the following code posts constraints that implement the basic rules for a Sudoku.

```
Matrix<IntVarArray> m(x, 9, 9);

for (int i=0; i<9; i++)
  distinct(home, m.row(i));
for (int i=0; i<9; i++)
  distinct(home, m.col(i));
for (int i=0; i<9; i+=3)
  for (int j=0; j<9; j+=3)
    distinct(home, m.slice(i, i+3, j, j+3));
```

For more examples that use the `Matrix` class, see Chapter 21, Chapter 17, Chapter 20, Chapter 16, Magic squares, and Nonogram.

Element constraints. A matrix can also be used with an element constraint that propagates information about the row and column of matrix entries.

For example, the following code assumes that x is an integer array of type IntArgs with 12 elements.

```
Matrix<IntArgs> m(x, 3, 4);
IntVar r(home,0,1024), c(home,0,1024), v(home,0,1024);
element(home, m, r, c, v);
```

constrains the variable v to the value at position $\langle r, c \rangle$ of the matrix m (see also GCCAT: element_matrix).

Tip 7.3 (Element for matrix can compromise propagation)**.** Whenever it is possible one should use an array rather than a matrix for posting element constraints, as an element constraint for a matrix will provide rather weak propagation for the row and column variables.

Consider the following array of integers x together with its matrix interface m

```
IntArgs x(4, 0,2,2,1);
Matrix<IntArgs> m(x,2,2);
```

That is, m represents the matrix

$$\begin{pmatrix} 0 & 2 \\ 2 & 1 \end{pmatrix}$$

Consider the following example using an element constraint on an integer array:

```
IntVar i(home,0,8), v(home,0,1);
element(home, x, i, v);
```

After performing propagation, i will be constrained to the set $\{0, 3\}$ (as 2 is not included in the values of v).

Compare this to propagating an element constraint over the corresponding matrix as follows:

```
IntVar r(home,0,8), c(home,0,8), v(home,0,1);
element(home, m, r, c, v);
```

Propagation of element will determine that only the fields $\langle 0, 0 \rangle$ and $\langle 1, 1 \rangle$ are still possible. But propagating this information to the row and column variables, yields the values $\{0, 1\}$ for both r and c: each value for the coordinates is still possible even though some of their combinations are not. ◀

7.3 Support for cost-based optimization

Support for cost-based optimization provides subclasses of Space, IntMinimizeSpace and IntMaximizeSpace to search for a solution of minimal and maximal, respectively, integer cost.

operation	meaning
`REG r`	initialize `r` as ϵ (empty)
`REG r(4)`	initialize `r` as single integer (symbol) `4`
`REG r(IntArgs(3, 0,2,4))`	initialize `r` as alternative of integers `0`|`2`|`4`
`r + s`	`r` followed by `s`
`r | s`	`r` or `s`
`r += s`	efficient shortcut for `r = r + s`
`r |= s`	efficient shortcut for `r = r | s`
`*r`	repeat `r` arbitrarily often (Kleene star)
`+r`	repeat `r` at least once
`r(n)`	repeat `r` at least `n` times
`r(n,m)`	repeat `r` at least `n` times, at most `m` times

Figure 7.5: Constructing regular expressions (`r` and `s` are regular expressions, `n` and `m` are unsigned integers)

In order to use these abstract classes, a class inheriting from them must implement a virtual cost function of type

```
virtual IntVar cost(void) const { ··· }
```

The function must return an integer variable for the cost. For an example, see Section 3.2.

Tip 7.4 (Cost must be assigned for solutions)**.** In case the `cost()` function is called on a *solution*, the variable returned by `cost()` *must* be assigned. If the variable is unassigned for a solution, an exception of type `Int::ValOfUnassignedVar` is thrown. ◀

Support for cost-based optimization also provides subclasses `FloatMinimizeSpace` and `FloatMaximizeSpace` for searching a solution of minimal and maximal, respectively, float cost.

7.4 Regular expressions for extensional constraints

Regular expressions are implemented as instances of the class `REG` and provide an alternative, typically more convenient, interface for the specification of extensional constraints than DFAs do. The construction of regular expressions is summarized in Figure 7.5.

Let us reconsider the Swedish drinking protocol from Section 4.4.13. The protocol can be described by a regular expression `r` constructed by

```
REG r = *REG(0) + *(REG(1) + +REG(0));
```

A sequence of activities `x` (an integer or Boolean variable array) can be constrained by

```
DFA d(r);
extensional(home, x, d);
```

after a DFA for the regular expression has been computed.

Tip 7.5 (Creating a DFA only once)**.** Please make it a habit to create a DFA explicitly from a regular expression `r` rather than implicitly by

```
extensional(home, x, r);
```

Both variants work, however the implicit variant disguises the fact that each time the code fragment is executed, a new DFA for the regular expression `r` is computed (think about the code fragment being executed inside a loop and your C++ compiler being not too smart about it)![3] ◀

For examples on using regular expressions for extensional constraints, see the nonogram case study in Chapter 16 or the examples Solitaire domino, Nonogram, and Pentominoes. The models are based on ideas described in [22], where regular expressions for extensional constraints nicely demonstrate their usefulness.

7.5 Channeling functions

Channel functions are functions to channel a Boolean variable to an integer variable and vice versa, to channel a float variable to an integer variable, and to channel between integer variables and a set variable.

For an integer variable `x`,

```
channel(home, x);
```

returns a new Boolean variable that is equal to `x`. Likewise, for a Boolean variable `x` an equal integer variable is returned.

For a float variable `x`, `channel(home, x)` returns an integer variable equal to `x`.

For an array of integer variables `x`, `channel(home, x)` returns a set variable equal to all the integers in `x`.

7.6 Aliases for integer constraints

Aliases for integer constraints provide some popular aliases. Figure 7.6 lists the aliases and their corresponding definitions.

7.7 Aliases for set constraints

Aliases for set constraints provide aliases and convenience post functions for useful set constraints.

[3] The integer module cannot know anything about regular expressions. Hence, it is impossible in C++ to avoid the implicit conversion. This is due to the fact that the conversion is controlled by a type operator (that must reside in the MiniModel module) and not by a constructor that could be made **`explicit`**.

alias	constraint posted	GCCAT
`atmost(home, x, u, v);`	`count(home, x, u, IRT_LQ, v);`	`atmost`
`atleast(home, x, u, v);`	`count(home, x, u, IRT_GQ, v);`	`atleast`
`exactly(home, x, u, v);`	`count(home, x, u, IRT_EQ, v);`	`exactly`
`lex(home, x, r, y);`	`rel(home, x, r, y);`	`lex`
`values(home, x, s);`	`dom(home, x, s);` `nvalues(home, x, IRT_EQ, s.size());`	

Figure 7.6: Aliases for integer constraints (`x` and `y` are integer variable arrays, `u` and `v` are integers or integer variables, `r` is an integer relation type, `s` is an integer set)

`channel(home, x, y)` is an alias for `rel(home, SOT_UNION, x, y)`, posting the constraint that `y` is exactly the set of integers $\{x_0, \ldots, x_{|x|-1}\}$. In addition to the union constraint, it posts an `nvalues` constraint for stronger propagation (see Section 4.4.9).

`range(home, x, y, z)`, where `x` is an array of integer variables and `y` and `z` are set variables, is an alias for `element(home, SOT_UNION, x, y, z)`. This constraints treats `x` as defining a function, and constrains `z` to be the range of the function restricted to `y`:

$$z = \bigcup_{i \in y} \{x_i\}$$

Conversely, `roots(home, x, y, z)` constrains `y` to be the roots of the elements in `z`, i.e., those indices mapping to elements in `z`:

$$y = \bigcup_{i \in z} \{j \mid x_j = i\}$$

(see also GCCAT: `roots`).

8 Branching

This chapter discusses how *branching* is used for solving Gecode models. Branching defines the shape of the search tree. Exploration defines a strategy how to explore parts of the search tree and is discussed in Chapter 9.

Overview. Section 8.1 explains the basics of Gecode's predefined branchings. An overview of available branchings for integer and Boolean variables is provided in Section 8.2, for set variables in Section 8.3, and for float variables in Section 8.4. These sections belong to the basic reading material of Part M.

Advanced topics for branchings are discussed in the remaining sections: local versus shared variable selection (Section 8.5), random selection (Section 8.6), user-defined variable (Section 8.7) and value (Section 8.8) selection, tie-breaking (Section 8.9), dynamic symmetry breaking (Section 8.10), branch filter functions (Section 8.11), variable-value print functions (Section 8.12), assigning variables (Section 8.13), and executing code between branchers (Section 8.14).

Convention. Note that the same conventions hold as in Chapter 4.

8.1 Branching basics

Gecode offers predefined *variable-value branching*: when calling `branch()` to post a branching, the third argument defines which variable is selected for branching, whereas the fourth argument defines which values are selected for branching.

For example, for an array of integer or Boolean variables `x` the following call to branch

```
branch(home, x, INT_VAR_MIN_MIN(), INT_VAL_SPLIT_MIN());
```

selects a variable y with the smallest minimum value (in case of ties, the first such variable in `x` is selected) and creates a choice with two alternatives $y \leq n$ and $y > n$ where

$$n = \left\lfloor \frac{\min(y) + \max(y)}{2} \right\rfloor$$

The posted brancher assigns all variables and then ceases to exist. If more branchers exist, search continues with the next brancher.

The `branch()` function also accepts a branch filter function and a variable-value print function as optional arguments, see Section 8.11 and Section 8.12 for details.

Several branchers. A space in Gecode can have *several* branchers posted on behalf of a *branching* that are executed in order of creation. Assume that in

```
branch(home, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
...
branch(home, y, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

both calls to `branch()` create a brancher. Search branches first on the variables `x` and then on the variables `y`. Here, it does not matter whether propagators are created in between the creation of branchers.

Branching on single variables. In addition to branching on an array of variables, Gecode also supports branching on a single variable.

For example, if `x` is an integer variable of type `IntVar`, then

```
branch(home, x, INT_VAL_MIN());
```

branches on the single variable `x` by first trying the smallest value of `x`.

Assume that `x` is an array of integer variables. Then the following code

```
for (int i=0; i<x.size(); i++)
  branch(home, x[i], INT_VAL_MIN());
```

is equivalent, albeit considerably less efficient, to

```
branch(home, x, INT_VAR_NONE(), INT_VAL_MIN());
```

Brancher handles. When creating a brancher, the call to `branch()` returns a brancher handle of class `BrancherHandle`. For example, by

```
BrancherHandle b = branch(home, x, INT_VAR_NONE(), INT_VAL_MIN());
```

a new brancher is created and a handle `b` to it is returned.

A brancher handle offers only very few member functions. The most important ones are:

- `b(home)` checks whether the brancher referred to by `b` is still active.
- `b.kill(home)` deletes the brancher. This is the most useful operation provided by a brancher handle: one can create a brancher for one stage of search and delete it later when it is not any longer useful.

Note that in case the home space is failed, the call to `branch()` might return a brancher handle `b` that does in fact not refer to any brancher. In this case `b(home)` returns **`false`** and `b.kill(home)` does nothing.

`INT_VAR_NONE()`	first unassigned
`INT_VAR_RND(r)`	randomly
`INT_VAR_MERIT_MIN(m,t*)`	smallest value of merit function `m`
`INT_VAR_MERIT_MAX(m,t*)`	largest value of merit function `m`
`INT_VAR_DEGREE_MIN(t*)`	smallest degree
`INT_VAR_DEGREE_MAX(t*)`	largest degree
`INT_VAR_AFC_MIN(afc+,t*)`	smallest accumulated failure count (AFC)
`INT_VAR_AFC_MAX(afc+,t*)`	largest accumulated failure count (AFC)
`INT_VAR_ACTIVITY_MIN(act+,t*)`	lowest activity
`INT_VAR_ACTIVITY_MAX(act+,t*)`	highest activity
`INT_VAR_MIN_MIN(t*)`	smallest minimum value
`INT_VAR_MIN_MAX(t*)`	largest minimum value
`INT_VAR_MAX_MIN(t*)`	smallest maximum value
`INT_VAR_MAX_MAX(t*)`	largest maximum value
`INT_VAR_SIZE_MIN(t*)`	smallest domain size
`INT_VAR_SIZE_MAX(t*)`	largest domain size
`INT_VAR_DEGREE_SIZE_MIN(t*)`	smallest degree divided by domain size
`INT_VAR_DEGREE_SIZE_MAX(t*)`	largest degree by domain size
`INT_VAR_AFC_SIZE_MIN(afc+,t*)`	smallest AFC by domain size
`INT_VAR_AFC_SIZE_MAX(afc+,t*)`	largest AFC by domain size
`INT_VAR_ACTIVITY_SIZE_MIN(act+,t*)`	smallest activity by domain size
`INT_VAR_ACTIVITY_SIZE_MAX(act+,t*)`	largest activity by domain size
`INT_VAR_REGRET_MIN_MIN(t*)`	smallest minimum-regret
`INT_VAR_REGRET_MIN_MAX(t*)`	largest minimum-regret
`INT_VAR_REGRET_MAX_MIN(t*)`	smallest maximum-regret
`INT_VAR_REGRET_MAX_MAX(t*)`	largest maximum-regret

Figure 8.1: Integer and Boolean variable selection

8.2 Branching on integer and Boolean variables

Important. Do not forget to add

```
#include <gecode/int.hh>
```

to your program when you want to branch on integer and Boolean variables.

For integer and Boolean variables, variable selection is defined by a value of class `IntVarBranch` and value selection is defined by a value of type `IntValBranch`. Values of these types are obtained by calling functions (possibly taking arguments) that correspond to variable and value selection strategies. For example, a call `INT_VAR_SIZE_MIN()` returns an object of class `IntVarBranch`.

For an overview of the available variable selection strategies, see Figure 8.1 (see also Variable selection for integer and Boolean variables) where $\cdot^*$ denotes an optional argument

`INT_VAL_RND(r)`	random value
`INT_VAL(v,c*)`	defined by value function `v` and commit function `c`
`INT_VAL_MIN()`	smallest value
`INT_VAL_MED()`	greatest value not greater than the median
`INT_VAL_MAX()`	largest value
`INT_VAL_SPLIT_MIN()`	values not greater than mean of smallest and largest value
`INT_VAL_SPLIT_MAX()`	values greater than mean of smallest and largest value
`INT_VAL_RANGE_MIN()`	values from smallest range, if domain has several ranges; otherwise, values not greater than mean
`INT_VAL_RANGE_MAX()`	values from largest range, if domain has several ranges; otherwise, values greater than mean
`INT_VALUES_MIN()`	all values starting from smallest
`INT_VALUES_MAX()`	all values starting from largest
`INT_VAL_NEAR_MIN(n)`	values near to value in array `n` takes smaller value in case of ties
`INT_VAL_NEAR_MAX(n)`	values near to value in array `n` takes larger value in case of ties
`INT_VAL_NEAR_INC(n)`	values larger than value in array `n` first
`INT_VAL_NEAR_DEC(n)`	values smaller than value in array `n` first

Figure 8.2: Integer and Boolean value selection

and $\cdot^+$ is a special argument to be explained below. Here, an argument `r` refers to a random number generator of type `Rnd`. Using random number generators for branching is discussed in Section 8.6. An argument `m` refers to a user-defined merit function of type `IntBranchMerit` for integer variables and `BoolBranchMerit` for Boolean variables. User-defined merit functions are discussed in Section 8.7. An argument `afc` refers to accumulated failure count (AFC) information for integer or Boolean variables (of class `IntAFC`). An argument `act` refers to activity information for integer or Boolean variables (of class `IntActivity`). For a discussion of AFC and activity, see Section 8.5, both `afc`$^+$ and `act`$^+$ can also be optional arguments of type **`double`** defining a decay-factor. The optional argument `t` refers to a tie-breaking limit function of type `BranchTbl` and is discussed in Section 8.9.

An overview of the available value selection strategies for integer and Boolean variables can be found in Figure 8.2 (see also Value selection for integer and Boolean variables) where $\cdot^*$ denotes an optional argument. Here, an argument `r` refers to a random number generator of type `Rnd` which is discussed in Section 8.6. An argument `v` refers to a value selection function of type `IntBranchVal` for integer variables and `BoolBranchVal` for Boolean variables. An optional argument `c` refers to a commit function of type `IntBranchCommit` for integer variables and of type `BoolBranchCommit` for Boolean variables. Value and commit functions are discussed in Section 8.8. The argument `n` must be an array of integers (or a shared array of integers, see Tip 4.9) of the same size as the variable array used for branching.

Note that variable-value branchers are just common cases for branching based on the

idea of selecting variables and values. In Gecode also arbitrary other branchers can be programmed, see Part B.

Tip 8.1 (Variables are re-selected during branching)**.** A variable-value branching selects a variable for each choice it creates. Consider as an example a script using an integer variable array x with three variables and domains $[1 .. 4]$ created by

```
IntVarArray x(home, 3, 1, 4);
```

Let us assume that no constraints are posted on the variables in x and that a branching is posted by

```
branch(home, x, INT_VAR_SIZE_MAX(), INT_VAL_SPLIT_MIN());
```

The branching starts by selecting x[0] as the first variable with the largest domain in the array x and creates the choice

$$(\texttt{x[0]} \leq 2) \vee (\texttt{x[0]} > 2)$$

Now assume that search explores the first alternative which results in the domain $\{1,2\}$ for x[0]. When search continues, the branching again selects the first variable with a largest domain: hence x[1] is selected and *not* x[0].

In other words, a variable-value branching does not stick to a selected variable until the variable becomes assigned. Instead, a variable-value branching re-selects a variable for each choice it creates. ◀

Tip 8.2 (Do not try all values)**.** Note that for INT_VALUES_MIN() and INT_VALUES_MAX(), a variable-value branching creates a choice for each selected variable with one alternative per value of the variable.

This is typically a poor choice, as none of the alternatives can benefit from propagation that arises when other values of the same variable are tried. These branchings exist for instructional purposes (well, they do create beautiful trees in Gist). ◀

Selecting values near to a value. Assume that by search you have found a solution to your problem (say values for an array of integer variables x). What one might want to try is to find a solution *nearby* when for example restarting search (see also Section 9.4). For this purpose, Gecode offers the value selection strategies INT_VAL_NEAR_MIN(), INT_VAL_NEAR_MAX(), INT_VAL_NEAR_INC(), and INT_VAL_NEAR_DEC(). All of them expect an array of integers as an argument.

Let us create an array of integers c corresponding to the solution for the variables in x as follows:

```
IntArgs c(x.size());
for (int i=0; i<x.size(); i++)
  c[i] = x[i].val();
```

Now assume that we restart search and post the following branching by:

```
branch(home, x, INT_VAR_NONE(), INT_VAL_NEAR_MIN(c));
```

Then, during branching for a variable in `x` at position `i` the value that will be tried first is c_i (provided it has not already been pruned by propagation). Then all values close to c_i will be tried.

For example, assume that $c_i = 3$. Then for the following example domains of x_i, the values that are tried first (shown as domain → value) are:

$$\{0,1,2,3,4,5,6\} \to 3 \qquad \{0,1,2,4,5,6\} \to 2 \qquad \{0,1,4,5,6\} \to 4$$
$$\{0,1,5,6\} \to 1 \qquad \{0,5,6\} \to 5 \qquad \{0,6\} \to 0$$

That is, when two values are as near as possible, the smaller of the two values is selected. `INT_VAL_NEAR_MAX()` selects the larger value instead.

`INT_VAL_NEAR_INC(c)` first tries to choose values in increasing order that are at least as large as c_i. Then, it tries values smaller than c_i in decreasing order. For example, `INT_VAL_NEAR_INC(c)` for $c_i = 3$ tries the values for the following example domains of x_i:

$$\{0,1,2,3,4,5,6\} \to 3 \qquad \{0,1,2,4,5,6\} \to 4 \qquad \{0,1,2,5,6\} \to 5$$
$$\{0,1,2,6\} \to 6 \qquad \{0,1,2\} \to 2 \qquad \{0,1\} \to 1$$

That is, values are chosen starting from c_i in increasing order if possible and in decreasing order if not. `INT_VAL_NEAR_DEC(c)` selects the values in inverse order.

8.3 Branching on set variables

Important. Do not forget to add

```
#include <gecode/set.hh>
```

to your program when you want to branch on set variables.

For set variables, variable selection is defined by a value of class `SetVarBranch` (see also Selecting set variables) and value selection is defined by a value of type `SetValBranch` (see also Value selection for set variables).

For an overview of the available variable selection strategies, see Figure 8.3 (see also Selecting set variables) where $\cdot^*$ denotes an optional argument and $\cdot^+$ is a special argument to be explained below. Here, an argument `r` refers to a random number generator of type `Rnd`. Using random number generators for branching is discussed in Section 8.6. An argument `m` refers to a user-defined merit function of type `SetBranchMerit`. User-defined merit functions are discussed in Section 8.7. An argument `afc` refers to accumulated failure count (AFC) information for set variables (of class `SetAFC`). An argument `act` refers to activity information for set variables (of class `SetActivity`). For a discussion of AFC and activity, see Section 8.5, both $\texttt{afc}^+$ and $\texttt{act}^+$ can also be optional arguments of type **`double`** defining a decay-factor. The optional argument `t` refers to a tie-breaking limit function of type `BranchTbl` and is discussed in Section 8.9.

SET_VAR_NONE()	first unassigned
SET_VAR_RND(r)	randomly
SET_VAR_MERIT_MIN(m,t*)	smallest value of merit function m
SET_VAR_MERIT_MAX(m,t*)	largest value of merit function m
SET_VAR_DEGREE_MIN(t*)	smallest degree
SET_VAR_DEGREE_MAX(t*)	largest degree
SET_VAR_AFC_MIN(afc+,t*)	smallest accumulated failure count (AFC)
SET_VAR_AFC_MAX(afc+,t*)	largest accumulated failure count (AFC)
SET_VAR_ACTIVITY_MIN(act+,t*)	lowest activity
SET_VAR_ACTIVITY_MAX(act+,t*)	highest activity
SET_VAR_MIN_MIN(t*)	smallest minimum unknown element
SET_VAR_MIN_MAX(t*)	largest minimum unknown element
SET_VAR_MAX_MIN(t*)	smallest maximum unknown element
SET_VAR_MAX_MAX(t*)	largest maximum unknown element
SET_VAR_SIZE_MIN(t*)	smallest unknown set
SET_VAR_SIZE_MAX(t*)	largest unknown set
SET_VAR_DEGREE_SIZE_MIN(t*)	smallest degree divided by domain size
SET_VAR_DEGREE_SIZE_MAX(t*)	largest degree divided by domain size
SET_VAR_AFC_SIZE_MIN(afc+,t*)	smallest AFC divided by domain size
SET_VAR_AFC_SIZE_MAX(afc+,t*)	largest AFC divided by domain size
SET_VAR_ACTIVITY_SIZE_MIN(act+,t*)	smallest activity divided by domain size
SET_VAR_ACTIVITY_SIZE_MAX(act+,t*)	largest activity divided by domain size

Figure 8.3: Set variable selection

SET_VAL_RND_INC(r)	include random element
SET_VAL_RND_EXC(r)	exclude random element
SET_VAL(v,c*)	defined by value function v and commit function c
SET_VAL_MIN_INC()	include smallest element
SET_VAL_MIN_EXC()	exclude smallest element
SET_VAL_MED_INC()	include median element (rounding downwards)
SET_VAL_MED_EXC()	exclude median element (rounding downwards)
SET_VAL_MAX_INC()	include largest element
SET_VAL_MAX_EXC()	exclude largest element

Figure 8.4: Set value selection

`FLOAT_VAR_NONE()`	first unassigned
`FLOAT_VAR_RND(r)`	randomly
`FLOAT_VAR_MERIT_MIN(m,t*)`	smallest value of merit function `m`
`FLOAT_VAR_MERIT_MAX(m,t*)`	largest value of merit function `m`
`FLOAT_VAR_DEGREE_MIN(t*)`	smallest degree
`FLOAT_VAR_DEGREE_MAX(t*)`	largest degree
`FLOAT_VAR_AFC_MIN(afc+,t*)`	smallest accumulated failure count (AFC)
`FLOAT_VAR_AFC_MAX(afc+,t*)`	largest accumulated failure count (AFC)
`FLOAT_VAR_ACTIVITY_MIN(act+,t*)`	lowest activity
`FLOAT_VAR_ACTIVITY_MAX(act+,t*)`	highest activity
`FLOAT_VAR_MIN_MIN(t*)`	smallest minimum value
`FLOAT_VAR_MIN_MAX(t*)`	largest minimum value
`FLOAT_VAR_MAX_MIN(t*)`	smallest maximum value
`FLOAT_VAR_MAX_MAX(t*)`	largest maximum value
`FLOAT_VAR_SIZE_MIN(t*)`	smallest domain size
`FLOAT_VAR_SIZE_MAX(t*)`	largest domain size
`FLOAT_VAR_DEGREE_SIZE_MIN(t*)`	smallest degree divided by domain size
`FLOAT_VAR_DEGREE_SIZE_MAX(t*)`	largest degree divided by domain size
`FLOAT_VAR_AFC_SIZE_MIN(afc+,t*)`	smallest AFC divided by domain size
`FLOAT_VAR_AFC_SIZE_MAX(afc+,t*)`	largest AFC divided by domain size
`FLOAT_VAR_ACTIVITY_SIZE_MIN(act+,t*)`	smallest activity divided by domain size
`FLOAT_VAR_ACTIVITY_SIZE_MAX(act+,t*)`	largest activity divided by domain size

Figure 8.5: Float variable selection

An overview of the available value selection strategies for set variables can be found in Figure 8.4 where $\cdot^*$ denotes an optional argument. Here, an argument `r` refers to a random number generator of type `Rnd` which is discussed in Section 8.6. An argument `v` refers to a value selection function of type `SetBranchVal`. An optional argument `c` refers to a commit function of type `SetBranchCommit`. Value and commit function are discussed in Section 8.8.

8.4 Branching on float variables

Important. Do not forget to add

```
#include <gecode/float.hh>
```

to your program when you want to branch on float variables.

For float variables, variable selection is defined by a value of class `FloatVarBranch` (see also Variable selection for float variables) and value selection is defined by a value of type `FloatValBranch` (see also Value selection for float variables).

`FLOAT_VAL(v,c*)`	defined by value function `v` and commit function `c`
`FLOAT_VAL_SPLIT_RND(r)`	values not smaller or larger than mean (smaller or larger is randomly selected)
`FLOAT_VAL_SPLIT_MIN()`	values not greater than mean
`FLOAT_VAL_SPLIT_MAX()`	values not smaller than mean

Figure 8.6: Float value selection

For an overview of the available variable selection strategies, see Figure 8.5 (see also Variable selection for float variables) where $\cdot^*$ denotes an optional argument and $\cdot^+$ is a special argument to be explained below. Here, an argument `r` refers to a random number generator of type `Rnd`. Using random number generators for branching is discussed in Section 8.6. An argument `m` refers to a user-defined merit function of type `FloatBranchMerit`. User-defined merit functions are discussed in Section 8.7. An argument `afc` refers to accumulated failure count (AFC) information for float variables (of class `FloatAFC`). An argument `act` refers to activity information for float variables (of class `FloatActivity`). For a discussion of AFC and activity, see Section 8.5, both `afc`$^+$ and `act`$^+$ can also be optional arguments of type **`double`** defining a decay-factor. The optional argument `t` refers to a tie-breaking limit function of type `BranchTbl` and is discussed in Section 8.9.

An overview of the available value selection strategies for float variables can be found in Figure 8.6 where $\cdot^*$ denotes an optional argument. Here, an argument `r` refers to a random number generator of type `Rnd` which is discussed in Section 8.6. An argument `v` refers to a value selection function of type `FloatBranchVal`. An optional argument `c` refers to a commit function of type `FloatBranchCommit`. Value and commit function are discussed in Section 8.8.

8.5 Local versus shared variable selection criteria

The criteria used for selecting variables are either *local* or *shared*. A *local* variable selection criterion depends only on a brancher's home space. A *shared* variable selection criterion depends not only on the brancher's home space but also on all spaces that have been created during search sharing the same root space where the brancher had originally been posted. That entails that a shared criterion can use information that is collected during search. In terms of Section 9.1, a shared variable selection criterion depends on all equivalent spaces created by cloning.

8.5.1 Local variable selection criteria

All selection criteria but those based on *AFC* and *activity* are local: they either select variables without using any information on a variable (`INT_VAR_NONE()`), select variables randomly (`INT_VAR_RND(r)`, see also Section 8.6), or use the degree or domain of a variable for selec-

tion. The user-defined selection criteria `INT_VAR_MERIT_MIN()` and `INT_VAR_MERIT_MAX()` in addition have access to the home space and the selected variable's position, see Section 8.7 for details.

The *degree* of a variable is the number of propagators depending on the variable (useful as an approximate measure of how constrained a variables is).

The *minimum-regret* for integer and Boolean variables is the difference between the smallest and second smallest value in the domain of a variable (*maximum-regret* is analogous).

8.5.2 Selection using accumulated failure count

The accumulated failure count (AFC) of a variable is a shared selection criterion. It is defined as the sum of the AFCs of all propagators depending on the variable plus its degree (to give a good initial value if the AFCs of all propagators are still zero). The AFC of a propagator counts how often the propagator has failed during search. The AFC of a variable is also known as the weighted degree of a variable [9].

AFC in Gecode supports decay as follows. Each time a propagator fails during constraint propagation (by executing the `status()` function of a space, see also Tip 2.2), the AFC of all propagators is updated:

- If the propagator `p` failed, the AFC `afc(p)` of `p` is incremented by 1:

  $$\texttt{afc(p)} = \texttt{afc(p)} + 1$$

  For all other propagators `q`, the AFC `afc(p)` of `q` is updated by a decay-factor `d` ($0 < \texttt{d} \leq 1$):

  $$\texttt{afc(q)} = \texttt{d} \cdot \texttt{afc(q)}$$

- The AFC `afc(x)` of a variable `x` is then defined as:

  $$\texttt{afc(x)} = \texttt{afc}(\texttt{p}_1) + \cdots + \texttt{afc}(\texttt{p}_n)$$

  where the propagators $\texttt{p}_i$ depend on `x`.

- The AFC `afc(p)` of a propagator `p` is initialized to 1. That entails that the AFC of a variable `x` is initialized to its degree.

In order to use AFC for branching, one must create an object of class `IntAFC` for integer or Boolean variables, an object of class `SetAFC` for set variables, or an object of class `FloatAFC` for float variables. The object is responsible for recording AFC information[1].

[1] Gecode cheats a little bit with the implementation of AFC: while it is possible (but not common) to have more than a single AFC object, all will use the same decay-factor `d`. The decay-factor used is the one defined by the AFC object created last. But as using several AFC objects with different decay-factors is not really that useful, Gecode takes a shortcut here.

If x is an integer variable array, then

```
IntAFC afc(home,x,0.99);
```

initializes the AFC information afc for the variables in x with decay-factor d = 0.99. The decay-factor is optional and defaults to no decay (d = 1).

The decay-factor can be changed later, say to d = 0.95, by

```
afc.decay(0.95);
```

and afc.decay() returns the current decay-factor of afc.

A branching for integer variables using AFC information must be given an object of type IntAFC as argument:

```
branch(home, x, INT_VAR_AFC_MAX(afc), INT_VAL_MIN());
```

Here the integer variable array x must be exactly the same that has been used for creating the integer AFC object afc.

The AFC object can be omitted if one does not want to change the decay-factor later, hence it is sufficient to pass the decay-factor as argument. For example:

```
branch(home, x, INT_VAR_AFC_MAX(0.99), INT_VAL_MIN());
```

uses AFC information with a decay-factor of 0.99. Even the decay-factor can be omitted and defaults to 1 (that is, no decay).

AFC for other variable types is analogous.

For an example using a decay-factor with AFC, see Section 21.4.

8.5.3 Selection using activity

The activity of a variable is a shared criterion and captures how much a variable has been involved in constraint propagation.

The activity of a variable is maintained by constraint propagation as follows. Each time constraint propagation finishes during search (by executing the status() function of a space, see also Tip 2.2), the activity of a variable x is updated [33]:

- If the variable x has not been pruned (that is, no values have been removed from the domain of x through propagation), the activity activity(x) of x is updated by a decay-factor d ($0 < d \leq 1$):

$$\texttt{activity}(x) = d \cdot \texttt{activity}(x)$$

- If the variable x has been pruned, the activity activity(x) of x is incremented by 1:

$$\texttt{activity}(x) = \texttt{activity}(x) + 1$$

- The activity of a variable x is initialized to be zero.

In order to use activity for branching, one must create an object of class `IntActivity` for integer or Boolean variables, an object of class `SetActivity` for set variables, or an object of class `FloatActivity` for float variables. The object is responsible for recording activity information.

If x is an integer variable array, then

```
IntActivity act(home,x,0.99);
```

initializes the activity information `act` for the variables in x with decay-factor `d` = `0.99`. The decay-factor is optional and defaults to no decay (`d` = 1).

The activity of each variable in an array x can be initialized by a merit function, see Section 8.7. If `bm` is a merit function, then

```
IntActivity act(home,x,0.99,&bm);
```

initializes the activity of `x[i]` to the value returned by `bm(home,x[i],i)`.

The decay-factor can be changed later, say to `d` = `0.95`, by

```
act.decay(0.95);
```

and `act.decay()` returns the current decay-factor of `act`.

A branching for integer variables using activity information must be given an object of type `IntActivity` as argument:

```
branch(home, x, INT_VAR_ACTIVITY_MAX(act), INT_VAL_MIN());
```

Here the integer variable array x must be exactly the same that has been used for creating the integer activity object `act`.

The activity object can be omitted if one does not want to change the decay-factor later, hence it is sufficient to pass the decay-factor as argument. For example:

```
branch(home, x, INT_VAR_ACTIVITY_MAX(0.99), INT_VAL_MIN());
```

uses activity information with a decay-factor of `0.99`. Even the decay-factor can be omitted and defaults to `1` (that is, no decay).

Activity for other variable types is analogous.

8.6 Random variable and value selection

One particular strategy for variable and value selection is by random. For integer and Boolean variables, `INT_VAR_RND(r)` selects a random variable and `INT_VAL_RND(r)` selects a random value where `r` is a random number generator of class `Rnd`. For set variables, `SET_VAR_RND(r)` selects a random variable and `SET_VAL_RND_INC(r)` and `SET_VAL_RND_EXC(r)` include and exclude a random value from a set variable. For float

variables, `FLOAT_VAR_RND(r)` selects a random variable and `FLOAT_VAL_SPLIT_RND(r)` randomly selects the lower or upper half of the domain of a float variable.

The random number generators used for random variable and value selection follow a uniform distribution and must be initialized by a seed value. For example, a random number generator `r` is created and initialized with a seed value of `1` (the seed value must be an **`unsigned int`**) by

```
Rnd r(1U);
```

The seed value can be changed with the `seed()` function (if needed, the `seed()` function initializes the random number generator). For example, by

```
r.seed(2U);
```

the seed value is set to `2` (the `seed()` function also expects an argument of type **`unsigned int`**).

A random number generator is passed by reference to the brancher. In the terms of Section 4.2, a random number generator is a proper data structure. When a random number generator is stored as a member of a space it must be updated by using the `update()` function of the random number generator.

It is possible to use the same random number generator for both variable and value selection. For example, by

```
Rnd r(1U);
branch(home, x, INT_VAR_RND(r), INT_VAL_RND(r));
```

both the variable in `x` as well as its value are randomly selected using the numbers generated by `r`. It is of course also possible to use two separate random number generators as in:

```
Rnd r1(1U), r2(1U);
branch(home, x, INT_VAR_RND(r1), INT_VAL_RND(r2));
```

8.7 User-defined variable selection

Variables can be selected according to user-defined criteria implemented as a *merit function*. For integer variables, the type of the merit function is `IntBranchMerit`, for Boolean variables `BoolBranchMerit`, for set variables `SetBranchMerit`, and for float variables `FloatBranchMerit`. For integer variables, the type is defined as

```
typedef double (*IntBranchMerit)(const Space& home, IntVar x, int i);
```

where `home` refers to the home space, `x` is the integer variable for which a merit value should be computed and `i` refers to the position of `x` in the integer variable array passed as argument to the `branch()` function. The merit function types for Boolean, set, and float variables are analogous.

For example, the following static merit function

```
static double m(const Space& home, IntVar x, int i) {
  return x.size();
}
```

simply returns the domain size of the integer variable `x` as the merit value. The merit function can be used to select a variable with either smallest or largest merit value. By

```
branch(home, INT_VAR_MERIT_MIN(&m), INT_VAL_MIN());
```

a variable with least merit value according to the merit function `m()` is selected (that is, the first variable in the array with smallest size). A variable with maximal merit value is selected by:

```
branch(home, INT_VAR_MERIT_MAX(&m), INT_VAL_MIN());
```

Tip 8.3 (Using a member function as merit function)**.** Often it is more convenient to use a const member function instead of a static function as a merit function. This is in particular the case when the merit function needs access to members of a script.

For example, the following member function of a class `Model` for a script (a subclass of `Space`):

```
double merit(IntVar x, int i) const {
  return ···;
}
```

is used when defining a static function as follows

```
double trampoline(const Space& home, IntVar x, int i) {
  return static_cast<const Model&>(home).merit(x,i);
```

and passing a pointer to `trampoline()` as an argument, for example by

```
branch(home, INT_VAR_MERIT_MIN(&trampoline), INT_VAL_MIN());
```

◀

8.8 User-defined value selection

The value selected for branching and how the selected value is used for branching can be defined by *branch value functions* and *branch commit functions*.

A branch value function takes a constant reference to a space, a variable, and the variable's position and returns a value, where the type of the value depends on the variable type. Figure 8.7 lists the branch value function types and the value types for the different variable types. For example, the type `IntBranchVal` for value functions for integer variables is defined as:

Variable type	Value function type	Value type
IntVar	IntBranchVal	**int**
BoolVar	BoolBranchVal	**int**
SetVar	SetBranchVal	**int**
FloatVar	FloatBranchVal	FloatNumBranch

Figure 8.7: Branch value functions

```
typedef int (*IntBranchVal)(const Space& home, IntVar x, int i);
```

A branch commit function takes a reference to a space, the number of the alternative a (`0` for the first alternative and `1` for the second alternative), a variable, the variable's position, and a value selected by a branch value function. For example, the type `IntBranchCommit` for branch commit functions for integer variables is defined as:

```
typedef void (*IntBranchCommit)(Space& home, unsigned int a,
                                IntVar x, int i, int n);
```

Let us consider `INT_VAL_MIN()` as an example, but re-implemented by value and commit functions. The value function can be defined as:

```
static int v(const Space& home, IntVar x, int i) {
  return x.min();
}
```

and the commit function as:

```
static void c(Space& home, unsigned int a,
              IntVar x, int i, int n) {
  if (a == 0U) {
    rel(home, x, IRT_EQ, n);
  } else {
    rel(home, x, IRT_NQ, n);
  }
}
```

A branching using the value and commit function then can be posted by:

```
branch(home, x, INT_VAR_NONE(), INT_VAL(&v,&c));
```

The commit function is optional. If the commit function is omitted, a default commit function depending on the variable type is used. For integer variables, for example, the commit function corresponds to the commit function from the previous example. Hence, it is sufficient to post the brancher as:

```
branch(home, x, INT_VAR_NONE(), INT_VAL(&v));
```

Variable type	Commit function type	Default behavior
`IntVar`	`IntBranchCommit`	$(x = n) \vee (x \neq n)$
`BoolVar`	`BoolBranchCommit`	$(x = n) \vee (x \neq n)$
`SetVar`	`SetBranchCommit`	$(n \in x) \vee (n \notin x)$
`FloatVar`	`FloatBranchCommit`	$(x \leq n) \vee (x \geq n)$

Figure 8.8: Branch commit functions

Figure 8.8 lists the commit function types and the behavior of the default commit function for the different variable types. The variable `x` refers to the variable selected by the brancher and `n` to the value selected by the branch value function.

Note that both value and commit functions can also be implemented as member functions, similar to merit functions as described in Tip 8.3.

For examples which use value functions to implement problem-specific branching, see Black hole patience and The balanced academic curriculum problem.

8.9 Tie-breaking

The default behavior for tie-breaking during variable selection is that the first variable (that is the variable with the lowest index in the array) satisfying the selection criteria is selected. For many applications that is not sufficient.

A typical example for integer variables is to select a most constrained variable first (the variable most propagators depend on, that is, with largest degree). Then, among the most constrained variables select the variable with the smallest domain. This can be achieved by using the `tiebreak()` function:

```
branch(home, x, tiebreak(INT_VAR_DEGREE_MAX(), INT_VAR_SIZE_MIN()),
                INT_VAL_MIN());
```

The overloaded function `tiebreak()` (see Tie-breaking for variable selection) takes up to four variable selection values.

Random selection is particularly interesting for tie-breaking. For example, breaking ties by first selecting a variable with smallest domain and then selecting a random variable among those with smallest domain is obtained by:

```
branch(home, x, tiebreak(INT_VAR_SIZE_MIN(), INT_VAR_RND(r)),
                INT_VAL_MIN());
```

Here, `r` must be a random number generator as discussed in Section 8.6.

Using tie-breaking limit functions. In the discussion so far only exact ties have been considered. Often it is necessary to consider several variables as ties even though some of them are not among the best variables. Which variables are considered as ties can be controlled by *tie-breaking limit functions*.

A tie-breaking limit function has the type `BranchTbl` which is defined as:

```
typedef double (*BranchTbl)(const Space& home, double w, double b);
```

The function takes a constant reference to a space `home`, the worst merit value `w`, and the best merit value `b` as arguments. The value returned by the function determines which variables are considered as ties.

Let us consider an example where we branch over four integer variables from the integer variable array `x` where the domains of the variables are as follows:

$$\texttt{x[0]} \in \{1,2,3,4\} \qquad \texttt{x[1]} \in \{2,3,4\} \qquad \texttt{x[2]} \in \{1,2,4\} \qquad \texttt{x[3]} \in \{1,2,3,4,5,6,7\}$$

Without a tie-breaking limit function as in (here, `r` is a random number generator):

```
branch(home, x, tiebreak(INT_VAR_SIZE_MIN(), INT_VAR_RND(r)),
                INT_VAL_MIN());
```

the variables `x[1]` and `x[2]` (both with size as the merit value 3.0) are considered as ties and random variable selection will choose one of them.

Likewise, when branching with

```
branch(home, x, tiebreak(INT_VAR_SIZE_MAX(), INT_VAR_RND(r)),
                INT_VAL_MIN());
```

only variable `x[3]` will be considered as the single variable with the best merit value 7.0.

The following tie-breaking limit function (defined as a static member function of a script, see also Tip 8.3):

```
static double tbl(const Space& home, double w, double b) {
  return (w + b) / 2.0;
}
```

returns the average of the worst merit value `w` and the best merit value `b`. Using the function `tbl()` for tie-breaking is done by passing it as additional argument.

For example, when using `tbl()` with

```
branch(home, x, tiebreak(INT_VAR_SIZE_MIN(&tbl), INT_VAR_RND(r)),
                INT_VAL_MIN());
```

the function `tbl()` is called with $\texttt{w} = 7.0$ and $\texttt{b} = 3.0$ and returns $(7.0+3.0)/2.0 = 5.0$. Hence, the three variables `x[0]`, `x[1]`, and `x[2]` are considered for tie-breaking and random selection will make a choice among these three variables.

For example, when using `tbl()` with

LATIN SQUARE LDSB ≡ [DOWNLOAD]

```
class LatinSquare : public Script {
  ...
  LatinSquare(const SizeOptions& opt)
    : n(opt.size()), x(*this,n*n,0,n-1) {
    Matrix<IntVarArgs> m(x, n, n);
    for (int i=0; i<n; i++)
      distinct(*this, m.row(i));
    for (int i=0; i<n; i++)
      distinct(*this, m.col(i));
    ▶ SYMMETRY BREAKING
  }
  ...
};
```

Figure 8.9: A Gecode model for Latin Squares with LDSB

```
branch(home, x, tiebreak(INT_VAR_SIZE_MAX(&tbl), INT_VAR_RND(r)),
                INT_VAL_MIN());
```

the function `tbl()` is called with `w` $= 3.0$ and `b` $= 7.0$ and returns $(3.0 + 7.0)/2.0 = 5.0$. Hence, only variable `x[3]` is considered for tie-breaking.

Note that worse and best depends on whether the variable selection tries to minimize or maximize the merit value. If a tie-breaking limit function returns a value that is worse than the worst merit value, all variables are considered for tie-breaking. If a function returns a value that is better than the best value, the returned value is ignored and the best value is considered as limit (in which case, tie-breaking works exactly the same as if not using a tie-breaking limit function at all).

8.10 Lightweight Dynamic Symmetry Breaking

Gecode supports automatic symmetry breaking with *Lightweight Dynamic Symmetry Breaking* (LDSB [30]). To use LDSB, you specify your problem's symmetries as part of the `branch()` function.

Consider the model for the Latin Square problem in Figure 8.9. A Latin Square is an $n \times n$ matrix (see Section 7.2) where each cell takes a value between 0 and $n-1$ and no two values in a row or a column are the same. This is easily implemented using integer variables and `distinct` constraints.

The model has many solutions that are essentially the same due to symmetry. For example, the four solutions in Figure 8.10 are symmetric: from the top-left solution, we can get the top-right one by exchanging the first two rows, the bottom-left one by exchanging the second and third column, and the bottom-right one by swapping the values 1 and 3.

0	1	2	3
1	0	3	2
2	3	0	1
3	2	1	0

1	0	3	2
0	1	2	3
2	3	0	1
3	2	1	0

0	2	1	3
1	3	0	2
2	0	3	1
3	1	2	0

0	3	2	1
3	0	1	2
2	1	0	3
1	2	3	0

Figure 8.10: Symmetric solutions of the Latin Square problem

Gecode supports *dynamic symmetry breaking*, i.e., given a specification of the symmetries, it can avoid visiting symmetric states during the search, which can result in dramatically smaller search trees and greatly improved runtime for some problems.

Symmetries are specified by passing an object of type `Symmetries` to the `branch()` function. In the case of Latin Squares, we can easily break the value symmetry (that is, values that are interchangeable) as follows:

```
SYMMETRY BREAKING ≡
  Symmetries syms;
  syms << ValueSymmetry(IntArgs::create(n,0));
  ▶ ROW/COLUMN SYMMETRY
  branch(*this, x, INT_VAR_NONE(), INT_VAL_MIN(), syms);
```

Here, `IntArgs::create(n,0)` creates an array of integers with values $0, 1, \ldots, n-1$ which specifies that all these values are symmetric, that is, interchangeable.

For the row and column symmetries, we need to declare a `VariableSequenceSymmetry` (see Symmetry declarations), which states that certain *sequences* of variables (in this case the rows and columns) are interchangeable:

```
ROW/COLUMN SYMMETRY ≡
  IntVarArgs rows;
  for (int r = 0; r < m.height(); r++)
    rows << m.row(r);
  syms << VariableSequenceSymmetry(rows, m.width());
  IntVarArgs cols;
  for (int c = 0; c < m.width(); c++)
    cols << m.col(c);
  syms << VariableSequenceSymmetry(cols, m.height());
```

Now the number of Latin squares found and the search effort required are greatly reduced. The code for the example in Figure 8.9 has command line options for toggling between no symmetry breaking and LDSB.

For examples, consider Clique-based graph coloring and Steel-mill slab design problem.

8.10.1 Specifying Symmetry

LDSB supports four basic types of symmetry (see Symmetry declarations). Collections of symmetries are stored in a `Symmetries` object, which is passed to the `branch()` function. Any combination of symmetries is allowed.

- A `VariableSymmetry` represents a set of *variables* that are interchangeable.
- A `ValueSymmetry` represents a set of *values* that are interchangeable.
- A `VariableSequenceSymmetry` represents a set of *sequences of variables* that are interchangeable.
- A `ValueSequenceSymmetry` represents a set of *sequences of values* that are interchangeable.

In addition to constructing these symmetries directly, there are also some convenient functions for creating common kinds of symmetry:

- `values_reflect()`, to map L to U, $L+1$ to $U-1$ and so on, where L and U are the bounds of a variable
- `rows_interchange()`, to specify that the rows of a matrix are interchangeable (see `Gecode::Matrix`)
- `columns_interchange()`, to specify that the columns of a matrix are interchangeable (see `Gecode::Matrix`)
- `rows_reflect()`, to specify that a matrix's rows can be reflected (first row to last row, second row to second-last row and so on, see `Gecode::Matrix`)
- `columns_reflect()`, to specify that a matrix's columns can be reflected (see `Gecode::Matrix`)
- `diagonal_reflect()`, to specify that a matrix can be reflected around its main diagonal (the matrix must be square, see `Gecode::Matrix`)

8.10.2 Notes

Symmetry breaking by LDSB is not guaranteed to be complete. That is, a search may still return two distinct solutions that are symmetric.

Combining LDSB with other forms of symmetry breaking — such as static ordering constraints — is not safe in general, and can cause the search to miss some solutions.

BRANCH FILTER FUNCTION SKETCH ≡

```
class Model : public Space {
protected:
  IntVarArray x;
public:
  Model(void) : ··· {
    ...
    ▶ POST BRANCHING
  }
  ...
  ▶ DEFINE FILTER FUNCTION
};
```

Figure 8.11: Model sketch for branch filter function

LDSB works with integer, Boolean, and set variables, and with any variable selection strategy. For integer variables, only value selection strategies that result in the variable being assigned on the left branch (such as `INT_VAL_MIN()`, `INT_VAL_MED()`, `INT_VAL_MAX()` or `INT_VAL_RND()`) are supported, other parameters throw an exception.

8.11 Using branch filter functions

By default, a variable-value branching continues to branch until all variables passed to the branching are assigned. This behavior can be changed by using a *branch filter function*.

A branch filter function is called during branching for each variable to be branched on. If the filter function returns **`true`**, the variable is considered for branching. Otherwise, the variable is simply ignored.

A branch filter function can be passed as the second to last (optional) argument when calling the `branch()` function.

The type of a branch filter function depends on the variable type. For integer variables, the type `IntBranchFilter` is defined as

```
typedef bool (*IntBranchFilter)(const Space& home, IntVar x, int i);
```

That is, a branch filter function takes the `home` space and the position `i` of the variable `x` as argument. The position `i` refers to the position of the variable `x` in the array of variables used for posting the branching. For Boolean variables, the type is `BoolBranchFilter`, for set variables `SetBranchFilter`, and for float variables `FloatBranchFilter`.

Assume, for example, that we want to branch only on variables from a variable array `x` for branching with a domain size of at least `4`. Consider the sketch of a model shown in Figure 8.11.

The branch filter function can be defined as a static member function of the class `Model` as follows:

```
DEFINE FILTER FUNCTION ≡
  static bool filter(const Space& home, IntVar y, int i) {
    return y.size() >= 4;
  }
```

Specifying that the branching should use the filter function is done as follows:

```
POST BRANCHING ≡
  branch(home, x, ···, ···, &filter);
```

In many cases it can be more convenient to perform the actual filtering in a **const** member function. This can be done along the lines of Tip 8.3.

8.12 Using variable-value print functions

Search engines such as Gist (see Section 10.3.4) or others (see Tip 39.1) use `print()` functions provided by branchers to display information about the alternatives that are explored during search. The information displayed for variable-value branchers can be programmed by using a *variable-value print function*.

A variable-value print function can be passed as the last (optional) argument when calling the `branch()` function.

The type of a variable-value print function depends on the variable type. For integer variables, the type `IntVarValPrint` is defined as

```
  typedef void (*IntVarValPrint)(const Space &home,
                                 const BrancherHandle& bh,
                                 unsigned int a,
                                 IntVar x, int i, const int& n,
                                 std::ostream& o);
```

That is, a variable-value print function takes the `home` space, a handle to the brancher `bh`, the number of the alternative `a`, the position `i` of the variable `x`, and the integer value `n` as argument. The information will be printed on the standard output stream `o`. The position `i` refers to the position of the variable `x` in the array of variables used for posting the branching. For Boolean variables, the type is `BoolVarValPrint`, for set variables `SetVarValPrint`, and for float variables `FloatVarValPrint`.

For an example of how to use variable-value print functions, see Chapter 21. In many cases it can be more convenient to perform the actual printing in a **const** member function. This can be done along the lines of Tip 8.3.

8.13 Assigning integer, Boolean, set, and float variables

A special variant of branching is *assigning* variables: for a not yet assigned variable the branching creates a single alternative which assigns the variable a value. The effect of assigning is that assignment is interleaved with constraint propagation. That is, after an assignment has been done, the next assignment will be done only after the effect of the previous assignment has been propagated.

For example, the next code fragment assigns all integer variables in `x` their smallest possible value:

```
assign(home, x, INT_ASSIGN_MIN());
```

The strategy to select the value for assignment is defined by a value of class `IntAssign` (see also Value selection for assigning integer and Boolean variables) for integer and Boolean variables, by a value of class `SetAssign` (see also Assigning set variables) for set variables, and by a value of class `FloatAssign` (see also Value selection for assigning float variables) for float variables.

Figure 8.12 summarizes the value selection strategies for assigning integer, Boolean, set, and float variables. Here, an argument `r` refers to a random number generator of type `Rnd` which is discussed in Section 8.6. An argument `v` refers to a value selection function of type `IntBranchVal` for integer variables, `BoolBranchVal` for Boolean variables, `SetBranchVal` for set variables, and `FloatBranchVal` for float variables. An optional argument `c` refers to a commit function of type `IntBranchCommit` for integer variables, of type `BoolBranchCommit` for Boolean variables, of type `SetBranchCommit` for set variables, and of type `FloatBranchCommit` for float variables. Value and commit function can be used in the same way for assigning than for branching as described in Section 8.8. The only difference is that the number of the alternative passed to the commit function is always zero (as there is only a single alternative).

An assignment also accepts a branch filter function as described in Section 8.11 and can assign a single variable.

8.14 Executing code between branchers

A common scenario is to post some constraints only after part of the branching has been executed. This is supported in Gecode by a brancher (see Branch with a function) that executes a function (either a function or a static member function but not a member function).

`INT_ASSIGN_MIN()`	smallest value
`INT_ASSIGN_MED()`	median value (rounding downwards)
`INT_ASSIGN_MAX()`	maximum value
`INT_ASSIGN_RND(r)`	random value
`INT_ASSIGN(v,c*)`	defined by value function `v` and commit function `c`

`SET_ASSIGN_MIN_INC()`	include smallest element
`SET_ASSIGN_MIN_EXC()`	exclude smallest element
`SET_ASSIGN_MED_INC()`	include median element (rounding downwards)
`SET_ASSIGN_MED_EXC()`	exclude median element (rounding downwards)
`SET_ASSIGN_MAX_INC()`	include largest element
`SET_ASSIGN_MAX_EXC()`	exclude largest element
`SET_ASSIGN_RND_INC(r)`	include random element
`SET_ASSIGN_RND_EXC(r)`	exclude random element
`SET_ASSIGN(v,c*)`	defined by value function `v` and commit function `c`

`FLOAT_ASSIGN_MIN()`	median value of lower part
`FLOAT_ASSIGN_MAX()`	median value of upper part
`FLOAT_ASSIGN_RND(r)`	median value of randomly chosen part
`FLOAT_ASSIGN(v,c*)`	defined by value function `v` and commit function `c`

Figure 8.12: Value selection for assigning variables

Suppose the following code fragment defining a model `Model`:

```
class Model : public Space {
public:
  Model(void) : ··· {
    ···
    ▶ POST BRANCHINGS
  }
  ···
  ▶ DEFINE FUNCTIONS
};
```

where the constructor posts two branchers

```
POST BRANCHINGS ≡
  branch(home, x, INT_VAR_NONE(), INT_VAL_MIN());
  branch(home, &Model::post);
```

The second branching takes a function pointer to the static member function `Model::post` which is defined as

```
DEFINE FUNCTIONS ≡
  void more(void) {
    ···
  }
  static void post(Space& home) {
    static_cast<Model&>(home).more();
  }
```

As soon as the first branching is finished, the second branching is executed. This branching provides just a single alternative that calls the function `Model::post` with the current space as its argument. Then, the function casts the `home` to `Model&` and calls `more` on `home`. While one could post the additional constraints and/or branchings in `Model::post` directly, the member function `Model::more` is more convenient to use.

Tip 8.4 (Propagation is still explicit)**.** It is tempting to believe that the variables in `x` in the above example are all assigned when `more()` is executed. This is not necessarily true.

It will be true for the first time `more()` is executed. But `more()` will be executed possibly quite often during recomputation (see the following Section). And then, the only guarantee one can rely on is that the brancher has created enough alternatives to guarantee that the variables in `x` are assigned *but only after constraint propagation has been performed* (see Tip 2.2). ◀

9 Search

This chapter discusses how *exploration* for search is used for solving Gecode models. Exploration defines a strategy how to explore parts of the search tree and how to possibly modify the tree's shape during exploration (for example, during branch-and-bound best solution search by adding new constraints). This chapter restricts itself to simple search engines to find solutions, Gist as an interactive and graphical search engine is discussed in Chapter 10.

Overview. Section 9.1 explains how search in Gecode makes use of hybrid recomputation and why it is efficient. Even though this section does not belong to the basic reading material, you are highly encouraged to read it.

Section 9.2 explains how parallel search can be used in Gecode and what can be expected of parallel search in principle. How search engines can be used is explained in Section 9.3. Restart-based search is discussed in Section 9.4, followed by a discussion in Section 9.5 how no-goods from restarts can be used.

Convention. Note that the same conventions hold as in Chapter 4.

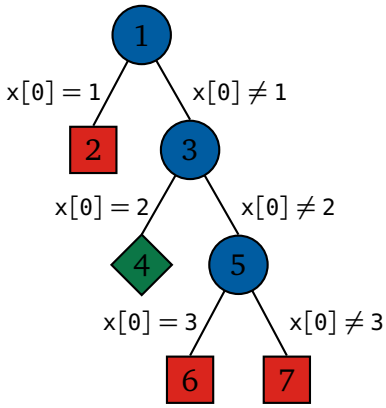
9.1 Hybrid recomputation

A central requirement for search is that it can return to previous states: as spaces constitute nodes of the search tree, a previous state is nothing but a space. Returning to a previous space might be necessary because an alternative suggested by a branching did not lead to a solution, or even if a solution has been found more solutions might be requested. As propagation and branching change spaces, provisions must be taken that search can actually return to a previous space, or an equivalent version of that space.

Two space are *equivalent* if propagation and branching and hence search behave exactly the same on both spaces. Equivalent spaces can be different, for example, they contain different yet equivalent propagators, or are allocated at a different memory area.

Gecode employs a hybrid of two techniques for restoring spaces: *recomputation* and *cloning*.

If you want to know how search engines can be programmed in Gecode, please consult Part S.



node	x[0]	x[1]	x[2]
1	{1,...,5}	{1,...,5}	{2,...,6}
3	{2,...,5}	{1,...,3}	{3,...,6}
4	2	2	4
5	{3,...,5}	{1,...,2}	{4,...,6}

Figure 9.1: Example search tree

9.1.1 Cloning

Cloning creates a clone of a space (this is supported by the virtual `copy` member function as discussed in Chapter 2). A clone and the original space are of course equivalent. Restoration with cloning is straightforward: before following a particular alternative during search, a clone of the space is made and used later if necessary.

9.1.2 Recomputation

Recomputation remembers what has happened during branching: rather than storing an entire clone of a space just enough information to redo the effect of a brancher is stored. The information stored is called a *choice* in Gecode. Redoing the effect is called to *commit* a space: given a space and a choice committing re-executes the brancher as described by the choice and the alternative to be explored (for example, left or right).

Consider the following part of a model, which constrains both the sum and the product of `x[0]` and `x[1]` to be equal to `x[2]`:

```
IntVarArray x(home, 3, 1, 6);
rel(home, x[0] + x[1] == x[2]);
mult(home, x[0], x[1], x[2]);
branch(home, x, INT_VAR_NONE(), INT_VAL_MIN());
```

The corresponding search tree is shown in Figure 9.1. A red box corresponds to a failed node, a green diamond to a solution, and a blue circle to a choice node (a node that has a not-yet finished brancher left). An example choice for node 3 is $(\texttt{x[0]} = 2) \vee (\texttt{x[0]} \neq 2)$ where the left alternative (or the 0-th alternative) is $\texttt{x[0]} = 2$ and the right alternative (1-st alternative) is $\texttt{x[0]} \neq 2$. Committing a space for node 3 to the 1-st alternative posts the constraint $\texttt{x[0]} \neq 2$.

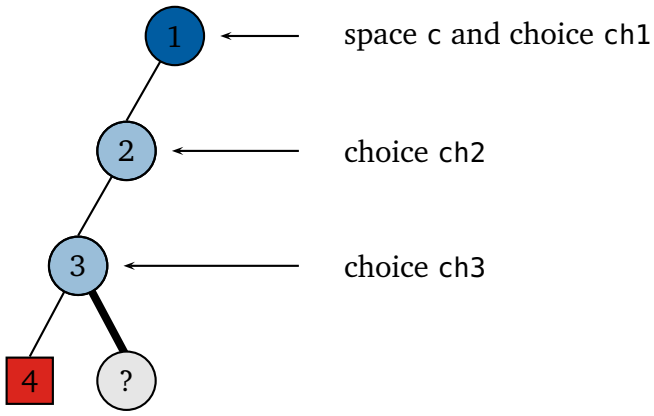


Figure 9.2: Hybrid recomputation

More precisely, a choice does not store the actual variables but the position among the variables of the brancher (storing `0` rather than `x[0]`). By that, a choice can be used with an equivalent yet different space. This is essential as the space used during recomputation will be different from the space for which the choice has been created.

Tip 9.1 (Search is indeterministic)**.** Gecode has been carefully designed to support non-monotonic propagators: they are essential for example for randomized or approximation propagation algorithms. A propagator in Gecode must be *weakly* monotonic: essentially, a propagator must be correct but it does not need to always prune exactly the same way. A consequence of this is that search is indeterministic: it might be that two different searches find solutions in a different order (possibly returning a different first solution) or that the number of explored nodes is different. However, search is always sound and complete: it never misses any solution, it does not duplicate solutions, nor does it report non-solutions as solutions.

If you want to know more about weakly monotonic propagators and their interaction with search, we recommend to consult [50]. ◀

9.1.3 Hybrid recomputation

The hybrid of recomputation and cloning works as follows. For each new choice node, a choice is stored. Then, every now and then search also stores a clone of a space (say, every eight steps). Now, restoring a space at a certain position in the search tree traverses the path in the tree upwards until a clone c is found on the path. Then recomputation creates a clone c' of c (in certain cases, recomputation might use c directly as an optimization). Then all choices on the path are committed on c' yielding an equivalent space.

To recompute the node ? for the example shown in Figure 9.2, the following operations are executed:

```
Space* s = c->clone();
s->commit(ch1, 0);
s->commit(ch2, 0);
s->commit(ch3, 1);
```

9.1.4 Why recomputation is almost for free

An absolutely fundamental property of the above hybrid is that an equivalent space is computed without performing any constraint propagation! Remember: committing just reposts constraints but does not perform constraint propagation.

Reconsider the example from Figure 9.2. Search has just failed at node 4 and must compute a space for node ?.

Suppose that only cloning but no recomputation is used. Then, a clone of the space for node 3 is created (from the clone that is stored in node 3) and that clone is committed to the first alternative of `d3` (this corresponds to the slightly thicker edge in Figure 9.2). After that, constraint propagation is performed (by executing the `status()` function of a space, see also Tip 2.2) to find out if and how search must continue. That is: there is one `clone` operation, one `commit` operation, and one `status` operation to perform constraint propagation.

With hybrid recomputation, one `clone` operation, three `commit` operations, and one `status` operation to perform constraint propagation are needed (as shown above). The good news is that `commit` operations are very cheap (most often, just modifying a single variable or posting a constraint). What is essential is that in both cases only a *single* `status` operation is executed. Hence, the cost for constraint propagation during hybrid recomputation turns out to be not much higher than the cost without recomputation.

For hybrid recomputation, some additional propagation might have to be done compared to cloning. As it turns out, the additional cost is rather small. This is due to the fact that constraint propagation executes all propagators that might be able to remove values for variables until no more propagation is possible (a fixpoint for the propagators is computed). Due to the approximative nature of "might be able to remove values" the additional propagation tends to show only a very small increase in runtime.

9.1.5 Adaptive recomputation

Consider the case that a search engine finds a failed node. That means that some brancher has made an erroneous decision and now search has to recover from that decision. It is quite likely that not only the last decision is wrong but that the decision that lead to failure is somewhere higher up in the search tree. With other words, it is quite likely that search following a depth-first left-most strategy must explore an entire failed subtree to recover from the erroneous decision. In that case it would be better for hybrid recomputation if there was a clone close to the failed node rather than far away.

To optimize recomputation in this example scenario, Gecode uses *adaptive recomputation*: if a node must be recomputed, adaptive recomputation creates an additional clone in the middle of the recomputation path. A clone created during adaptive recomputation is likely to be a good investment. Most likely, an entire failed subtree will be explored. Hence, the clone will be reused several times for reducing the amount of constraint propagation during recomputation.

More information about search based on recomputation (although not using choices) can be found in [44]. Search using choices has been inspired by batch recomputation [13] and decomposition-based search [32]. For an empirical evaluation of different techniques for search, see [39].

9.1.6 Controlling recomputation

Hybrid and adaptive recomputation can be easily controlled by two integers c_d (*commit distance*) and a_d (*adaptive distance*). The value for c_d controls how many clones are created during exploration: a search engine creates clones during exploration to ensure that recomputation executes at most c_d commit operations. The value for a_d controls adaptive recomputation: only if the clone for recomputation is more than a_d commit operations away from the node to be recomputed, adaptive recomputation is used.

Values for c_d and a_d are used to configure the behavior of search engines using hybrid and adaptive recomputation, see more in the next Section.

The number of commit operations as distance measure is approximately the same as the length of a path in the search tree. It is only an approximation as search engines use additional techniques to avoid some unused clone and commit operations.

Tip 9.2 (Values for c_d and a_d)**.** If $c_d = 1$, recomputation is never used (you might not want to try that for any other reason but curiosity; it takes too much memory to be useful). Likewise, to switch off cloning, you can use a value for c_d that is larger than the expected depth of the search tree. If $a_d \geq c_d$, adaptive recomputation is never used. ◀

9.2 Parallel search

Parallel search has but one motivation: try to make search more efficient by employing several threads (or workers) to explore different parts of the search tree in parallel.

Gecode uses a standard work-stealing architecture for parallel search: initially, all work (the entire search tree to be explored) is given to a single worker for exploration, making the worker busy. All other workers are initially idle, and try to steal work from a busy worker. Stealing work means that part of the search tree is given from a busy worker to an idle worker such that the idle worker can become busy itself. If a busy worker becomes idle, it tries to steal new work from a busy worker.

As work-stealing is indeterministic (depending on how threads are scheduled, machine load, and other factors), the work that is stolen varies over different runs for the very same

problem: an idle worker could potentially steal different subtrees from different busy workers. As different subtrees contain different solutions, it is indeterministic which solution is found first.

When using parallel search one needs to take the following facts into account (note that some facts are not particular to parallel search, check Tip 9.1: they are just more likely to occur):

- The order in which solutions are found might be different compared to the order in which sequential search finds solutions. Likewise, the order in which solutions are found might differ from one parallel search to the next. This is just a direct consequence of the indeterministic nature of parallel search.
- Naturally, the amount of search needed to find a first solution might differ both from sequential search and among different parallel searches. Note that this might actually lead to super-linear speedup (for n workers, the time to find a first solution is less than $1/n$ the time of sequential search) or also to real slowdown.
- For best solution search, the number of solutions until a best solution is found as well as the solutions found are indeterministic. First, any better solution is legal (it does not matter which one) and different runs will sometimes be lucky (or not so lucky) to find a good solution rather quickly. Second, as a better solution prunes the remaining search space the size of the search space depends crucially on how quickly good solutions are found.
- As a corollary to the above items, the deviation in runtime and number of nodes explored for parallel search can be quite high for different runs of the same problem.
- Parallel search needs more memory. As a rule of thumb, the amount of memory needed scales linearly with the number of workers used.
- For parallel search to deliver some speedup, the search tree must be sufficiently large. Otherwise, not all threads might be able to find work and idle threads might slow down busy threads by the overhead of unsuccessful work-stealing.
- From all the facts listed, it should be clear that for depth-first left-most search for just a single solution it is notoriously difficult to obtain consistent speedup. If the heuristic is very good (there are almost no failures), sequential left-most depth-first search is optimal in exploring the single path to the first solution. Hence, all additional work will be wasted and the work-stealing overhead might slow down the otherwise optimal search.

Tip 9.3 (Be optimistic about parallel search)**.** After reading the above list of facts you might have come to the conclusion that parallel search is not worth it as it does not exploit the parallelism of your computer very well. Well, why not turn the argument upside down: your machine will almost for sure have more than a single processing unit and maybe quite some. With sequential search, all units but one will be idle anyway.

```
GolombRuler
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 90, 105, 121}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 45, 61, 82, 96, 118}
        ··· (additional solutions omitted)
        m[12] = {0, 2, 6, 24, 29, 40, 43, 55, 68, 75, 76, 85}

Initial
        propagators: 58
        branchers:   1

Summary
        runtime:      14.866 (14866.000 ms)
        solutions:    17
        propagations: 519555681
        nodes:        3836351
        failures:     1918148
        restarts:     0
        no-goods:     0
        peak depth:   26
```

Figure 9.3: Output for Golomb rulers with eight workers

The point of parallel search is to make search go faster. It is not to perfectly utilize your parallel hardware. Parallel search makes good use (and very often excellent use for large problems with large search trees) of the additional processing power your computer has anyway.

For example, on my machine with eight cores and using Gecode 4.2.0, running Finding optimal Golomb rulers for size 12 as follows

```
golomb-ruler.exe -threads 8 12
```

prints something like shown in Figure 9.3.

Compared to sequential search where one gets something like shown in Figure 9.4 one gets a speedup of 7.2. ◀

Parallel search is controlled by the number of threads (or workers) used for search. If a single worker is requested, sequential search is used. The number of threads to be used for search is controlled by the search options passed to a search engine, see the following section for details.

```
GolombRuler
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 44, 65, 90, 105, 121}
        m[12] = {0, 1, 3, 7, 12, 20, 30, 45, 61, 82, 96, 118}
        ··· (additional solutions omitted)
        m[12] = {0, 2, 6, 24, 29, 40, 43, 55, 68, 75, 76, 85}

Initial
        propagators: 58
        branchers:   1

Summary
        runtime:      1:47.316 (107316.000 ms)
        solutions:    16
        propagations: 692676452
        nodes:        5313357
        failures:     2656663
        restarts:     0
        no-goods:     0
        peak depth:   24
```

Figure 9.4: Output for Golomb rulers with one worker

member	type	meaning
`propagate`	**`unsigned long int`**	propagators executed
`fail`	**`unsigned long int`**	failed nodes explored
`node`	**`unsigned long int`**	nodes explored
`restart`	**`unsigned long int`**	restarts performed
`nogood`	**`unsigned long int`**	no-goods generated
`depth`	**`unsigned long int`**	maximal depth of explored tree

Figure 9.5: Search statistics (partial)

Tip 9.4 (Do not optimize by branching alone)**.** A common modeling technique for optimization problems that does not work for parallel search is the following. Suppose, one has a variable `c` for the cost of a problem and one wants to minimize the cost. Then, one could use the following code fragment

```
branch(home, c, INT_VAL_MIN());
```

which will try the values for `c` in increasing order.

With sequential search, searching for the first solution with a standard depth-first left-most search engine will deliver a best solution, that is, a solution with least cost for `c`.

With parallel search, the first solution found might of course not be a best one. Hence, instead of using plain left-most depth-first search, one should use best solution search with a proper constrain function that guarantees that `c` will be minimized. This will as always guarantee that the last solution found is the best.

For an example, see the naive model for the bin packing case study in Section 19.2 where a branching first branches on the number of required bins. ◀

9.3 Search engines

Important. Do not forget to add

```
#include <gecode/search.hh>
```

to your program when you want to use search engines.

All search engines in Gecode are parametric (are templates) with respect to a subclass `T` of `Space` (for example, `SendMoreMoney` in Section 2.1). Moreover, all search engines share the same interface:

- The search engine is initialized by a constructor taking a pointer to an instance of the space subclass `T` as argument. By default, the search engine takes a clone of the space passed.

  This behavior can be changed, as can be other aspects of a search engine, see Section 9.3.1.

engine	shortcut	exploration	best solution	parallel
DFS	dfs	depth-first left-most		✓
BAB	bab	branch-and-bound	✓	✓

Figure 9.6: Available search engines

- A next solution can be requested by a `next()` member function. If no more solutions exist, `next()` returns `NULL`. Otherwise, the engine returns a solution which again is an instance of `T`. The client of the search engine is responsible for deleting solutions.
- A search engine can be asked for statistics information by the `statistics()` member function. The function returns an object of type `Search::Statistics`. The statistics information provided is partially summarized in Figure 9.5 (see Section 9.4 for the meaning of `restart` and Section 9.5 for the meaning of `nogood`).
- A search engine can be queried by `stopped()` whether the search engine has been stopped by a *stop object*. Stop objects are discussed in Section 9.3.2.
- The destructor deletes all resources used by the search engine.

Note that search engines use pointers to objects rather than references to objects. The reason is that some pointers might be `NULL`-pointers (for example, if `next()` fails to find a solution) and that users of search engines have to think about deleting solutions computed by search engines.

For each search engine there also exists a convenient shortcut function (of the same name but entirely in lowercase letters) that returns either the first solution or, in the case of best solution search, the last (and hence best) solution. The available search engines are summarized in Figure 9.6.

`BAB` continues search when a solution is found by adding a constraint (through the `constrain()` function as discussed in Section 2.5) to search for a better solution to all remaining nodes of the search tree.

9.3.1 Search options

All search engines can take a default option value of type `Search::Options` when being created. The options are summarized in Figure 9.7. The default values for the options are defined in the namespace `Search::Config`.

The meaning of the values for the search options are straightforward but for `threads` (`cutoff` is explained in Section 9.4 and `nogoods_limit` is explained in Section 9.5). Assume that your computer has m processing units[1] and that the value for `threads` is n.

[1]This is a very rough characterization: a processing unit could be a CPU, a processor core, or a multi-threading unit. If you want to find out how many processing units Gecode believes your machine has, invoke the configuration summary as described in Tip 3.3.

member	type	meaning
`threads`	**`double`**	number of parallel threads to use
`c_d` `a_d`	**`unsigned int`** **`unsigned int`**	commit recomputation distance adaptive recomputation distance
`clone`	**`bool`**	whether engine uses a clone when created
`nogoods_limit`	**`unsigned int`**	depth limit for no-good generation
`stop`	`Search::Stop*`	stop object (`NULL` if none)
`cutoff`	`Search::Cutoff*`	cutoff object (`NULL` if none)

Figure 9.7: Search options

- If $n = 0$, then m threads are used (as many as available processing units).
- If $n \geq 1$, then n threads are used (absolute number of threads to be used).
- If $n \leq -1$, then $m + n$ threads are used (absolute number of processing units not to be used). For example, when $n = -6$ and $m = 8$, then 2 threads are used.
- If $0 < n < 1$, then $n \cdot m$ threads are used (relative number of processing units to be used). For example, when $n = 0.5$ and $m = 8$, then 4 threads are used.
- If $-1 < n < 0$, then $(1 + n) \cdot m$ threads are used (relative number of processing units not to be used). For example, when $n = -0.25$ and $m = 8$, then 6 threads are used.

Note that all values are of course rounded and that at least one thread will be used.

9.3.2 Stop objects

A stop object (a subclass of `Search::Stop`) implements a single virtual member function `stop()` that takes two arguments, the first of type `Search::Statistics` and the second of type `Search::Options`, and returns either **`true`** or **`false`**. If a stop object is passed to a search engine (by passing it as `stop` member of a search option), the search engine calls the `stop()` function of the stop object before every exploration step and passes the current statistics as argument. If the `stop()` function returns true, the search engine stops its execution.

When a search engine is stopped its `next()` function returns `NULL` as solution. To find out whether a search engine has been stopped or whether there are no more solutions, the `stopped()` member function of a search engine can be used. Search can be resumed by calling `next()` again after the stop object has been modified (for example, by increasing the node or time limit).

Note that when using several threads for parallel search, each thread checks whether it is stopped independently using the very same stop object. If one thread is stopped, then the entire search engine is stopped.

class	description
`Search::NodeStop`	node limit exceeded
`Search::FailStop`	failure limit exceeded
`Search::TimeStop`	time limit exceeded

Figure 9.8: Predefined stop objects

Gecode provides several predefined stop objects, see Stop-objects for stopping search. For an overview see Figure 9.8.

Tip 9.5 (Number of threads for stop objects)**.** As mentioned above, each thread in parallel search uses the very same stop object. For example, when using the predefined `Search::NodeStop` stop object with a node limit of n, then each thread can explore up to n nodes.

If you want to have finer control (say, only allow each thread to explore up to n/m nodes where m is the number of threads) you can use the search option argument that is passed as the second argument to the `stop` member function to scale the node limit according to the number of available threads. ◀

9.4 Restart-based search

The idea of restart-based search is to run search with a given *cutoff* (Gecode uses the number of failures during search as cutoff-measure). When search reaches the cutoff, it is stopped and then restarted with a new and typically increased cutoff.

The whole point of restarting search is that it is not restarted on exactly the same problem but on a modified or re-configured problem. Possible modifications include but are not limited to:

- Improved information from the search for branching heuristics such as activity (see Section 8.5.3) or AFC (see Section 8.5.2): the now stopped search has gathered some information of which branching can take advantage in the next restart of search.
- The next search can use different random numbers that controls branching. A typical strategy would be to use tie-breaking for combining a branching heuristic with a random branching heuristic (see Section 8.9) and control the degree of randomness by a tie-breaking limit function.
- The next search uses an entirely different branching heuristic.
- The next search adds so-called no-goods derived from the now stopped search. No-goods are additional constraints that prevent the restarted search to make decisions during search that lead to failure in the stopped search. No-goods are discussed in detail in Section 9.5.

- The next search "keeps" only a randomly selected part of a previous solution and tries to find a different solution. This is often used for optimization problems and is known as LNS (Large Neighborhood Search) [52]. How restart-based can be used for LNS is discussed in Section 9.4.5.

For an example illustrating the effect of restart-based search, see Section 21.4. A general overview of restart-based search can be found in [58].

9.4.1 Restart-based search as a meta search engine

Restart-based search in Gecode is implemented as a *meta search engine*: the meta search engine uses one of the Gecode search engines discussed in Section 9.3 to perform search for each individual restart. In fact, the restart meta engine can use any search engine that implements the interface defined by `EngineBase`. The meta engine then controls the engine, the cutoff values, and how the problem is configured before each restart. The interface of the meta search engine in Gecode is exactly the same as the interface of a non-meta search engine.

The restart meta search engine `RBS` is parametric with respect to both the search engine to be used and the script to be solved (a subclass of `Space`). For example, when we want to use the `DFS` engine for the script `s` of class type `Script`, the meta engine `e` can be created by (`o` are mandatory search options, see below):

```
RBS<DFS,Script> e(s,o);
```

Now `e` implements exactly the same interface as the normal engines (that is, `next()` to request the next solution, `statistics()` to return the meta engine's statistic, and `stopped()` to check whether the meta engine has been stopped).

The meta engine honors all search options as discussed in Section 9.3.1. If parallel search is requested, the engine used by the meta engine will use the specified number of processing units to perform parallel search. The meta engine requires that the search options specify a `Search::Cutoff` object defining the cutoff sequence.

Tip 9.6 (Controlling restart-based search with the commandline driver)**.** The commandline driver transparently supports restart-based search. Depending on which options are passed on the commandline, either a search engine or the restart-based meta search engine is used for search. See Section 11.1 for details. ◀

9.4.2 Cutoff generators

The meta engine uses a cutoff generator that generates a sequence of cutoff values. A cutoff generator must be implemented by inheriting from the class `Search::Cutoff`. This abstract class requires that two virtual member functions **`operator()()`** and **`operator++()`** are implemented, where the first returns the current cutoff value and the second increments to the next cutoff value and returns it. Cutoff values are of type **`unsigned long int`**.

When using the restart meta engine, an instance of a subclass of `Search::Cutoff` must be passed to the engine by using the search options (see Section 9.3.1). For example, when `s` is a space to be solved and `c` a cutoff generator, then the restart engine can be created by:

```
Search::Options o;
o.cutoff = c;
RBS<DFS,Script> e(s,o);
```

Gecode provides some commonly used cutoff generators:

Geometric. A geometric cutoff sequence is defined by the scale-factor `s` and the base `b`. Then, the sequence consists of the cutoff values:

$$\mathtt{s} \cdot \mathtt{b}^i \qquad \text{for } i = 0, 1, 2, \ldots$$

A cutoff generator `c` for a geometric cutoff sequence with scale-factor `s` (of type **`unsigned long int`**) and base `b` (of type **`double`**) is created by:

```
Search::Cutoff* c = Search::Cutoff::geometric(s,b);
```

Luby. A Luby cutoff sequence is based on the Luby-sequence from [28]. The sequence starts with a `1`. The next part of the sequence is the entire previous sequence (only `1`) with the last value of the previous sequence (`1` again) doubled. This construction is then repeated, leading to the sequence:

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$$

To be practically useful, the values in the Luby sequence are scaled by multiplying them with a scale-factor `s`.

A cutoff generator `c` for a Luby cutoff sequence with scale-factor `s` (of type **`unsigned long int`**) is created by:

```
Search::Cutoff* c = Search::Cutoff::luby(s);
```

Random. A random cutoff sequence consists of uniformly randomly chosen values between a lower bound `min` and an upper bound `max`. To focus on rather different values, only values from the set of $\mathtt{n}+1$ values:

$$\{\mathtt{min} + \lfloor i \cdot (\mathtt{max} - \mathtt{min})/\mathtt{n} \rfloor \mid i = 0, \ldots, \mathtt{n}\}$$

are chosen randomly.

A cutoff generator `c` for a random sequence with lower bound `min`, upper bound `max`, and number of values `n` (all of type **`unsigned long int`**) is created by:

```
Search::Cutoff* c = Search::Cutoff::rnd(seed,min,max,n);
```

where `seed` (of type **`unsigned int`**) defines the seed value for the random number generator used.

Constant. A constant cutoff sequence is defined by the scale-factor `s`. Then, the sequence consists of the cutoff values:

$$\mathtt{s}, \mathtt{s}, \mathtt{s}, \ldots$$

A cutoff generator `c` for a constant cutoff sequence with scale-factor `s` (of type **`unsigned long int`**) is created by:

```
Search::Cutoff* c = Search::Cutoff::constant(s);
```

Linear. A linear cutoff sequence is defined by the scale-factor `s`. Then, the sequence consists of the cutoff values:

$$1 \cdot \mathtt{s}, 2 \cdot \mathtt{s}, 3 \cdot \mathtt{s}, \ldots$$

A cutoff generator `c` for a linear cutoff sequence with scale-factor `s` (of type **`unsigned long int`**) is created by:

```
Search::Cutoff* c = Search::Cutoff::linear(s);
```

Append. An appended cutoff sequence `c` for `n` values from the cutoff sequence $\mathtt{c}_1$ (with values $k_0, k_1, k_2, \ldots$) followed by the values from the cutoff sequence $\mathtt{c}_2$ (with values $l_0, l_1, l_2, \ldots$) consists of the following values:

$$k_0, k_1, \ldots, k_{\mathtt{n}-2}, k_{\mathtt{n}-1}, l_0, l_1, l_2, \ldots$$

A cutoff generator `c` for an appended cutoff sequence with `n` (of type **`unsigned long int`**) values from cutoff generator `c1` followed by values from cutoff generator `c2` is created by:

```
Search::Cutoff* c = Search::Cutoff::append(c1,n,c2);
```

Merge. A merged cutoff sequence `c` for values from the cutoff sequence $\mathtt{c}_1$ (with values $k_0, k_1, k_2, \ldots$) merged with the values from the cutoff sequence $\mathtt{c}_2$ (with values $l_0, l_1, l_2, \ldots$) consists of the following values:

$$k_0, l_0, k_1, l_1, k_2, l_2, \ldots$$

A cutoff generator `c` for a merged cutoff sequence with values from cutoff generator `c1` merged with values from cutoff generator `c2` is created by:

```
Search::Cutoff* c = Search::Cutoff::merge(c1,c2);
```

Repeat. A repeated cutoff sequence `c` with repeat factor `n` (of type **`unsigned long int`**) for the cutoff sequence c' (with values $k_0, k_1, k_2, \ldots$) consists of the following values:

$$\underbrace{k_0, k_0, \ldots, k_0}_{\texttt{n times}}, \underbrace{k_1, k_1, \ldots, k_1}_{\texttt{n times}}, \underbrace{k_2, k_2, \ldots, k_2}_{\texttt{n times}}, \ldots$$

A cutoff generator `c` for a repeated cutoff sequence with repeat factor `n` (of type **`unsigned long int`**) and values from the cutoff generator `c1` is created by:

```
Search::Cutoff* c = Search::Cutoff::repeat(c1,n);
```

9.4.3 Computing a next solution

When the restart meta engine is asked for a next solution by calling `next()`, there are three possible scenarios:

- `next()` returns a pointer to a space (which might be `NULL` in case there are no more solutions or the engine has been stopped). Deleting the space is as with other engines the responsibility of the meta engine's user.

  By default, asking the meta engine for another solution will perform a restart. However, this behavior can be changed such that the current cutoff value (minus the failed nodes it took to find the current solution) is used for finding a next solution. For details, see Section 9.4.4.

- The meta engine reaches the current cutoff value. It restarts search with the next cutoff value.

- The meta engine is stopped by the stop object passed to it. Then `next()` returns `NULL` and `stopped()` returns **`true`** (to be able to distinguish this scenario from the one where there are no more solutions).

9.4.4 Master and slave configuration

The meta engine maintains a *master* space and each time the meta engine performs a restart, it passes a *slave* space (a clone of the master space) to the engine. Configuration is as follows: the master is configured, the slave is created as a clone of the master, and then the slave is configured. Initially, when the meta engine starts and creates a first slave space, it also configures the slave space.

More accurately, it leaves the actual configuration to the user: it calls the virtual member function `master()` on the master space and then calls the virtual member function `slave()` on the slave space. As mentioned above, the `slave()` function is also called the first time a slave is created. In that way, by redefining `master()` and `slave()` the user can control how master and slave are being configured (this is exactly the same idea how `constrain()` works for best solution search).

function	type	meaning
`restart()`	**`unsigned long int`**	number of restart
`solution()`	**`unsigned long int`**	number of solutions since last restart
`fail()`	**`unsigned long int`**	number of failures since last restart
`last()`	**`const`** `Space*`	last solution found (or `NULL`)
`nogoods()`	**`const`** `NoGoods&`	no-goods recorded from restart

Figure 9.9: Current restart information member functions

By default, every space implements the two member functions `master()` and `slave()`. Both functions take an argument of class `CRI` (for **c**urrent **r**estart **i**nformation) that contains information about the current restart. The class `CRI` provides the member functions as shown in Figure 9.9.

The default `slave()` function does nothing. The default `master()` function does the following:

- It calls the `constrain()` member function with the last solution found as argument (if a solution has already been found).
- It possibly posts no-goods as explained in Section 9.5.
- It returns **`true`** forcing a restart even if a solution has been found. Returning **`false`** instead would continue search without a restart.

For example, a class `Script` can define the member functions as follows:

```
class Script : public Space {
  ...
  virtual bool master(const CRI& cri) {
    // Configure the master
    ...
    // Whether to restart or not
    return true;
  }
  virtual void slave(const CRI& cri) {
    // Configure the slave
    ...
  }
};
```

The default `master()` and `slave()` member functions are shown in Figure 9.10.

```
virtual bool master(const CRI& cri) {
  if (cri.last() != NULL)
    constrain(*cri.last());
  cri.nogoods().post(*this);
  return true;
}
void Space::slave(const CRI&) {
}
```

Figure 9.10: Default `master()` and `slave()` functions

9.4.5 Large Neighborhood Search

The design of restart-based search in Gecode is general enough to support LNS (Large Neighborhood Search) [52]. The idea of LNS is quite simple, where LNS looks for a good solution:

- Search finds a first solution where typically preference is given to find a reasonably good solution quickly.

- During LNS, a new problem is generated that only keeps part of the so-far best solution (typically, the part to keep is randomly selected). This is often referred to as *relaxing* the so-far best solution. Then, search tries to find a next and better solution within a given cutoff.

  If the cutoff is reached without finding a solution, search can either decide to terminate or randomly retry again.

Figure 9.11 sketches a model that supports LNS. The constructor `Model()` initializes the model with all variables and constraints that are common for finding the first solution as well as for finding further solutions. Note that the model has a random number generator of class `Rnd` as member `r` to illustrate that typically some form of randomness is needed for restarting. However, in a real life problem additional data structures might be needed.

The `first()` function is responsible for posting additional constraints and branchings such that search can find the first solution. The `next()` function takes the so-far best solution `b` and posts additional constraints and branchings to find a next and better solution than `b`. This will typically mean that some but not all variables in the current space are assigned to values as defined by the so-far best solution `b` or that additional constraints are posted that depend on `b`.

MODEL SKETCH FOR LNS ≡

```
class Model : public Space {
protected:
  Rnd r;
public:
  Model(void) : ··· {
    // Initialize master
    ...
  }
  void first(void) {
    // Initialize slave for first solution
    ...
  }
  void next(const Model& b) {
    // Initialize slave for next solution
    ...
  }
  ...
  ▶ SLAVE FUNCTION
  virtual bool master(const CRI& cri) {
    ...
  }
};
```

Figure 9.11: Model sketch for LNS

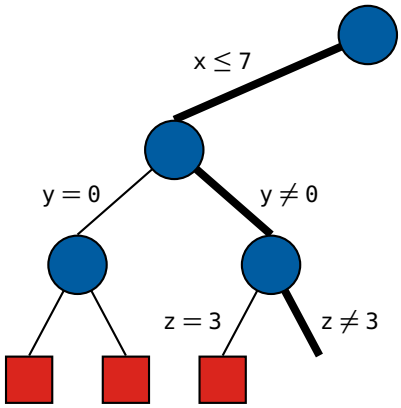


Figure 9.12: Search tree after cutoff 3 has been reached

Both `first()` and `best()` are executed on the slave space when the restart-based search engine performs a restart or executes search for the first time. This can be achieved by defining the `slave()` function as follows:

SLAVE FUNCTION ☰

```
virtual void master(slave CRI& cri) {
  if (cri.restart() == 0)
    first();
  else
    next(static_cast<const Model&>(*cri.last()));
}
```

The default `master()` function as shown in Section 9.4.4 is already general enough to support LNS.

9.5 No-goods from restarts

As discussed in the previous section, the idea of using restarts effectively is to restart with an improved problem with the hope that search is capable of solving the improved problem. No-goods are constraints that can be learned from the configuration of a depth-first search engine after it has stopped. The no-goods encode failures during search as constraints: after restarting, propagation of the no-goods ensures that decisions that lead to failure are avoided.

Consider the simple example depicted in Figure 9.12. It shows a search tree that has been explored by the restart-based search engine with a cutoff of 3 failures. The thick edges in the tree depict the path that is stored by a search engine using depth-first search (such as `DFS` or `BAB`).

What can be immediately derived from this configuration is that the following two constraints (they correspond to conjunctions of alternatives shown in Figure 9.12):

$$(x \leq 7) \wedge (y = 0) \qquad \text{and} \qquad (x \leq 7) \wedge (y \neq 0) \wedge (z = 3)$$

are *no-goods*: they cannot be satisfied by any solution of the problem (after all, search just proved that). The alternatives in the conjunction are also known as *no-good literals*.

When restarting search, the negation of the no-goods can be added to the master space as constraints and can be propagated. That is, the following constraints can be added:

$$\neg((x \leq 7) \wedge (y = 0)) \qquad \text{and} \qquad \neg((x \leq 7) \wedge (y \neq 0) \wedge (z = 3))$$

or, equivalently, the constraints:

$$\neg(x \leq 7) \vee \neg(y = 0) \qquad \text{and} \qquad \neg(x \leq 7) \vee \neg(y \neq 0) \vee \neg(z = 3)$$

Now assume that when restarting, after some search the constraint $x \leq 7$ becomes subsumed. Then, it can be propagated that $y \neq 0$. With other words, whenever $x \leq 7$ holds, then also $y \neq 0$ holds. This explains (if you know about resolution, you have figured out that one already anyway) that it is equivalent to use the following two constraints:

$$\neg(x \leq 7) \vee \neg(y = 0) \qquad \text{and} \qquad \neg(x \leq 7) \vee \neg(z = 3)$$

No-goods from restarts in Gecode follow the idea from [25], however the implementation differs. Moreover, the no-good literals used in Gecode can be arbitrary constraints and hence generalize the ideas from [21] and [25]. Also no-goods in Gecode support choices of arbitrary arity and are not restricted to binary choices. For more details, see Chapter 32 and in particular Section 32.2.

Generating and posting no-goods. When the restart-based search engine reaches the current cutoff limit or finds a solution it calls the `master()` member function as discussed in the previous section.

From the argument of class `CRI` that is passed to the `master()` function a no-good of class `NoGoods` can be retrieved by calling the `nogoods()` function. A no-good It has really only `post()` as its single important member function. The following `master()` function posts all constraints corresponding to the no-goods that can be derived from a restart (this is also the default `master()` function defined by the class `Space`, see also Section 9.4.4):

```
virtual bool master(const CRI& cri) {
  ...
  cri.nogoods().post(*this);
  ...
}
```

In order to post no-goods it must be enabled that a search engine maintains its internal state such that no-goods can be extracted. This is done by setting the no-goods depth limit (the member `nogoods_limit` of a search option of type **`unsigned int`**) to a value larger than zero. The value of `nogoods_limit` describes to which depth limit no-goods should be extracted from the path of the search tree maintained by the search engine. For example, the following code (it assumes that `c` refers to a cutoff object):

```
Search::Options o;
o.cutoff = c;
o.nogoods_limit = 128;
RBS<DFS,Script> e(s,o);
```

instructs the search engine to extract no-goods up to a depth limit of `128`.

The larger the depth limit, the more no-goods can of course be extracted. However, this comes at a cost:

- Gecode's search engines use an optimization that is called LAO (last alternative optimization, see also Section 40.4). LAO saves space by avoiding to store choices on the search engine's stack when the last alternative of a choice is being explored. But no-goods can only be extracted when LAO is disabled as the last alternatives are required for the computation of no-goods. LAO is automatically disabled for the part of the search tree with a depth less than the no-goods depth limit. That is, an engine requires more memory during search with a larger depth limit. This can pose an issue for very deep search trees.

  What also becomes clear from this discussion is that the peak search depth reported by a search engine increases with an increased depth limit.

- It is easy to see that no-goods get longer (with more literals) with increasing search tree depth. The larger a no-good gets, the less useful it gets: the likelihood that all literals but one are subsumed but not failed (which is required for propagation) decreases.

- When the no-goods are posted, a single propagator for all no-goods is created. This propagator requires $O(n)$ memory for n no-good literals. Hence, increasing the depth limit also increases the memory required by the propagator.

Tip 9.7 (Controlling no-goods with the commandline driver)**.** Whether no-goods are used and which no-goods depth limit is used can also be controlled from the commandline via the `-nogoods` and `-nogoods-limit` commandline options, see Chapter 11. ◀

No-goods from solutions restarts. The `master()` function shown above posts no-goods even when the restart meta search engine has found a solution. When the engine continues this means that the same solution might not be found again as it has been excluded by a no-good. The situation is even slightly more complicated: the solution might not be excluded if it has been found at a depth that exceeds the no-good depth limit.

If no-goods should not be posted when a solution has been found, the `master()` function can be redefined as:

```
virtual bool master(const CRI& cri) {
  ...
  if (cri.solution() == 0)
    cri.nogoods().post(*this);
  ...
}
```

Limitations. Not all branchers support the generation of no-goods. In that case the longest sequence of no-goods starting from the root of the search tree up to the first choice that belongs to a brancher that does not support no-goods is used.

All pre-defined branchers for integer, Boolean, and set variables support no-goods unless user-defined commit functions are used (see Section 8.8). Branchers for float variables and branchers for executing code (see Section 8.14) do not support no-goods.

No-goods and parallel search. The reason why after a restart no-goods can be extracted is because search computed the no-goods by exploring entire failed subtrees during search. This might not be true during parallel search. While parallel search engines also support the extraction of no-goods, the number of no-goods that can be extracted tend to be rather small.

10 Gist

The Graphical Interactive Search Tool, Gist, provides user-controlled search, search tree visualization, and inspection of arbitrary nodes in the search tree. Gist can be helpful when experimenting with different branching strategies, with different models for the same problem (for instance adding redundant constraints), or with propagation strength (for instance bounds versus domain propagation). It gives you direct feedback how the search tree looks like, if your branching heuristic works, or where propagation is weaker than you expected.

Overview. How the search tree of a problem is used as the central metaphor in Gist is sketched in Section 10.1. Section 10.2 explains how to invoke Gist, whereas Section 10.3 explains how to use Gist.

Important. Do not forget to add

```
#include <gecode/gist.hh>
```

to your program when you want to use Gist.

10.1 The search tree

The central metaphor in Gist is the *search tree*. Each node in the tree represents a fixpoint of propagatation (that is, an invocation of `status()`, see Tip 2.2). Inner nodes stand for

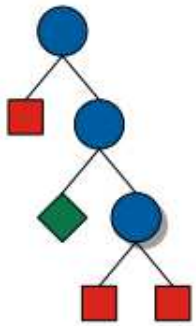


Figure 10.1: A search tree

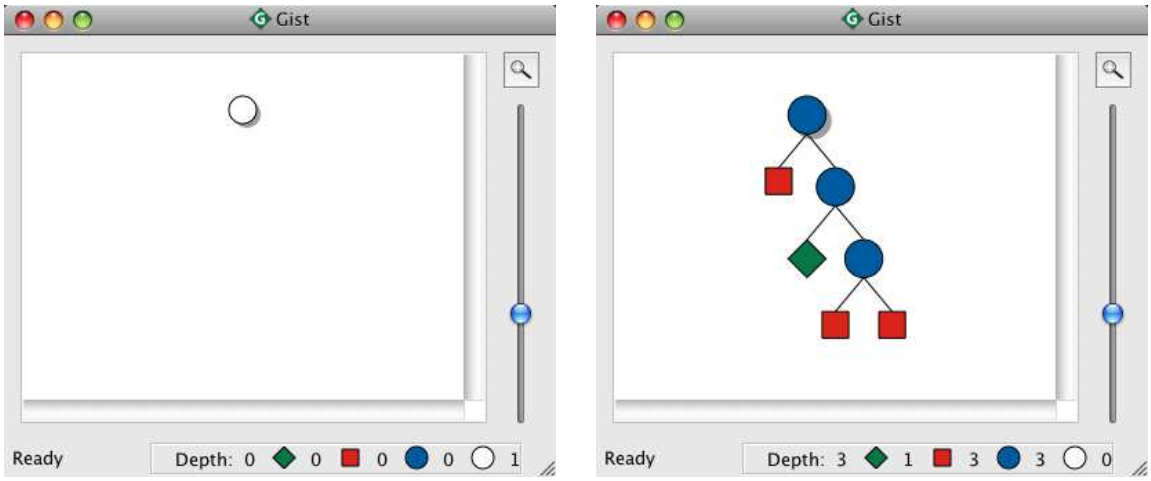


Figure 10.2: Gist, solving the Send More Money problem

choices (fixpoints with a subsequent brancher), while leaf nodes correspond to *failure* (dead ends in the search) or *solutions* of the problem. Figure 10.1 shows a search tree as drawn by Gist. The inner nodes (blue circles) are choices, the red square leaf nodes are failures, and the green diamond leaf node is a solution.

Conceptually, every node in the tree contains a corresponding space, which the user can access in order to inspect the node. Internally, Gist does not store all these spaces, but recomputes them on demand. That way, Gist can handle very large search trees (with millions of nodes) efficiently.

10.2 Invoking Gist

Gist is implemented using the Qt application framework. It can be invoked either as a standalone component, or as part of a bigger Qt-based application.

The screenshots in this chapter show Gist running on Mac OS X, but the functionality, the menus and the keyboard shortcuts are the same on all platforms (with small exceptions that will be mentioned later).

10.2.1 Standalone use

When used as a standalone component, invoking Gist merely amounts to calling

```
Gist::dfs(m);
```

where `m` is a pointer to the space that contains the model to be solved. This call opens a new instance of Gist, with the root node initialized with `m`. Figure 10.2 (left) shows Gist initialized with the Send More Money puzzle from Chapter 2.

If you want to solve an optimization problem using branch-and-bound search, you can invoke Gist with optimization turned on:

```
    Gist::bab(m);
```

10.2.2 Use as a Qt widget

If you are developing an application with a graphical user interface using the Qt toolkit, you can embed Gist as a widget. Either use `Gist::GistMainWindow`, which gives you an independent widget that inherits from `QMainWindow`, or directly embed the `Gist::Gist` widget into your own widgets. You have to include the files `gecode/gist/mainwindow.hh` for the independent widget, or `gecode/gist/qtgist.hh` for the widget you can embed.

Apart from the integration into your own application, the advantage over the standalone approach is that you get access to Gist's *signals* and *slots*. For example, you can use more than one inspector, or you can control the search programatically instead of by user input. The details of this are beyond the scope of this document, please refer to the reference documentation of the corresponding classes for more information, and have a look at the directory `gecode/gist/standalone-example` in the Gecode source distribution.

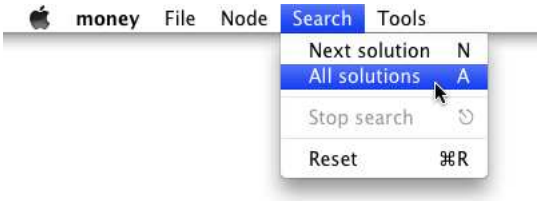
10.3 Using Gist

This section gives an overview of the functionality available in Gist. Most of Gist is intuitive to use, and the easiest way of learning how to use it is to start one of the examples that come with Gecode using the `-mode gist` commandline option, and then play around.

10.3.1 Automatic search

When you invoke Gist, it initializes the root node of the search tree, as seen in Figure 10.2. Any node that has not yet been explored by the search is drawn as a white circle, indicating that it is not yet known whether this node is in fact a choice, failure, or solution. White nodes are called *open*.

Obviously, the first thing a user will then want to do is to search for a solution of the given problem. Gist provides an automatic first-solution and all-solution depth-first search engine. It can be activated from the *Search* menu:



If you click *All solutions* (or alternatively press the *A* key), Gist will explore the entire tree. The result appears in Figure 10.2 (right). Depending on the preferences, Gist may collapse subtrees that contain only failed nodes (such as the rightmost subtree in the figure) into big red triangles.

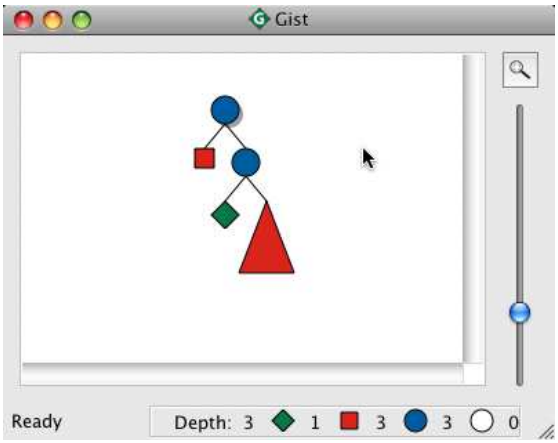


Figure 10.3: A hidden subtree in Gist

The search engines always only explore the subtree under the *currently selected node*, which is marked by a shadow. For example, the root node is selected after initialization, and the rightmost choice node is selected in Figure 10.2 (right). A single click selects a node, and there is always exactly one selected node.

Stopping search after exhausting a branching. If you want to learn more about how your branchings affect the shape of the search tree, you can add *stop branchings* to your problem using the `Gist::stopBranch()` function. This will install a brancher that does not modify the search tree (in fact, it simply inserts a single unary choice), but Gist will recognize the brancher and halt exploration. A special node (shaped like a stop-sign) marks the position in the tree where the stop brancher was active. You can toggle whether Gist continues exploration beyond the brancher using the options in the *Node* menu. Obviously, this is most useful if the call to `stopBranch` is placed *between* calls to other branchings.

10.3.2 Interactive search

As an alternative to automatic search, you can also explore the search tree interactively. Just select an arbitrary open (white) node and choose *Inspect* from the *Node-Inspect* menu (or press the *Return* key).

You can navigate in the tree using the arrow keys (or the corresponding options from the *Node* menu). Automatic and interactive search can be freely mixed. In order to focus on interesting parts of the search tree, you can hide subtrees using the *Hide/unhide* option, or hide all subtrees below the selected node that are completely failed. The option *Unhide all* expands all hidden subtrees below the current node again. The red triangle in Figure 10.3 is a hidden subtree.

If you want to start over, the *Search* menu provides an option to reset Gist.

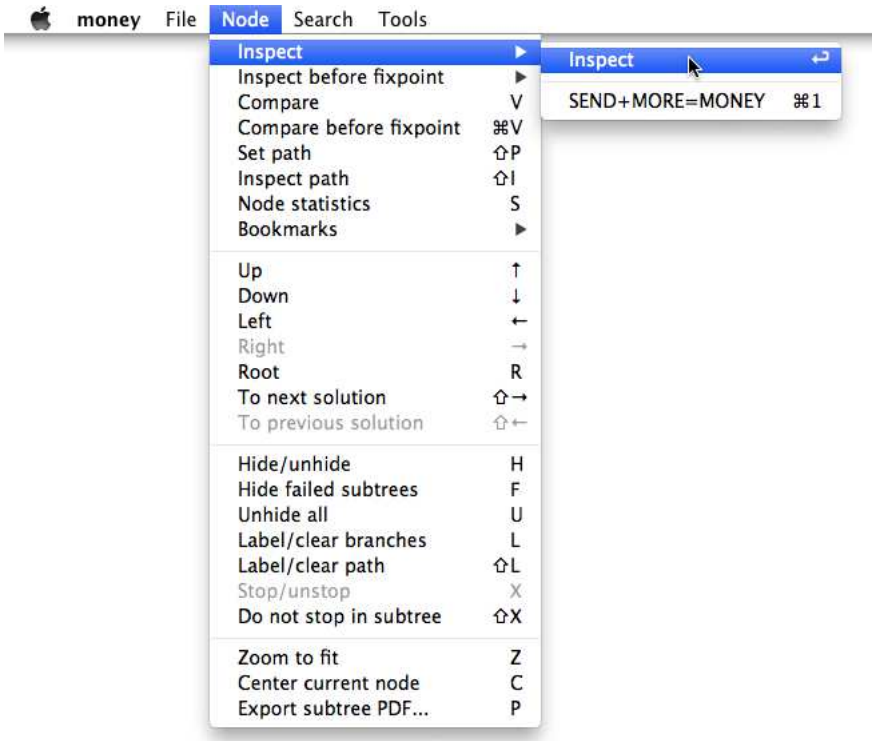


Figure 10.4: The *Node* menu

Bookmarks. The *Node* menu has a submenu *Bookmarks*, where you can collect nodes for quick access. You can set a bookmark for the currently selected node by chosing *Add/remove bookmark* from the submenue, or by pressing *Shift+B*. You can then enter a name for the bookmark (or leave it empty, then the bookmark will get a number). Chosing *Add/remove bookmark* for an already bookmarked node removes the bookmark. Bookmarked nodes are drawn with a small black circle. Selecting a bookmark moves you directly to the corresponding node.

10.3.3 Branch-and-bound search

If Gist has been invoked in branch-and-bound mode, it prunes the search tree with each new solution that is found. Branch-and-bound works with any order of exploration, so you can for instance explore interactively, or start automatic search for just a subtree.

The last solution that was found, which in branch-and-bound is always the best solution known so far, is displayed in orange instead of green.

10.3.4 Inspecting and comparing nodes

Of course just looking at the shape of the tree is most of the time not very enlightening by itself. We want to see the content of the nodes. For this, you can display information about

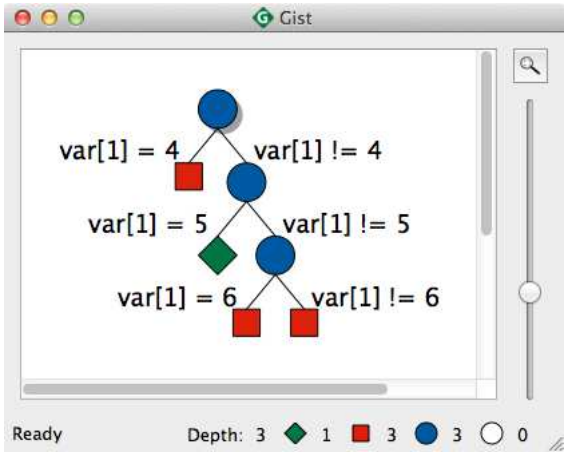


Figure 10.5: Branch information in Gist

the *alternatives* in the tree (provided by print functions, see Section 8.12), and add *inspectors* and *comparators* to Gist. Inspectors and comparators can be called on solution, choice, or failure nodes (but obviously not on open nodes).

Displaying branching information. The *Node* menu (Figure 10.4) has two options for displaying information about branches in the tree. Choosing *Label/clear branches* will add information on all branches in the subtree below the currently selected node (or clear that information if it is already present). *Label/clear path* adds that information on the path from the current node to the root. Figure 10.5 shows branching information for the Send More Money problem.

Gist uses print functions provided by the branchers of a space. For variable-value branchers as described in Chapter 8, the information displayed for the branches can be supplied by the user, see Section 8.12 for details. Also other branchers can define which information is printed for a branch, see Section 31.2.1.

Invoking inspectors and comparators. The *Node* menu (Figure 10.4) provides several options for inspecting and comparing nodes. If you choose *Inspect* from the *Inspect* submenu (or simply press *Return*), all double-click inspectors that are active in the *Tools* menu (see below) will be invoked for the currently selected node. You can also invoke a particular inspector by choosing it from th *Inspect* menu or typing its shortcut (the first nine inspectors get the shortcuts 0–9).

If you choose an inspector from the *Inspect before fixpoint* menu instead, the state of the node after branching but before fixpoint propagation is shown.

When choosing the *Compare* option, the mouse cursor turns into a crosshair, and clicking on another node will invoke the comparators that have been activated in the *Tools* menu. Again, *Compare before fixpoint* considers the state of the second node after branching but before fixpoint propagation. This is especially useful to find out what the branching does at

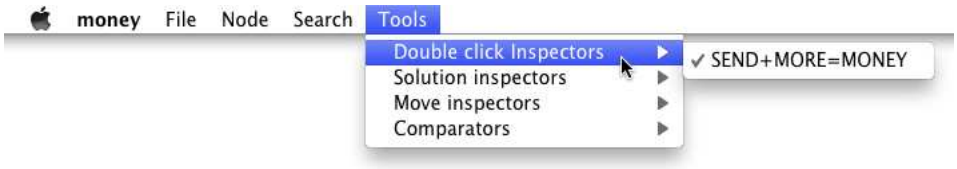


Figure 10.6: The *Tools* menu

a particular node: Select a node, choose *Compare before fixpoint*, and click on the child node you are interested in. The comparison will show you exactly how the branching has changed the variable domains.

Choosing the active inspectors and comparators. Gist distinguishes between three groups of inspectors, and the group of comparators, which can be chosen from the *Tools* menu (Figure 10.6):

- *Move inspectors* are called whenever the user moves to a different node.
- *Solution inspectors* are called for each new solution that is found.
- *Double-click inspectors* are called when the user explicitly inspects a node, for instance by choosing *Inspect* from the *Node* menu or by double-clicking.
- *Comparators* are called when the user chooses *Compare* or *Compare before fixpoint*, and then clicks on another node to compare the currently selected one to.

The printing inspector. The simplest way to add an inspector to your model is to use the `Gist::Print` inspector, as demonstrated in Section 2.4. Figure 10.7 shows the `Gist::Print` inspector after double-clicking the solution of the Send More Money problem.

Implementing inspectors. An inspector is an object that inherits from the abstract base class `Gist::Inspector`. The abstract base class declares a virtual member function, `inspect(const Space& s)`, which is called when one of the events described above happens. The space that is passed as an argument corresponds to the inspected node in the tree. The inspector is free to perform any **`const`** operation on the space.

The variable comparator. Similar to the printing inspector, there is a predefined comparator `Gist::VarComparator` that can be easily added to scripts. It requires the script to implement a member function `compare()` in which it outputs the result of comparing itself to another space. Figure 10.8 shows how to add comparison to the example from Section 2.4. It uses a convenience member function from the class `Gist::Comparator` that produces a string representation of the differences between two variable arrays.

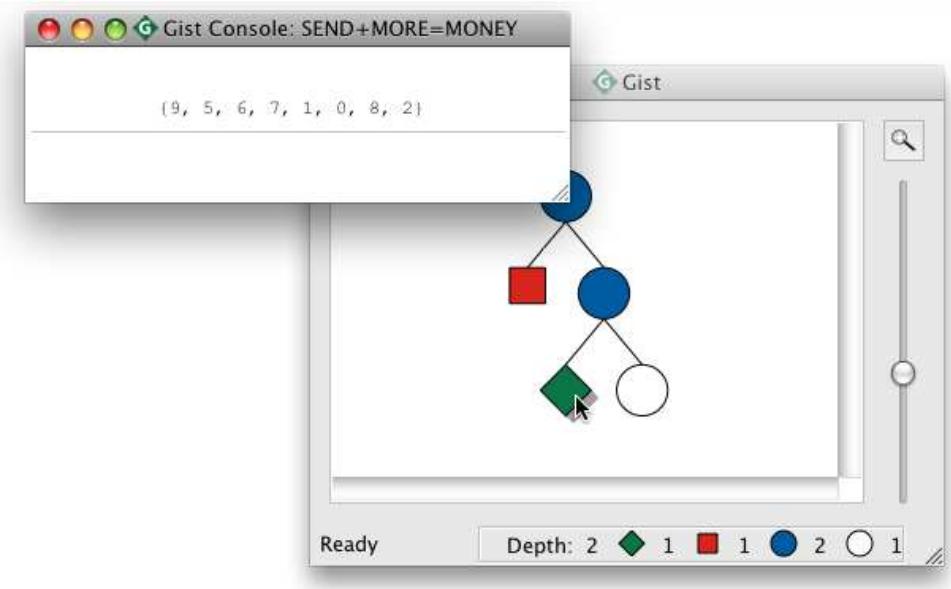


Figure 10.7: Inspecting a solution in Gist

Implementing comparators. A comparator inherits from `Gist::Comparator` and implements at least its `compare(const Space& s0, const Space& s1)` member function. This function is called when the currently selected node with space `s0` is compared to the node with space `s1`. As for inspectors, a comparator can perform any `const` operation on these two spaces.

Subtree statistics. The *Node* menu provides an option *Node statistics*, which when clicked opens a small window that displays statistics of the subtree rooted at the currently selected node, as shown in Figure 10.9. The statistics include the depth of the current node (the upper right number in Figure 10.9), the maximum depth of the subtree (the lower right number), and how many of the different node types the subtree contains (the numbers at the small nodes). The information is automatically updated when you select a different node.

10.3.5 Zooming, centering, exporting, printing

Here is some functionality you may have already found during your experiments with Gist.

Mouse wheel zoom. You probably noticed that the slider right of the search tree lets you zoom in and out. Another way of zooming is to hold *Shift* while using the mouse wheel. It will zoom in and out keeping the area under the mouse cursor visible.

SEND MORE MONEY WITH GIST COMPARISON ≡ [DOWNLOAD]

```
#include <gecode/int.hh>
#include <gecode/gist.hh>
...
class SendMoreMoney : public Space {
  ...
  void compare(const Space& s, std::ostream& os) const {
    os << Gist::Comparator::compare<IntVar>("l",l,
      static_cast<const SendMoreMoney&>(s).l);
  }

};

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  Gist::Print<SendMoreMoney> p("Print solution");
  Gist::VarComparator<SendMoreMoney> c("Compare nodes");
  Gist::Options o;
  o.inspect.click(&p);
  o.inspect.compare(&c);
  Gist::dfs(m,o);
  delete m;
  return 0;
}
```

Figure 10.8: Using Gist for Send More Money with node comparison

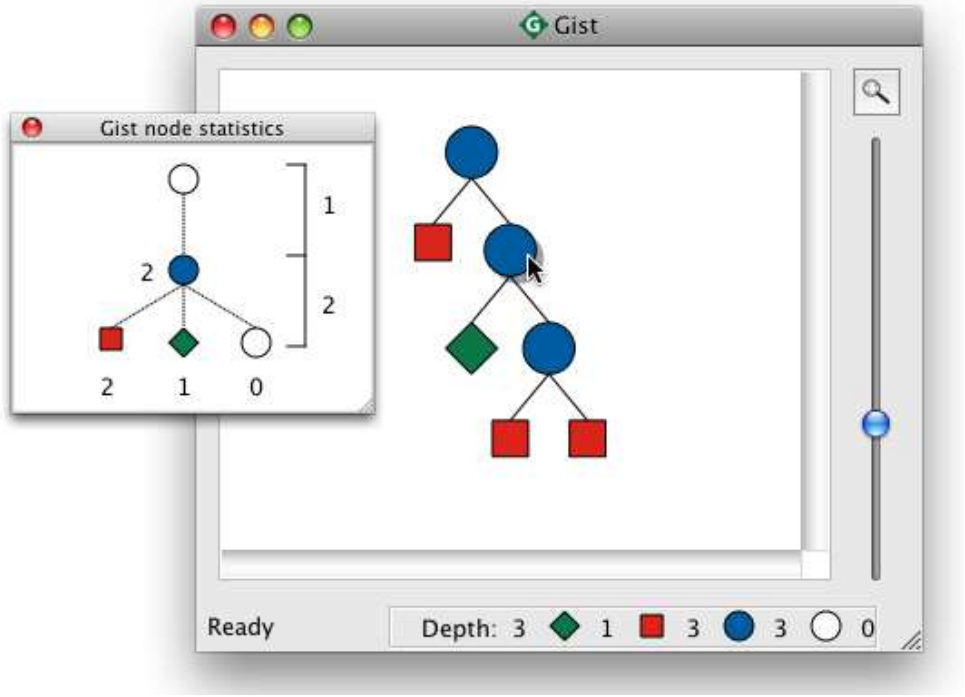


Figure 10.9: Node statistics

Zoom and center. In addition to manual zooming, Gist provides automatic options. The button with the magnifying glass icon above the zoom slider toggles the auto-zoom feature, which always zooms the tree such that as much of it as possible is visible in the window. During auto-zoom, the manual zoom is disabled. Instead of auto-zoom, you can also select *Zoom to fit* from the *Node* menu (or press *Z*) in order to adjust the zoom so that the current tree fits. When working with large trees, it is sometimes useful to scroll back to the currently selected node by choosing *Center current node* from the *Node* menu or pressing *C*.

Exporting and printing. The *File* menu provides options for exporting the current search tree as a PDF file or printing it. If you want to export a single subtree, select *Export subtree PDF* from the *Node* menu. The tree is exported or printed as seen, including hidden nodes.

10.3.6 Options and preferences

When invoking Gist, you can pass it an optional argument of type `Gist::Options`. This options class inherits from the standard search options discussed in Section 9.3.1, but adds a **`class`** with two member functions, `inspect.click()` and `inspect.solution()`, that you can use to pass inspectors to Gist.

The two options for recomputation, `c_d` and `a_d`, as well as the `clone` option of `Search::Options` are honored by Gist; the remaining options are ignored.

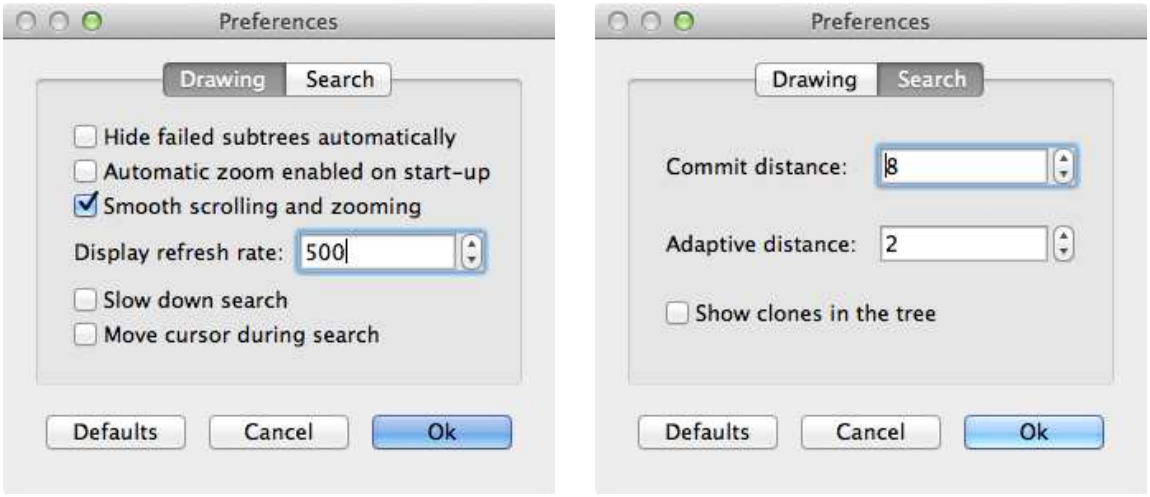


Figure 10.10: Gist preferences

During execution, Gist can be configured using the *Preferences* dialog, available from the program menu on Mac OS or the *File* menu on Windows and Linux.

The drawing preferences (Figure 10.10, left) let you specify whether failed subtrees are hidden automatically during search, and whether the auto-zoom and smooth scrolling features are enabled at start-up. Furthermore, you can set the refresh interval – this is the number of nodes that are explored before the tree is redrawn. If you set this to a large number, search will be faster, but you get less visual feedback. You can enable slow search, which will explore only around three nodes per second, useful for demonstrating how search proceeds. Finally, enabling the "Move cursor during search" option means that the move inspectors will be called for every single node during search. Again, this is great for demonstration and debugging purposes, but it slows down the search considerably. Drawing preferences (except for the slow-down and cursor moving options) are remembered between sessions and across different invocations of Gist.

The search preferences are exactly the parameters you can pass using the `Gist::Options` class. In addition, you can switch on the display of where Gist actually stores spaces in the tree, as shown in Figure 10.11. A small red circle indicates a clone used for recomputation, while a small yellow circle shows that the node has an active space used for exploration (a so-called *working clone*).

The recomputation parameters are not remembered between sessions.

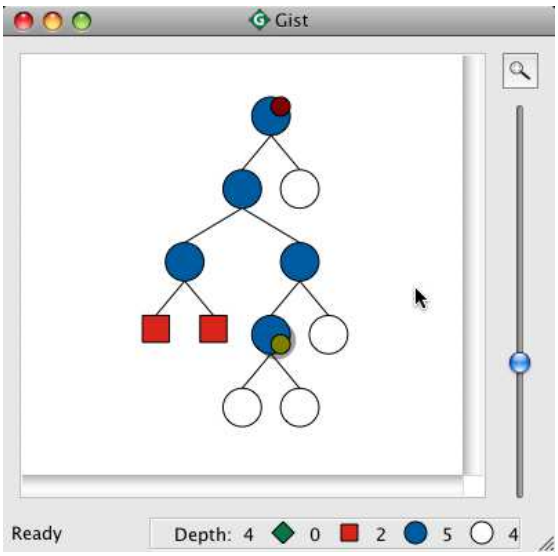


Figure 10.11: Displaying where Gist stores spaces in the tree

11 Script commandline driver

The commandline driver (see Script commandline driver) provides support for passing common commandline options to programs and a sub-class for spaces called `Script` that can take advantage of the options.

Overview. Section 11.1 summarizes the commandline options supported by the commandline driver. The base classes for scripts that work together with the commandline driver are sketched in Section 11.2.

Important. Do not forget to add

```
#include <gecode/driver.hh>
```

to your program when you want to use the script commandline driver.

11.1 Commandline options

The commandline driver provides classes `Options`, `SizeOptions`, and `InstanceOptions` that support parsing commandline options. All classes support the options as summarized in Figure 11.1 and Figure 11.2. Here, for a commandline option with name `-name`, the option classes provide two functions with name `name()`: one that takes no argument and returns the current value of the option, and one that takes an argument of the listed type and sets the option to that value. If the commandline options contains a hyphen `-`, then the member function contain an underscore `_` instead. For example, for the options `-c-d` and `-a-d` the member functions are named `c_d()` and `a_d()`.

The values for `-threads` are interpreted as described in Section 9.3.1.

The only option for which no value exists is `-help`: it prints some configuration information and a help text for the options and stops program execution.

The class `SizeOptions` accepts an unsigned integer as the last value on the commandline (of course, without an option). The value can be retrieved or set by member functions `size()`.

The class `InstanceOptions` accepts a string as the last value on the commandline (of course, without an option). The value can be retrieved or set by member functions `instance()`.

option	type	explanation
propagation options		
`-icl`	{`def`, `val`, `bnd`, `dom`}	integer consistency level
branching options		
`-decay`	**`double`**	decay-factor
search options		
`-solutions`	**`unsigned int`**	how many solutions (0 for all)
`-threads`	**`double`**	how many threads
`-c-d`	**`unsigned int`**	commit recomputation distance
`-a-d`	**`unsigned int`**	adaptive recomputation distance
`-node`	**`unsigned int`**	cutoff for number of nodes
`-fail`	**`unsigned int`**	cutoff for number of failures
`-time`	**`unsigned int`**	cutoff for time in milliseconds
restart-based search options		
`-restart`	{`none`, `geometric`, `luby`}	enable restarts, define cutoff
`-restart-scale`	**`unsigned int`**	scale-factor for cutoff values
`-restart-base`	**`double`**	base for geometric cutoff values
`-nogoods`	{`false`, `true`, `0`, `1`}	whether to post no-goods
`-nogoods-limit`	**`unsigned int`**	depth limit for no-good recording
execution options		
`-mode`	{`solution`, `time`, `stat`, `gist`}	script mode to run
`-samples`	**`unsigned int`**	how many samples
`-iterations`	**`unsigned int`**	how many iterations per sample
`-print-last`	{`false`, `true`, `0`, `1`}	whether to only print last solution
`-file-sol`	{`stdout`, `stdlog`, `stderr`}	where to print solutions
`-file-stat`	{`stdout`, `stdlog`, `stderr`}	where to print statistics
`-interrupt`	{`false`, `true`, `0`, `1`}	whether driver catches Ctrl-C

Figure 11.1: Predefined commandline options

option	type	explanation
`-branching`	string	branching options
`-model`	string	general model options
`-propagation`	string	propagation options
`-symmetry`	string	symmetry breaking options
`-search`	string	search options

Figure 11.2: User-definable commandline options

The different modes passed as argument for the option `-mode` have the following meaning:

- `solution` prints solutions together with some runtime statistics.
- `time` can be used for benchmarking: average runtime and coefficient of deviation is printed, where the example is run `-samples` times. For examples with short runtime, `-iterations` can be used to repeat the example several times before measuring runtime. Note that the runtime includes also setup time (the creation of the space for the model).
- `stat` prints short execution statistics.
- `gist` runs Gist rather than search engines that print information. Gist is put into depth-first mode, when a non best solution search engine is used (that is, `DFS`), and into branch-and-bound mode otherwise (that is, `BAB`).

  If Gecode has not been compiled with support for Gist (see Tip 3.3 for how to find out about Gecode's configuration), the mode `gist` will be ignored and the mode `solution` is used instead.

For an example, in particular, how to use the user-defined options, see Section 3.3. As all examples that come with Gecode use the script commandline driver, a plethora of examples is available (see Example scripts (models)). Also adding additional options is straightforward, for an example see Golf tournament.

Gist inspectors and comparators. The driver options can pass inspectors and comparators (see Section 10.3.4) to Gist. To register an inspector `i`, use the `inspect.click(&i)`, `inspect.solution(&i)`, or `inspect.move(&i)` methods of the option object, for a comparator `c`, use `inspect.compare(&c)`.

11.2 Scripts

Scripts (see Script classes) are subclasses of `Space` that are designed to work together with option objects of class `Options` and `SizeOptions`.

In particular, the driver module defines scripts `MinimizeScript` and `MaximizeScript` that can be used for finding best solutions based on a virtual `cost()` function, see also Section 3.2 and Section 7.3.

As scripts are absolutely straightforward, all can be understood by following some examples. For an example see Section 3.3 or all examples that come with Gecode, see Example scripts (models).

C

Case studies

Christian Schulte, Guido Tack, Mikael Z. Lagerkvist

This part presents a collection of modeling case studies. The case studies are ordered (roughly) according to their complexity.

Basic models. The basic models use classic constraint programming problems in order to demonstrate how typical modeling tasks are done with Gecode. The basic models are:

- Chapter 12 (Golomb rulers) shows a simple problem with a single `distinct` constraint and a few `rel` constraints. As the problem is so well known, it might serve as an initial case study of how to model with Gecode.
- Chapter 13 (Magic sequence) shows how to use counting constraints (`count`).
- Chapter 14 (Photo alignment) shows how to use reified constraints for solving an over-constrained problem.
- Chapter 15 (Locating warehouses) shows how to use `element`, `linear`, and global counting (`count`) constraints.
- Chapter 16 (Nonogram) shows how to use regular expressions and extensional constraints.
- Chapter 17 (Social golfers) presents a case study on modeling problems using set variables and constraints.

Advanced models. The following models are slightly more advanced:

- Chapter 18 (Knight's tour) demonstrates a problem-specific brancher inspired by a classic heuristic for a classic problem.
- Chapter 19 (Bin packing) also demonstrates a problem-specific brancher including techniques for breaking symmetries during branching.
- Chapter 20 (Kakuro) presents a model that employs user-defined constraints implemented as `extensional` constraints using tuple set specifications. Interestingly, the tuple set specifications are computed by solving a simple constraint problem.
- Chapter 21 (Crossword puzzle) presents a simple model using nothing but `distinct` and `element` constraints. The simple model is shown to work quite well compared to a dedicated problem-specific constraint solver. This underlines that an efficient general-purpose constraint programming system actually can go a long way.

Note that the first two case studies require knowledge on programming branchers, see Part B.

Acknowledgments. We thank Pierre Flener and Håkan Kjellerstrand for numerous detailed and helpful comments on the case studies. Their comments have considerably improved the presentation of the case studies.

12 Golomb rulers

This chapter studies a simple problem that is commonly used as an example for constraint programming. The model uses nothing but a single `distinct` constraint, a few `rel` constraints, and posts linear expressions. As the problem is so well known, it might serve as an initial case study of how to model with Gecode.

12.1 Problem

The problem is to find an optimal *Golomb ruler* (see Problem 6 in CSPLib) of size n. A Golomb ruler has n marks $0 = \mathtt{m}_0 < \mathtt{m}_1 < \cdots < \mathtt{m}_{n-1}$ such that the distances $\mathtt{d}_{i,j} = \mathtt{m}_j - \mathtt{m}_i$ for $0 \leq i < j < n$ are pairwise distinct. An optimal Golomb ruler is of minimal length (that is, $\mathtt{m}_{n-1}$ is minimal). Figure 12.1 shows an optimal Golomb ruler with 6 marks.

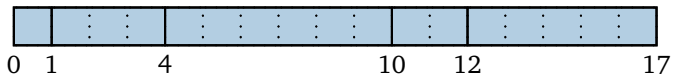


Figure 12.1: An optimal Golomb ruler with 6 marks

In the model for Golomb rulers, we are going to use the following construction for a Golomb ruler (a non-optimal ruler, though) as it provides upper bounds on the values for the marks of a ruler. The upper bounds improve the efficiency of our model, see below for more details.

Assume that the distance between marks i and $i+1$ is $m_{i+1} - m_i = 2^{i+1}$ (that is, for example, $m_1 - m_0 = 1$, $m_2 - m_1 = 2$, $m_3 - m_2 = 4$, and so on). Then the marks are

$$m_i = \sum_{k=1}^{i} 2^{k-1} = 2^i - 1.$$

Figure 12.2 shows a Golomb ruler with 6 marks following this construction.

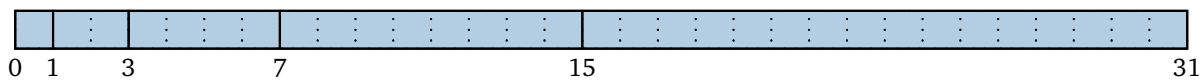


Figure 12.2: A constructed Golomb ruler with 6 marks

GOLOMB ≡ [DOWNLOAD]

```
...
class GolombRuler : public MinimizeScript {
protected:
  IntVarArray m;
public:
  GolombRuler(const SizeOptions& opt)
    : m(*this,opt.size(),0,
        (opt.size() < 31)
          ? (1 << (opt.size()-1)) - 1
          : Int::Limits::max) {
    ▶ CONSTRAINING MARKS
    ▶ NUMBER OF MARKS AND DISTANCES
    ▶ POSTING DISTANCE CONSTRAINTS
    ▶ IMPLIED CONSTRAINTS
    ▶ DISTANCES MUST BE DISTINCT
    ▶ SYMMETRY BREAKING
    ▶ BRANCHING
  }
  virtual IntVar cost(void) const {
    return m[m.size()-1];
  }
  ...
};
...
```

Figure 12.3: A script for computing Golomb rulers

Now consider the bit representation of m_i: exactly the least i bits are one. For the distances

$$d_{i,j} = m_j - m_i = \sum_{k=1}^{j} 2^{k-1} - \sum_{k=1}^{i} 2^{k-1}$$

we can easily see that in their bit representation the least i bits are zero, followed by $j - i$ ones. That means for $0 \leq i < j < n$ the bit representations of the $d_{i,j}$ are pairwise distinct. In other words, we can always construct a Golomb ruler with n marks of length $m_{n-1} = 2^{n-1} - 1$.

12.2 Model

Figure 12.3 shows the script for implementing the Golomb ruler model. The script stores a variable array `m` for the marks. The largest possible value of a mark is set to $2^{n-1} - 1$

according to the construction of a Golomb ruler in the previous section, provided that this value does not exceed the possible size limit of an integer (integers in Gecode are at least 32 bits, this is checked when Gecode is configured for compilation). If $n \geq 31$ we just choose the largest possible integer value for an integer variable (see `Int::Limits`).

Tip 12.1 (Small variable domains are still beautiful)**.** As mentioned in Tip 4.3, initializing variable domains to be small makes sense. For example, if we always chose `Int::Limits::max` rather than the smaller upper bounds for Golomb rulers with $n < 31$, the propagators for `linear` constraining the distances would have to resort to extended precision as the internal computations during propagation exceed the integer precision. That would mean that scripts for $n < 31$ would run approximately 15% slower! ◀

The script does not store a variable array for the distances (unlike the array for the marks), they are stored in an integer variable argument array. As the distances are only needed for posting constraints but not for printing the solution, it is more efficient to store them in an variable argument array but not in a variable array. More details on argument arrays and their relation to variable arrays can be found in Section 4.2.

The `cost()` function as required by the class `MinimizeScript` (see Section 11.2) just returns the largest mark on the ruler.

Marks. Assigning the first mark to zero and ordering the marks in increasing order is done by posting `rel` constraints (see Section 4.4.3):

CONSTRAINING MARKS ≡
```
rel(*this, m[0], IRT_EQ, 0);
rel(*this, m, IRT_LE);
```

Distances. The number of marks `n` and number of distances `n_d` are initialized so that they can be used for posting constraints:

NUMBER OF MARKS AND DISTANCES ≡
```
const int n = m.size();
const int n_d = (n*n-n)/2;
```

As mentioned, the distances are stored in an integer variable argument array `d`. The fields of the array `d` are initialized by the variable returned by the `expr()` function for linear expressions (see Section 7.1):

POSTING DISTANCE CONSTRAINTS ≡
```
IntVarArgs d(n_d);
for (int k=0, i=0; i<n-1; i++)
  for (int j=i+1; j<n; j++, k++)
    d[k] = expr(*this, m[j] - m[i]);
```

One might be tempted to optimize the posting of distance constraints for $\mathtt{d}_{0,j}$ for $0 < j < n$ as $\mathtt{m}_0 = 0$ and hence $\mathtt{d}_{0,j} = \mathtt{m}_j$ for $0 < j < n$. Optimizing avoids to create new variables (that is, the variables $\mathtt{m}_j$ are stored as $\mathtt{d}_{0,j}$ for $0 < j < n$) and posting propagators to implement the equality constraints $\mathtt{d}_{0,j} = \mathtt{m}_j$ for $0 < j < n$.

However, the `expr()` function does this automatically. As $\mathtt{m}_0$ is already assigned by posting a `rel` constraint, the `expr()` function simplifies the posted expressions accordingly.

Finally, all distances must be pairwise distinct (see Section 4.4.7) where bounds propagation is requested (see Section 4.3.5):

DISTANCES MUST BE DISTINCT ≡
```
distinct(*this, d, ICL_BND);
```

Intuitively, bounds propagation is sufficient as also the propagation for the distances is using bounds propagation.

Implied constraints. The following implied constraints are due to [55]. A distance $\mathtt{d}_{i,j}$ for $0 \leq i < j < n$ satisfies the property that it is equal to the sum of all distances between marks $\mathtt{m}_i$ and $\mathtt{m}_j$. That is

$$\mathtt{d}_{i,j} = \mathtt{d}_{i,i+1} + \mathtt{d}_{i+1,i+2} + \cdots + \mathtt{d}_{j-1,j}$$

This can be verified as follows:

$$\begin{aligned}
\mathtt{d}_{i,j} &= \mathtt{m}_j - \mathtt{m}_i \\
&= (\mathtt{m}_j - \mathtt{m}_{j-1}) + (\mathtt{m}_{j-1} - \mathtt{m}_{j-2}) + \cdots + (\mathtt{m}_{i+1} - \mathtt{m}_i) \\
&= \mathtt{d}_{j-1,j} + \mathtt{d}_{j-2,j-1} + \cdots + \mathtt{d}_{i,i+1} \\
&= \mathtt{d}_{i,i+1} + \mathtt{d}_{i+1,i+2} + \cdots + \mathtt{d}_{j-1,j}
\end{aligned}$$

As all distances $\mathtt{d}_{i,j}$ for $0 \leq i < j < n$ must be pairwise distinct, also the $j - i$ distances

$$\mathtt{d}_{i,i+1}, \mathtt{d}_{i+1,i+2}, \ldots, \mathtt{d}_{j-1,j}$$

must be pairwise distinct and hence must be $j - i$ distinct integers. That means that

$$\mathtt{d}_{i,j} = \mathtt{d}_{i,i+1} + \mathtt{d}_{i+1,i+2} + \cdots + \mathtt{d}_{j-1,j}$$

must be at least the sum of the first $j - i$ integers:

$$\mathtt{d}_{i,j} \geq \sum_{l=1}^{j-i} l = (j-i)(j-i+1)/2$$

The implied constraints can be posted as a lower bound with a `rel` constraint (see Section 4.4.3) for the distances as follows:

IMPLIED CONSTRAINTS ≡
```
for (int k=0, i=0; i<n-1; i++)
  for (int j=i+1; j<n; j++, k++)
    rel(*this, d[k], IRT_GQ, (j-i)*(j-i+1)/2);
```

Note that one could also combine the posting of the distance constraints with constraining the lower bounds of the distances for efficiency. However, we separate both for clarity. Anyway, the time spent on posting constraints is insignificant to the time spent on solving the model!

Symmetry breaking. Provided that the ruler has a sufficient number of marks (that is, $n > 2$) we can break (a few) symmetries by constraining the distance $d_{0,1}$ (stored at the first position in the array `d`) between the first and second mark to be smaller than the distance $d_{n-2,n-1}$ (stored at the last position in the array `d`) between the next to last and last mark as follows:

```
SYMMETRY BREAKING ≡
  if (n > 2)
    rel(*this, d[0], IRT_LE, d[n_d-1]);
```

Branching. The branching chooses the marks from left to right on the ruler and assigns the smallest possible value for a mark first:

```
BRANCHING ≡
  branch(*this, m, INT_VAR_NONE(), INT_VAL_MIN());
```

12.3 More information

This case study is also available as an example, see Finding optimal Golomb rulers. For a detailed discussion of how to model the Golomb ruler problem, see [55].

13 Magic sequence

This chapter shows how to use counting constraints for solving magic sequence puzzles.

13.1 Problem

The Magic Sequence puzzle (Problem 19 in CSPLib, first introduced as a constraint problem in [59]) requires finding a sequence of integers $x_0, \ldots, x_{n-1}$ such that for all $0 \leq i < n$, the number i occurs exactly x_i times in the sequence. For example, a magic sequence of length $n = 8$ is

$$\langle 4, 2, 1, 0, 1, 0, 0, 0 \rangle$$

13.2 Model

The outline of the script for solving magic sequence puzzles is shown in Figure 13.1. It contains the integer variable array x (see Section 4.2.1), which is initialized according to the problem specification.

Counting constraints. The problem can be modeled directly using one counting constraint (see Section 4.4.8) per variable. We will see later that there is a global constraint that combines all these individual counting constraints.

COUNTING CONSTRAINTS ≡
```
for (int i=0; i<x.size(); i++)
  count(*this, x, i, IRT_EQ, x[i]);
```

Implied linear constraints. The model as described so far completely captures the problem. Therefore, given enough time, Gecode will return all its solutions and only the solutions. However, it is sometimes useful to post additional, *implied constraints*, which do not change the meaning of the model (they do not change the set of solutions), but which provide additional constraint propagation that results in a smaller search tree.

MAGIC SEQUENCE ≡ [DOWNLOAD]

```
...
class MagicSequence : public Script {
  IntVarArray x;
public:
  MagicSequence(const SizeOptions& opt)
    : x(*this,opt.size(),0,opt.size()-1) {
    ▶ COUNTING CONSTRAINTS
    ▶ IMPLIED CONSTRAINTS
    ▶ BRANCHING
  }
  ...
};
...
```

Figure 13.1: A script for solving magic sequence puzzles

The two implied constraints that we will use for the magic sequence problem result from the fact that any magic sequence of length n satisfies the following two equations:

$$\sum_{i=0}^{n-1} \mathsf{x}_i = n \qquad\qquad \sum_{i=0}^{n-1} (i-1) \cdot \mathsf{x}_i = 0$$

The first equation is true because the sum of all occurrences, i.e. the overall number of items in the sequence, must be equal to the length of the sequence.

The second equation can be rewritten as

$$\sum_{i=0}^{n-1} i \cdot \mathsf{x}_i = \sum_{i=0}^{n-1} \mathsf{x}_i \iff \sum_{i=0}^{n-1} i \cdot \mathsf{x}_i = n$$

So it remains to be shown that $\sum_{i=0}^{n-1} i \cdot \mathsf{x}_i$ is also equal to the length of the sequence. This follows from the fact that x_i is the number of times i occurs in the sequence, so $i \cdot \mathsf{x}_i$ is the number of positions occupied by the sequence elements that are equal to i, and the sum over those must be equal to the length of the sequence.

The two equations translate easily into linear constraints (see Section 4.4.6), using integer argument arrays of type `IntArgs` (see Section 4.2.2) to supply coefficients.

IMPLIED CONSTRAINTS ≡

```
linear(*this, x, IRT_EQ, x.size());
linear(*this, IntArgs::create(x.size(),-1,1), x, IRT_EQ, 0);
```

You can do your own experiments, comparing runtime and search tree size of the model with and without implied constraints.

MAGIC SEQUENCE GCC ≡ [DOWNLOAD]

```
...
class MagicSequence : public Script {
  IntVarArray x;
public:
  MagicSequence(const SizeOptions& opt)
    : x(*this,opt.size(),0,opt.size()-1) {
    // Global counting constraint
    count(*this, x, x, opt.icl());
    // Implied constraint
    linear(*this, IntArgs::create(x.size(),-1,1), x, IRT_EQ, 0);
    // Branching
    branch(*this, x, INT_VAR_NONE(), INT_VAL_MAX());
  }
  ...
};
...
```

Figure 13.2: Magic sequence puzzles with a global counting constraint

Branching. For large sequences, many variables in the sequence will be 0 because the overall sum is only n (see previous paragraph on implied constraints). Therefore, x_0 should take a large value. We simply branch in the given order of the variables, starting with the largest values.

BRANCHING ≡

```
branch(*this, x, INT_VAR_NONE(), INT_VAL_MAX());
```

Global counting constraints. The global counting constraint (also known as global cardinality constraint, see Section 4.4.8) can express the combination of all the individual counting constraints, and yields stronger propagation. It also includes the propagation of the first of the two implied linear constraints. So, as an alternative to the n counting constraints above, we can use the code in Figure 13.2.

13.3 More information

The magic sequence puzzle is also included as a Gecode example, see Magic sequence. The example contains both the model using individual counting constraints and the one using a single global counting constraint.

14 Photo alignment

This chapter shows how to use reified constraints for solving an overconstrained problem.

14.1 Problem

Betty, Chris, Donald, Fred, Gary, Mary, Paul, Peter, and Susan want to align in a row for taking a photo. They have the following preferences:

1. Betty wants to stand next to Donald, Gary, and Peter.
2. Chris wants to stand next to Gary and Susan.
3. Donald wants to stand next to Fred and Gary.
4. Fred wants to stand next to Betty and Gary.
5. Gary wants to stand next to Mary and Betty.
6. Mary wants to stand next to Betty and Susan.
7. Paul wants to stand next to Donald and Peter.
8. Peter wants to stand next to Susan and Paul.

These preferences are obviously not satisfiable all at once (e.g., Betty cannot possibly stand next to three people at once). The problem is *overconstrained*. To solve an overconstrained problem, we turn it into an optimization problem: The task is to find an alignment that violates as few preferences as possible.

14.2 Model

We model the photo alignment as an array of integer variables `pos` such that `pos[p]` represents the position of person `p` in the final left-to-right order. The outline of a script for this problem is shown in Figure 14.1.

The `cost()` function as required by the class `MinimizeScript` (see Section 11.2) just returns the number of violations.

PHOTO ≡ [DOWNLOAD]

```
...
enum {
  Betty, Chris, Donald, Fred, Gary,
  Mary, Paul, Peter, Susan
};
const int n = 9;
const int n_prefs = 17;
int spec[n_prefs][2] = {
  {Betty,Donald}, {Betty,Gary}, {Betty,Peter},
  {Chris,Gary}, {Chris,Susan},
  {Donald,Fred}, {Donald,Gary},
  {Fred,Betty}, {Fred,Gary},
  {Gary,Mary}, {Gary,Betty},
  {Mary,Betty}, {Mary,Susan},
  {Paul,Donald}, {Paul,Peter},
  {Peter,Susan}, {Peter,Paul}
};

class Photo : public MinimizeScript {
  IntVarArray pos;
  IntVar      violations;
public:
  Photo(const Options& opt)
    : pos(*this,n,0,n-1), violations(*this,0,n_prefs) {
    ▶ CONSTRAIN POSITIONS
    ▶ COMPUTE VIOLATIONS
    ▶ SYMMETRY BREAKING
    ...
  }
  virtual IntVar cost(void) const {
    return violations;
  }
  ...
};
...
```

Figure 14.1: A script for the photo alignment problem

There are only two hard constraints for this model: no person can be in more than one place, and no two persons can stand in the same place. The first constraint is enforced automatically by the choice of variables, as each `pos` variable represents the unique position of a person (see also Tip 15.1). For the second constraint, the variables in the `pos` array must be pairwise distinct (see Section 4.4.7):

CONSTRAIN POSITIONS ≡
```
distinct(*this, pos, ICL_BND);
```

We choose the bounds consistent variant of `distinct` (by giving the extra argument `ICL_BND`, see Section 4.3.5) as also the other propagators perform only bounds reasoning.

The remaining constraints implement the preferences and turn them into a measure of *violation*, which expresses how many preferences are not fulfilled in a solution. A preference (i, j) is not fulfilled if the distance between the positions of person i and person j is greater than one. This can be implemented using a linear constraint, an absolute value constraint, and a reified constraint for each preference, as well as one linear constraint that constrains the sum of the violations:

PHOTO WITHOUT MODELING SUPPORT ≡ [DOWNLOAD]
```
...
    BoolVarArgs viol(*this,n_prefs,0,1);
    for (int i=0; i<n_prefs; i++) {
      IntVar distance(*this,0,n), diff(*this,-n,n);
      linear(*this, IntArgs(2, 1,-1),
             IntVarArgs() << pos[spec[i][0]] << pos[spec[i][1]],
             IRT_EQ, diff);
      abs(*this, diff, distance);
      rel(*this, distance, IRT_GR, 1, viol[i]);
    }
    linear(*this, viol, IRT_EQ, violations);
...
```

Using the MiniModel library (see Figure 7.2 and Section 7.1) yields more compact and readable code:

COMPUTE VIOLATIONS ≡
```
BoolVarArgs viol(n_prefs);
for (int i=0; i<n_prefs; i++) {
  viol[i] = expr(*this, abs(pos[spec[i][0]]-pos[spec[i][1]]) > 1);
}
rel(*this, violations == sum(viol));
```

We can observe that this problem has a symmetry, as reversing a solution yields again a solution. Symmetric solutions like this can be ruled out by arbitrarily picking two persons, and always placing one somewhere to the left of the other. For example, let us always place Betty somewhere to the left of Chris:

```
SYMMETRY BREAKING ≡
  rel(*this, pos[Betty] < pos[Chris]);
```

14.3 More information

This case study is also available as a Gecode example, see Placing people on a photo.

15 Locating warehouses

This chapter demonstrates the warehouse location problem. It shows how to use `element`, global counting (`count`), and `linear` constraints.

15.1 Problem

The problem is taken from [60, Chapter 10], see also Problem 34 in CSPLib. A company needs to construct warehouses to supply stores with goods. Each candidate warehouse has a certain capacity defining how many stores it can supply. Each store shall be supplied by exactly one warehouse. Maintaining a warehouse incurs a fixed cost. Costs for transportation from warehouses to stores depend on the locations of warehouses and stores.

We want to determine which warehouses should be opened (that is, supply to at least one store) and which warehouse should supply which store such that the overall cost (transportation costs plus fixed maintenance costs) is smallest.

In the following problem instance, the fixed maintenance cost `c_fixed` for a warehouse is 30. There are five candidate warehouses $w_0, \ldots, w_4$ and ten stores $s_0, \ldots, s_9$. The candidate warehouses have the following `capacity`:

w_0	w_1	w_2	w_3	w_4
1	4	2	1	3

The costs to supply a store by a candidate warehouse are defined by a matrix $\texttt{c_supply}_{i,j}$ ($0 \leq i < 10$, $0 \leq j < 5$) as follows:

	w_0	w_1	w_2	w_3	w_4
s_0	20	24	11	25	30
s_1	28	27	82	83	74
s_2	74	97	71	96	70
s_3	2	55	73	69	61
s_4	46	96	59	83	4
s_5	42	22	29	67	59
s_6	1	5	73	59	56
s_7	10	73	13	43	96
s_8	93	35	63	85	46
s_9	47	65	55	71	95

WAREHOUSES ≡ [DOWNLOAD]

```
...
const int n_warehouses = 5;
const int n_stores = 10;
const int capacity[n_warehouses] = {
  1, 4, 2, 1, 3
};
const int c_fixed = 30;
const int c_supply[n_stores][n_warehouses] = {
  ...
};

class Warehouses : public IntMinimizeScript {
protected:
  ▶ VARIABLES
public:
  Warehouses(const Options&)
    ▶ VARIABLE INITIALIZATION
  {
    {
      ▶ DO NOT EXCEED CAPACITY
    }
    ▶ OPEN WAREHOUSES
    ▶ COST FOR EACH WAREHOUSE
    ▶ TOTAL COST
    ▶ BRANCHING
  }
  ▶ COST FUNCTION
  ...
};
...
```

Figure 15.1: A script for locating warehouses

15.2 Model

The outline for the script implementing our model is shown in Figure 15.1. The data definitions are as described in the previous section.

As we need to minimize the total cost which is an integer variable, the script inherits from the class `IntMinimizeScript` which is a driver-defined subclass (similar to `Script` and `Space`) of `IntMinimizeSpace` (see Section 7.3), see Section 11.2. This in particular means that the class must define a virtual `cost()` function returning an integer variable, see below.

Variables. The script declares the following variables:

```
VARIABLES ≡
  IntVarArray  supplier;
  BoolVarArray open;
  IntVarArray  c_store;
  IntVar       c_total;
```

where:

- for each store s, there is a variable $\texttt{supplier}_s$ such that $\texttt{supplier}_s = w$ if warehouse w supplies store s;
- for each warehouse w, there is a Boolean variable $\texttt{open}_w$ which equals one, if the warehouse w supplies at least one store;
- for each store s, there is a variable $\texttt{c_store}_s$ which defines the cost for s to be supplied by warehouse $\texttt{supplier}_s$;
- a variable `c_total` which defines the total cost.

Tip 15.1 (Choose variables to avoid constraints)**.** Just by the choice of `supplier` variables where $\texttt{supplier}_s = w$ if warehouse w supplies store s, the problem constraint that each store shall be supplied by exactly one warehouse is enforced. Hence, no explicit constraints must be posted in our model. After all, no `supplier` variable can take on two different values!

This is good modeling practice: choosing variables such that some of the problem constraints are automatically enforced. ◀

The variables are initialized as follows:

```
VARIABLE INITIALIZATION ≡
  : supplier(*this, n_stores, 0, n_warehouses-1),
    open(*this, n_warehouses, 0, 1),
    c_store(*this, n_stores)
```

We only declare but do not initialize the cost variables `c_store` and `c_total`, as we will assign them by results obtained by posting expressions, see Section 7.1. The difference between declaring variables and initializing them such that new variables are created is explained in detail in Section 4.1.1.

Constraints. For a given warehouse w the following must hold: the number of stores s supplied by w is not allowed to exceed the capacity of w. This can be expressed by a counting constraint `count` (see Section 4.4.8) as follows:

```
DO NOT EXCEED CAPACITY ≡
  IntSetArgs c(n_warehouses);
  for (int w=0; w<n_warehouses; w++)
    c[w] = IntSet(0,capacity[w]);
  count(*this, supplier, c, ICL_DOM);
```

Here, the array of integer sets `c` defines how many stores can be supplied by a warehouse, where c_w contains every legal number of occurrences of w in `supplier`. To achieve strong propagation for the `count` constraint, we choose domain propagation by providing the additional argument `ICL_DOM` (see Section 4.3.5).

For a given warehouse w the following must hold: if the number of stores s supplied by w is at least one, then $\texttt{open}_w$ equals one. That is, $\texttt{open}_{\texttt{supplier}_s}$ must be `1` for all stores s. This is expressed by using `element` constraints (see Section 4.4.12) as follows:

```
OPEN WAREHOUSES ≡
  for (int s=0; s<n_stores; s++)
    element(*this, open, supplier[s], 1);
```

Cost computation. The cost $\texttt{c_store}_s$ for each store s is computed by an `element` constraint (see Section 4.4.12), mapping the warehouse supplying s to the appropriate cost:

```
COST FOR EACH WAREHOUSE ≡
  for (int s=0; s<n_stores; s++) {
    IntArgs c(n_warehouses, c_supply[s]);
    c_store[s] = expr(*this, element(c, supplier[s]));
  }
```

The total cost `c_total` is defined by the cost of open warehouses (that is, the sum of $\texttt{open}_w$ for all warehouses w multiplied with the fixed maintenance cost for a warehouse) and the cost of stores (that is, the sum of the $\texttt{c_store}_s$ for all stores s):

```
TOTAL COST ≡
  c_total = expr(*this, c_fixed*sum(open) + sum(c_store));
```

Note that the linear expression posted for defining the total cost mixes integer and Boolean variables and is automatically decomposed into the appropriate `linear` constraints, see Section 7.1.

Tip 15.2 (Small variable domains are still beautiful)**.** As mentioned in Tip 4.3, initializing variable domains to be small makes sense.

Here we choose to post expressions (see Section 7.1) via the `expr()` function that returns an integer variable. The integer variable returned by `expr()` has automatically computed and reasonable bounds.

A different choice would be to initialize the variables `c_store` and `c_total` in the constructor of the script and constrain them explicitly (for example, via the `rel()` function where the expression for `expr()` is turned into an equality relation). But that would mean that we either have to compute some estimates for the bounds used for initializing the variables or resort — without real necessity — to the largest possible integer value. ◀

Branching. The branching proceeds in two steps, implemented by two different branchings. The first branching assigns values to the variables $\texttt{c_store}_s$ and follows the strategy of maximal regret: select the variable for which the difference between the smallest value and the next larger value is maximal (see Section 8.2). The second branching makes sure that all stores are being assigned a warehouse. This branching is necessary as supplying a store could have the same cost for several different warehouses (depending on the data used for the model). Hence, even though all variables in `c_store` are assigned, not all variables in `supplier` must be assigned and hence the total cost is also not assigned. The branchings are posted in the appropriate order as follows:

BRANCHING ≡
```
branch(*this, c_store, INT_VAR_REGRET_MIN_MAX(), INT_VAL_MIN());
branch(*this, supplier, INT_VAR_NONE(), INT_VAL_MIN());
```

Cost function. The cost function `cost()` to be used by the search engine is defined to return the total cost `c_total`:

COST FUNCTION ≡
```
virtual IntVar cost(void) const {
  return c_total;
}
```

15.3 More information

This problem is also available as a Gecode example, see Locating warehouses. The model presented in [60, Chapter 10] proposes a better branching than the branching shown in the previous section.

16 Nonogram

This chapter shows how to use regular expressions and extensional constraints for solving nonogram puzzles.

16.1 Problem

Nonograms (Problem 12 in CSPLib) are popular puzzles in which the puzzler shades squares in a matrix. Each instance of the puzzle has constraints on the rows and columns of the matrix, specifying the number and length of the groups of consecutive marks in that row or column. For example, a row that in the solution has the marks

□□■■□□■■■□■□□□

has the *hint* `2 3 1`, indicating that there are three separate groups of marks, with lengths 2, 3, and 1. Given two groups of marks, there must be at least one empty square in between. An example nonogram is given in Figure 16.1 and its solution is shown in Figure 16.2. The general nonogram problem is NP-complete, as shown in [57].

16.2 Model

The model follows naturally from the constraints of the problem. The variables needed are a matrix x_{ij} of `0-1` variables, representing the squares to shade. For the hint `2 3 1` on row i, we post the following `extensional` constraint (see Section 4.4.13):

$$\texttt{extensional}(x_{i\bullet}, 0^*1^20^+1^30^+10^*)$$

The regular expression starts and ends with zero or more zeroes. Each group is represented by as many ones as the group length. In between the groups, one or more zeroes are placed. Using this construction, we get one constraint per row and column of the matrix. The outline of the script is shown in Figure 16.3.

Puzzle specification. The puzzle is specified by an array of integers. The first two integers specify the width and the height of the grid. These are followed first by the column and then the row hints. Each hint specifies the number of groups and the length of each group. For

			3	2 3	2 2	2 2	2 2	2 2	2 2	2 3	3
	2	2									
	4	4									
1	3	1									
2	1	2									
	1	1									
	2	2									
	2	2									
		3									
		1									

Figure 16.1: Example nonogram puzzle

			3	2 3	2 2	2 2	2 2	2 2	2 2	2 3	3
	2	2		■	■				■	■	
	4	4	■	■	■	■		■	■	■	■
1	3	1	■			■	■	■			■
2	1	2	■	■			■			■	■
	1	1		■						■	
	2	2		■	■				■	■	
	2	2			■	■		■	■		
		3				■	■	■			
		1					■				

Figure 16.2: Solution to the example puzzle

NONOGRAM ≡ [DOWNLOAD]

```
...
► PUZZLE
class Nonogram : public Script {
  int width, height;
  BoolVarArray b;

  DFA line(const int*& p) {
    ► LINE FUNCTION
  }
public:
  Nonogram(const Options& opt)
    : width(spec[0]), height(spec[1]),
      b(*this,width*height,0,1) {
    Matrix<BoolVarArray> m(b, width, height);
    ► INTIALIZE HINT POINTER
    ► COLUMN CONSTRAINTS
    ► ROW CONSTRAINTS
    ► BRANCHING
  }
  ...
};

int main(int argc, char* argv[]) {
  ...
}
```

Figure 16.3: A script for solving nonogram puzzles

example, the hint used as an example above is specified as `3, 2, 3, 1`. The puzzle from Figure 16.1 is written as follows.

PUZZLE ≡
```
const int spec[] =
  { 9, 9,
    // Column hints
    1, 3,
    2, 2, 3,
    ...
    // Row hints
    2, 2, 2,
    ...
  };
```

Line function. For the hint that starts at `p` (that is, `p` points to a position in the array of integers `spec[]` where a hint starts), the following code constructs a regular expression (see Section 7.4) that matches that hint.

LINE FUNCTION ≡
```
int nhints = *p++;
REG r0(0), r1(1);
REG border = *r0;
REG separator = +r0;
REG result = border;
if (nhints > 0) {
  result += r1(*p,*p);
  p++;
  for (int i=nhints-1; i--; p++)
    result += separator + r1(*p,*p);
}
return result + border;
```

The variables `r0` and `r1` represent the constants `0` and `1` (this is a slight optimization to construct a regular expression for the constants `0` and `1` just once). The variables `border` and `separator` represent sequences of zeroes at the borders and between marks. The loop adds all hints (as repeated `r1`s) to the `result` expression with `separators` in between. Note that if the hint is just `0` (representing an empty line with no mark), then the `result` will be just the same as a `border+border`, which is $0^*0^* = 0^*$.

Constraints. Given the `line()` function, posting the appropriate constraints is as follows. The pointer `p` is initialized to point to the first hint:

INTIALIZE HINT POINTER ≡
```
const int* p = spec+2;
```

Two loops go through all the hints, get the regular expression for the line, and post the constraints for the appropriate variables. First, the column constraints are posted:

```
COLUMN CONSTRAINTS ≡
  for (int w=0; w<width; w++)
    extensional(*this, m.col(w), line(p));
```

followed by the row constraints:

```
ROW CONSTRAINTS ≡
  for (int h=0; h<height; h++)
    extensional(*this, m.row(h), line(p));
```

Branching. Choosing a branching for nonograms is not obvious. For many nonogram puzzles, using an AFC-based branching (see Section 8.2) is a good idea:

```
BRANCHING ≡
  branch(*this, b, INT_VAR_AFC_MAX(), INT_VAL_MAX());
```

The choice to use `INT_VAL_MAX()` is because most puzzles will have fewer marks than empty spaces. For the example puzzle from Figure 16.1, propagation alone solves the puzzle. To solve really hard puzzles, a custom branching may be needed.

16.3 More information

The nonogram puzzle is also included as a Gecode example, see Nonogram. The example in particular features several grids to try the model on.

Despite its simplicity, the program for solving nonograms works amazingly well. Extensive information on nonogram puzzles and a comparison of different nonogram solvers (including the model described in this chapter) is Survey of Paint-by-Number Puzzle Solvers.

17 Social golfers

This chapter presents a case study on modeling problems using set variables and constraints.

17.1 Problem

The social golfers' problem (Problem 10 in CSPLib) requires finding a schedule for a golf tournament. There are $g \cdot s$ golfers who want to play a tournament in g groups of s golfers each over w weeks, such that no two golfers play against each other more than once during the tournament.

Here is a solution for the instance $w = 4$, $g = 3$, and $s = 3$, where the players are numbered from 0 to 8:

	Group 0			*Group 1*			*Group 2*		
Week 0	0	1	2	3	4	5	6	7	8
Week 1	0	3	6	1	4	7	2	5	8
Week 2	0	4	8	1	5	6	2	3	7
Week 3	0	5	7	1	3	8	2	4	6

17.2 Model

The model for the social golfers' problem closely follows the above problem description. Its outline is shown in Figure 17.1. The script defines an array of set variables `groups` of size $\mathtt{g} \cdot \mathtt{w}$, where each group can contain the players $0 \ldots \mathtt{g} \cdot \mathtt{s} - 1$ and has cardinality `s` (see Section 5.1).

The script also defines a matrix `schedule` with g columns and w rows on top of the variable array, such that `schedule(i,j)` is the set of members of group `i` in week `j`.

The constraints are straightforward. For each week, the union of all groups must be disjoint and contain all players. This can be expressed directly using a disjoint union constraint (see Section 7.1) on the rows of the `schedule`:

GROUPS IN A WEEK ≡

```
SetVar allPlayers(*this, 0,g*s-1, 0,g*s-1);
for (int i=0; i<w; i++)
  rel(*this, setdunion(schedule.row(i)) == allPlayers);
```

GOLF ≡ [DOWNLOAD]

```
...
class GolfOptions : public Options {
...
};

class Golf : public Script {
  int g, s, w;
  SetVarArray groups;
public:
  Golf(const GolfOptions& opt)
  : g(opt.g()), s(opt.s()), w(opt.w()),
    groups(*this,g*w,IntSet::empty,0,g*s-1,s,s) {
    Matrix<SetVarArray> schedule(groups,g,w);
    ▶ GROUPS IN A WEEK
    ▶ OVERLAP BETWEEN GROUPS
    ▶ BREAK GROUP SYMMETRY
    ▶ BREAK WEEK SYMMETRY
    ▶ BREAK PLAYER SYMMETRY
    branch(*this, groups, SET_VAR_MIN_MIN(), SET_VAL_MIN_INC());
  }
  ...
};
...
```

Figure 17.1: A script for the social golfers' problem

Each group can have at most one player in common with any other group. This can be expressed by a constraint that states that the cardinality of the intersection between any two groups must be at most 1:

OVERLAP BETWEEN GROUPS ≡

```
for (int i=0; i<groups.size()-1; i++)
  for (int j=i+1; j<groups.size(); j++)
    rel(*this, cardinality(groups[i] & groups[j]) <= 1);
```

Symmetry breaking. Using set variables to model the groups already avoids introducing symmetry among the players in a group. For example, if we had modeled each group as s integer variables, any permutation of these variables would produce an equivalent solution.

But there are more symmetries in this problem, and some of them can be avoided easily by introducing additional *symmetry breaking constraints*.

Within a week, the order of the groups is irrelevant. Therefore, we can impose a static order requiring that all minimal elements of each group are ordered increasingly (see Section 5.2.5 for the minimal element constraint, Section 7.1 for the MiniModel support, and Section 4.4.3 for ordering integer variables):

BREAK GROUP SYMMETRY ≡
```
for (int j=0; j<w; j++) {
  IntVarArgs m(g);
  for (int i=0; i<g; i++)
    m[i] = expr(*this, min(schedule(i,j)));
  rel(*this, m, IRT_LE);
}
```

Similarly to the group symmetry, the order of the weeks is irrelevant. Again, the symmetry can be broken by imposing an order on the group elements. The previous constraint made sure that player `0` will always be in `schedule(0,j)` for any week `j`. So imposing an order on the second smallest element of `schedule(0,j)` will do the trick:

BREAK WEEK SYMMETRY ≡
```
IntVarArgs m(w);
for (int j=0; j<w; j++)
  m[j] = expr(*this, min(schedule(0,j)-IntSet(0,0)));
rel(*this, m, IRT_LE);
```

Finally, the players can be permuted arbitrarily. For example, swapping the numbers 2 and 6 in the initial example produces a symmetric solution:

	Group 0			*Group 1*			*Group 2*		
Week 0	0	1	6	3	4	5	2	7	8
Week 1	0	3	2	1	4	7	6	5	8
Week 2	0	4	8	1	5	2	6	3	7
Week 3	0	5	7	1	3	8	6	4	2

This symmetry can be broken using the `precede` constraint (see Section 5.2.9):

BREAK PLAYER SYMMETRY ≡
```
precede(*this, groups, IntArgs::create(groups.size(),0));
```

It enforces for any pair of players s and t that t can only appear in a group without s if there is an earlier group where s appears without t. This establishes an order that breaks the value symmetry between the players. In the example above, the constraint rules out that 6 appears in group 0, week 0, because that would require 2 to appear in an earlier group. The only solution that remains after symmetry breaking is the one in the initial table in Section 17.1.

Note that these symmetry breaking constraints do not necessariyl break all symmetries of the problem completely. We mainly discussed them as additional examples of modeling with set variables and constraints.

17.3 More information

The case study is also available as a Gecode example, see Golf tournament. You can find a discussion of the symmetry breaking constraints presented here and a number of additional implied constraints in [2].

18 Knight's tour

This chapter demonstrates a problem-specific brancher inspired by a classic heuristic for a classic problem.

Important. This case study requires knowledge on programming branchers, see Part B.

18.1 Problem

The problem of a knight's tour is to find a series of knight's moves (a knight can move two fields vertically and simultaneously one field horizontally, or vice versa) on an empty $n \times n$ board starting from some initial field such that:

- each field is visited exactly once, and
- there is a further knight's move from the last field of the tour to the initial field.

Figure 18.1 shows a knight's tour for $n = 8$, where the tour starts at the lower left corner.

18.2 Model

The model for the knight's tour uses a successor representation for the knight's moves to make posting the constraints straightforward. To further simplify posting the constraints, the variables in the model use *fields* on the board as values. The field 0 has $\langle x, y \rangle$-coordinates $\langle 0, 0 \rangle$ on the board, the field 1 has coordinates $\langle 1, 0 \rangle$, the field n on an $n \times n$ board has coordinates $\langle 0, 1 \rangle$, and so on. That is, fields on the board are counted first in x-direction and then in y-direction.

The successor representation means that a variable in the model has the field numbers as possible values that can be reached by a knight's move. The model uses the `circuit` constraint (see Section 4.4.17) to enforce that the tour is in fact a Hamiltonian circuit. The only additional constraints needed are that fields must be reachable only by knight's moves.

Figure 18.2 outlines the program to implement the knight's tour model. An object of class `Knights` stores the size of the board as its member `n`. The variables for the knight moves are stored in the integer variable array (see Section 4.2.1) `succ`. The array `succ` has n^2 elements where the variable at position `f` stores the successor of the field `f`. The function

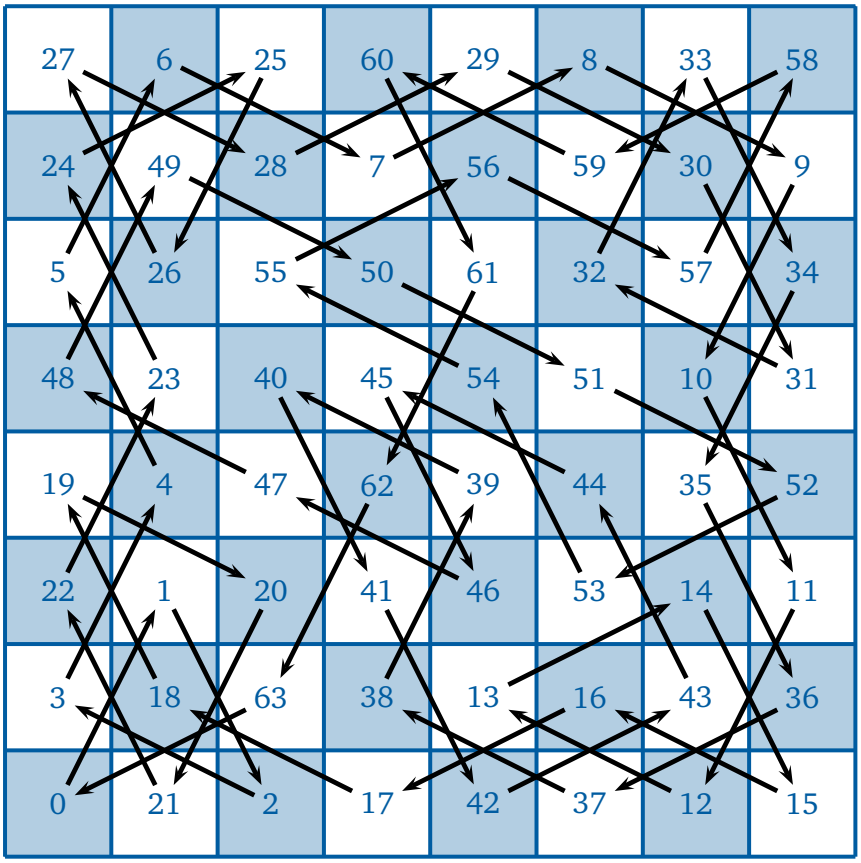


Figure 18.1: 8×8-knight's tour

KNIGHTS ≡ [DOWNLOAD]

```
#include <climits>
...
▶ BRANCHER
class Knights : public Script {
protected:
  const int n;
  IntVarArray succ;
public:
  int f(int x, int y) const { return x + y*n; }
  ...
  IntSet neighbors(int i) {
    ...
  }
  Knights(const SizeOptions& opt)
    : n(opt.size()), succ(*this,n*n,0,n*n-1) {
    ▶ KNIGHT'S MOVES
    ▶ FIX FIRST MOVE
    ▶ HAMILTONIAN CIRCUIT
    warnsdorff(*this, succ);
  }
  ...
};
...
```

Figure 18.2: A script for the knight's tour problem

`f(x,y)` computes the field number for coordinates $\langle x, y\rangle$. For example, `f(0,0)=0`, `f(1,0)=1`, and `f(0,1)=n`.

Enforcing knight's moves. Domain constraints `dom` (see Section 4.4.1) are used to constrain moves to knight's moves:

```
KNIGHT'S MOVES ≡
  for (int i=0; i<n*n; i++)
    dom(*this, succ[i], neighbors(i));
```

The function `neighbors(i)` returns an integer set which contains the fields that are reachable from field `i` by a knight's move. For example, for $n = 8$, `neighbors(0)` (the field `0` has coordinates $\langle 0, 0\rangle$) returns the integer set $\{17, 10\}$ (that is, the fields with coordinates $\langle 2, 1\rangle$ and $\langle 1, 2\rangle$) and `neighbors(27)` (the field `27` has coordinates $\langle 3, 3\rangle$) returns the integer set $\{17, 33, 10, 42, 12, 44, 21, 37\}$ (that is, the fields with coordinates $\langle 2, 1\rangle$, $\langle 4, 1\rangle$, $\langle 1, 2\rangle$, $\langle 5, 2\rangle$, $\langle 1, 4\rangle$, $\langle 5, 4\rangle$, $\langle 2, 5\rangle$, and $\langle 4, 5\rangle$).

Fixing the first move. Without loss of generality we fix that the knight's first move goes to field `f(1,2)`:

```
FIX FIRST MOVE ≡
  rel(*this, succ[0], IRT_EQ, f(1,2));
```

Fixing the first move can be seen as breaking a symmetry in the model and hence reduces the amount of search needed for finding a knight's tour.

Enforcing a Hamiltonian circuit. The `circuit` constraint (see Section 4.4.17) enforces that the tour of knight's moves forms a Hamiltonian circuit:

```
HAMILTONIAN CIRCUIT ≡
  circuit(*this, succ, ICL_DOM);
```

We request domain propagation for the `circuit` constraint by providing `ICL_DOM` as argument (see Section 4.3.5).[1] Intuitively, we want to have the strongest possible propagation available for `circuit` as it is the only constraint.

18.3 Branching

The really interesting aspect of solving the knight's tour puzzle is to find a branching that works well. A classic approach is Warnsdorff's heuristic [65]: move the knight to a field that has the least number of further possible moves.

[1] Of course, domain propagation for `circuit` does not achieve domain consistency, as the problem of finding Hamiltonian circuits is NP-complete [15, p. 199].

In terms of our model, Warnsdorff's heuristic has of course no procedural notion of *moving* the knight! Instead, our branching finds a yet unassigned field i on the board (a field whose successor is not known yet). Then, it first tries a value for $\texttt{succ}_i$ that moves the knight to a field with the least number of further possible moves. That is, the branching first tries a value `n` for $\texttt{succ}_i$ such that the domain size of $\texttt{succ}_n$ is smallest. If there are several such values, it just tries the smallest first (as it is most natural to implement).

The brancher. Figure 18.3 shows an outline of a branching and a brancher implementing Warnsdorff's heuristic. The members of `Warnsdorff` are exactly the same as for the example brancher in Section 31.2.2: the view array `x` stores the knight's moves, the **`mutable`** integer `start` points to the current unassigned view in `x`, and the `PosVal` choice stores the position of the view and the value to be used for branching.

Status computation. The `status()` function tries to find a yet unassigned view for branching in the view array `x`. It starts inspecting the views from position `start`. If it finds an assigned view, it moves `start` to the position in the view array as defined by the assigned view's value. In other words, `status()` follows partially constructed knight's tours. If `status()` finds an unassigned view, it returns **`true`**:

STATUS FUNCTION ☰

```
virtual bool status(const Space&) const {
  for (int n=0; n<x.size(); n++) {
    if (!x[start].assigned())
      return true;
    start = x[start].val();
  }
  return false;
}
```

The number of attempts to find an unassigned view is limited by the number of views in the view array `x`. If the limit is exceeded, all views are assigned and hence the `status()` function returns **`false`**.

Choice computation. The `choice()` function implements the actual heuristic. It chooses the value of `x[start]` for branching that has the smallest domain size as follows:

CHOICE FUNCTION ☰

```
virtual Choice* choice(Space&) {
  int n=-1; unsigned int size=UINT_MAX;
  for (Int::ViewValues<Int::IntView> i(x[start]); i(); ++i)
    if (x[i.val()].size() < size) {
      n=i.val(); size=x[n].size();
    }
  return new PosVal(*this,start,n);
}
```

As mentioned above, to keep the implementation simple, if there are several values that have smallest domain size, the first is chosen (and hence the smallest).

Note that the value `UINT_MAX` is larger than the size of any view domain.[2] Hence, it is guaranteed that the **`for`**-loop will always choose a value from the domain of `x[start]`. The choice returned is the typical implementation to store position and value, similar to the examples in Section 31.2.2.

18.4 More information

The model is also available as a Gecode example, see n-Knights tour (model using circuit). The Gecode example also features a simple standard branching that can be compared to Warnsdorff's branching and a naive model using reification instead of `circuit` (see n-Knight's tour (simple model)).

[2] `UINT_MAX` (for the maximal value of an unsigned integer) is available as the program includes the `<climits>` header file.

BRANCHER ☰

```
class Warnsdorff : public Brancher {
protected:
  ViewArray<Int::IntView> x;
  mutable int start;
  class PosVal : public Choice {
    ...
  };
public:
  ...
  ▶ STATUS FUNCTION
  ▶ CHOICE FUNCTION
  ...
};

void warnsdorff(Home home, const IntVarArgs& x) {
  ...
}
```

Figure 18.3: A brancher for Warnsdorff's heuristic

19 Bin packing

This chapter studies the classic bin packing problem. Three models are presented: a first and naive model (presented in Section 19.2) that suffers from poor propagation to be feasible. This is followed by a model (Section 19.3) that uses the special `binpacking` constraint to drastically improve constraint propagation. A final model improves the second model by a problem-specific branching (Section 19.4) that also breaks many symmetries during search.

Important. This case study requires knowledge on programming branchers, see Part B.

19.1 Problem

The bin packing problem consists of packing `n` items of sizes $\texttt{size}_i$ ($0 \leq i < \texttt{n}$) into the smallest number of bins such that the capacity `c` of each bin is not exceeded.

For example, the 11 items of sizes

$$6, 6, 6, 5, 3, 3, 2, 2, 2, 2, 2$$

require at least four bins of capacity 10 as shown in Figure 19.1 where the items are numbered starting from zero.

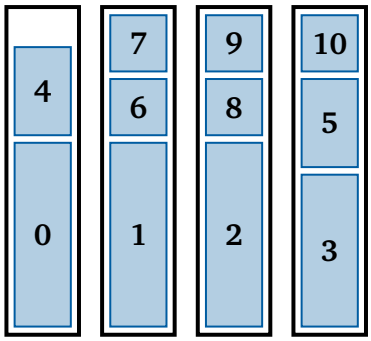


Figure 19.1: An example optimal bin packing

INSTANCE DATA ≡

```
const int c = 100;
const int n = 50;
const int size[n] = {
  99,98,95,95,95,94,94,91,88,87,86,85,76,74,73,71,68,60,55,54,51,
  45,42,40,39,39,36,34,33,32,32,31,31,30,29,26,26,23,21,21,21,19,
  18,18,16,15,5,5,4,1
};
```

Figure 19.2: Instance data for a bin packing problem

COMPUTE LOWER BOUND ≡

```
int lower(void) {
  int s=0;
  for (int i=0; i<n; i++)
    s += size[i];
  return (s + c - 1) / c;
}
```

Figure 19.3: Computing a lower bound for the number of bins

19.2 A naive model

Before turning our attention to a naive model for the bin packing problem, this section discusses instance data for the bin packing problem and how to compute lower and upper bounds for the number of required bins.

Instance data. Figure 19.2 shows an example instance of a bin packing problem, where `n` defines the number of items, `c` defines the capacity of each `bin`, and the array `size` defines the size of each item. For simplicity, we assume that the item size are ordered in decreasing order.

The data corresponds to the instance `N1C1W1_N` taken from [42]. More information on other data instances can be found in Section 19.5.

Computing a lower bound. A simple lower bound L_1 (following Martello and Toth [29]) for the number of bins required for a bin packing problem just considers the size of all items and the bin capacity as follows:

$$L_1 = \left\lceil \frac{1}{c} \sum_{i=0}^{n-1} \texttt{size}_i \right\rceil$$

```
COMPUTE UPPER BOUND ≡
  int upper(void) {
    int* free = new int[n];
    for (int i=0; i<n; i++)
      free[i] = c;
    int u=0;
    ▶ PACK ITEMS INTO FREE BINS
    delete [] free;
    return u+1;
  }
```

Figure 19.4: Computing an upper bound for the number of bins

The computation of the lower bound L_1 is as to be expected and is shown in Figure 19.3. The ceiling operation is replaced by adding `c` $-$ 1 followed by truncating integer division with `c`.

Note that more accurate lower bounds are known, see Section 19.5 for more information.

Computing an upper bound. An obvious upper bound for the number of bins required is the number of items: each item is packed into a separate bin (provided that no item size exceeds the bin capacity `c`). A better upper bound can be computed by constructing a solution by packing items into bins following a first-fit strategy: pack all items into the first bin of sufficient free capacity.

Figure 19.4 shows the function `upper()` that returns an upper bound for the number of bins. It initializes an array `free` of `n` integers with the bin capacity `c`. The integer `u` refers to the index of the last used bin (that is, a bin into which an item has been packed). The function returns the number of used bins (that is, the index `u` plus one).

Each item is packed into a bin with sufficient free capacity where `j` refers to the next free bin:

```
PACK ITEMS INTO FREE BINS ≡
  for (int i=0; i<n; i++) {
    int j=0;
    ▶ FIND FREE BIN
    u = std::max(u,j);
  }
```

The next free bin `j` is searched for as follows:

```
FIND FREE BIN ≡
  while (free[j] < size[i])
    j++;
  free[j] -= size[i];
```

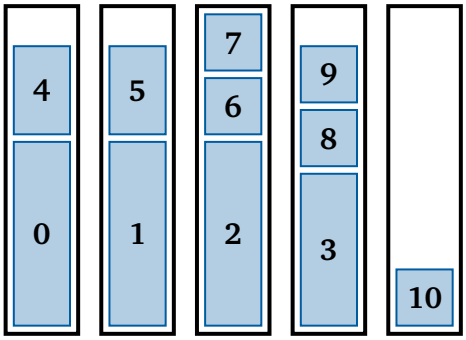


Figure 19.5: An non-optimal bin packing found during upper bound computation

The loop always terminates as there is one bin for each item.

Note that `upper()` has $O(n^2)$ complexity in the worst case but could be made more efficient by speeding up finding a fitting bin.

The solution constructed during computation of `upper()` is not necessarily optimal. As an example, consider the packing computed by `upper()` for the example from Section 19.1 shown in Figure 19.5: it takes five rather than four bins because one of the items 4 and 5 should be packed together with item 3 rather than with one of the items 0, 1, and 2.

If both lower and upper bound coincide the solution constructed is of course optimal and we are done solving the bin packing problem. For reasons of simplicity, our model just forsakes this opportunity and re-computes an optimal solution by constraint programming.

Model proper. Figure 19.6 shows a script for the bin packing model. The script defines integers `l` and `u` that store the lower and upper bound as discussed above. A *load variable* $\texttt{load}_i$ (taking values from $[0 .. c]$) defines the total size of all items packed into bin i. The script uses `u` load variables as the upper bound guarantees that `u` bins are sufficient to find an optimal solution. A *bin variable* $\texttt{bin}_i$ (taking values from $[0 .. u-1]$ defines for each item i into which bin it is packed. The variable `bins` defines the number of *used bins*. A bin is *used* if at least one item is packed into it, otherwise it is an *excess* bin.

The integer `s` is initialized as the size of all items and `sizes` is initialized as an integer argument array of all sizes.

The `cost()` function as required by the class `MinimizeScript` (see Section 11.2) returns the number of `bins`.

Excess bins. If the script finds a solution that uses less than `u` bins, say `k` (the value of the `bins` variable), then $u-k$ of the load variables corresponding to excess bins are zero. To remove many symmetrical solutions that only differ in which bins are excess bins, the script constrains the excess bins to be the bins $k, \ldots, u-1$:

```
EXCESS BINS ≡
  for (int i=1; i<=u; i++)
    rel(*this, (bins < i) == (load[i-1] == 0));
```

BIN PACKING NAIVE ≡ [DOWNLOAD]

```
...
► INSTANCE DATA
► COMPUTE LOWER BOUND
► COMPUTE UPPER BOUND
class BinPacking : public MinimizeScript {
protected:
  const int l;
  const int u;
  IntVarArray load;
  IntVarArray bin;
  IntVar bins;
public:
  BinPacking(const Options& opt)
    : l(lower()), u(upper()),
      load(*this, u, 0, c),
      bin(*this, n, 0, u-1), bins(*this, l, u) {
    ► EXCESS BINS
    int s=0;
    for (int i=0; i<n; i++)
      s += size[i];
    IntArgs sizes(n, size);
    ► LOADS ADD UP TO ITEM SIZES
    ► LOADS ARE EQUAL TO PACKED ITEMS
    ► SYMMETRY BREAKING
    ► PACK ITEMS THAT REQUIRE A BIN
    ► BRANCHING
  }
  virtual IntVar cost(void) const {
    return bins;
  }
  ...
};
...
```

Figure 19.6: A naive script for solving a bin packing problem

Constraining load and bin variables. The sum of all load variables must be equal to the size of all items:

LOADS ADD UP TO ITEM SIZES ≡
```
linear(*this, load, IRT_EQ, s);
```

The load variable for a bin must be constrained according to which items are packed into the bin. A standard formulation of this constraint uses Boolean variables $x_{i,j}$ which determine whether item i has been packed into bin j. That is, for each item $0 \leq i < n$ the following constraint must hold:

$$x_{i,j} = 1 \iff \mathtt{bin}_i = j \qquad (0 \leq j < u)$$

A more efficient propagator for the very same constraint is available as a `channel` constraint between an array of Boolean variables and a single integer variable, see Section 4.4.11. That is, for each item $0 \leq i < n$ the following constraint must hold:

$$\mathtt{channel}(\langle x_{i,0}, x_{i,1}, \ldots, x_{i,u-1} \rangle, \mathtt{bin}_i)$$

Note that $\langle x_{i,0}, x_{i,1}, \ldots, x_{i,u-1} \rangle$ corresponds to `x.col(`i`)`.

Furthermore, the size of all items packed into a bin must equal the corresponding load variable. Both constraints are expressed as follows, using a matrix `x` (see Section 7.2) of Boolean variables `_x`:

LOADS ARE EQUAL TO PACKED ITEMS ≡
```
BoolVarArgs _x(*this, n*u, 0, 1);
Matrix<BoolVarArgs> x(_x, n, u);
for (int i=0; i<n; i++)
  channel(*this, x.col(i), bin[i]);
for (int j=0; j<u; j++)
  linear(*this, sizes, x.row(j), IRT_EQ, load[j]);
```

Symmetry breaking. Items of the same size are equivalent as far as the model is concerned. To break symmetries, the bins for items of the same size are ordered:

SYMMETRY BREAKING ≡
```
for (int i=1; i<n; i++)
  if (size[i-1] == size[i])
    rel(*this, bin[i-1] <= bin[i]);
```

The loop exploits that items are ordered according to size and hence items of the same size are adjacent.

Pack items that require a bin. If the size s of an item exceeds $\lceil \frac{c}{2} \rceil$ (or, equivalently, $2s > c$), the item cannot share a bin with any other item also exceeding half of the capacity. That is, items exceeding half of the capacity can be directly assigned to different bins:

```
PACK ITEMS THAT REQUIRE A BIN ≡
  for (int i=0; (i < n) && (i < u) && (size[i] * 2 > c); i++)
    rel(*this, bin[i] == i);
```

The assignment of items to bins is compatible with the symmetry breaking constraints discussed previously.

Branching. We choose a naive branching strategy that first branches on the number of required `bins`, followed by trying to assign items to bins.

```
BRANCHING ≡
  branch(*this, bins, INT_VAL_MIN());
  branch(*this, bin, INT_VAR_NONE(), INT_VAL_MIN());
```

Note that by choosing the `bin` variables with order `INT_VAR_NONE()` assigns the largest item to a bin first as items are sorted by decreasing size.

The script in Figure 19.6 does not show that the script uses branch-and-bound search to find a best solution. Why depth-first search is not sufficient with parallel search is discussed in Tip 9.4.

Running the model. When running the naive model[1], it becomes apparent that the model is indeed naive. Finding the best solution takes 29.5 seconds and 2 451 018 failures. Clearly, that leaves ample room for improvement in the following sections!

19.3 Improving propagation

This section improves (and simplifies) the naive model from the previous section by using a dedicated `binpacking` constraint.

Model. The improved model is shown in Figure 19.7. Instead of using Boolean variables `x`, `linear` constraints, and `channel` constraints it uses the `binpacking` constraint (see also Section 4.4.15). The constraint enforces that the packing of items as defined by the `bin` variables corresponds to the `load` variables.

Running the model. Finding a best solution using the model with improved propagation takes 1.5 seconds and 64 477 failures. That is, this model runs almost 20 times faster than the naive model and reduces the number of failures by a factor of 38.

[1] All measurements in this chapter have been made on a laptop with an Intel i5 M430 processor (2.27 GHz, two cores, hyper-threading), 4 GB of main memory, running Windows 7 x64, and using Gecode 3.4.3.

BIN PACKING PROPAGATION ≡ [DOWNLOAD]

```
...
class BinPacking : public MinimizeScript {
  ...
public:
  BinPacking(const Options& opt)
    : ... {
    ...
    IntArgs sizes(n, size);
    binpacking(*this, load, bin, sizes);
    ...
  }
  ...
};
...
```

Figure 19.7: A script with improved propagation for solving a bin packing problem

19.4 Improving branching

This section describes a problem specific branching to improve the model of the previous section even further.

Complete decreasing best fit branching. The improved branching for bin packing is called complete decreasing best-fit (CDBF) and is due to Gent and Walsh [17]. The branching uses some additional improvements suggested by Shaw in [53].

The branching tries to assign items to bins during search where the items are tried in order of decreasing size. The bin is selected according to a best fit strategy: try to put the item into a bin with sufficient but least free space. The space of the bin after packing an item is called the bin's *slack*. If there is no bin with sufficient free space left, CDBF fails.

Suppose that CDBF selects item `i` and bin `b`. Then the following actions are taken during branching:

- If there is a perfect fit (that is, the slack is zero), branching assigns item `i` to bin `b`. This corresponds to a branching with a single alternative.
- If all possible bins have the same slack, branching assigns item `i` to bin `b`. Again, this corresponds to a branching with a single alternative.
- Otherwise, CDBF tries two alternatives in the following order:
  - Assign item `i` to bin `b`.

BIN PACKING BRANCHING ≡ [DOWNLOAD]

```
...
►CDBF

class BinPacking : public MinimizeScript {
  ...
public:
  BinPacking(const Options& opt)
    : ... {
    ...
    branch(*this, bins, INT_VAL_MIN());
    cdbf(*this, load, bin, sizes);
  }
  ...
};
...
```

Figure 19.8: A script with improved branching for solving a bin packing problem

- Not only prune bin `b` from the potential bins for item `i` but also prune all bins with the same slack as `b` from the potential bins for all items with the same size as `i`.

Note that the second alternative of CDBF performs symmetry breaking during search as it prunes also with respect to equivalent items and bins.

Model. The only change to the model compared to Section 19.3 is that it uses the branching `cdbf` for assigning items to bins during search.

Brancher creation. The `cdbf` branching takes load variables (for computing the free space of a bin), bin variables (to pack items into bins), and the item sizes (to compute how much space an item requires) as input and posts the `CDBF` brancher as shown in Figure 19.9.

The branching post function `cdbf()` creates view arrays for the respective variables, creates a shared integer array of type `IntSharedArray` (see Tip 4.9) and posts a `CDBF` brancher. The advantage of using a shared array is that the sizes are stored only once in memory and branchers in different spaces have shared access to the same memory area (see also Section 30.3).

The brancher `CDBF` stores the load variables, bin variables, and item sizes together with an integer `item`. The integer `item` is used to find the next unassigned item. It is declared **`mutable`** so that the **`const`** `status()` function (see below) can modify it. The brancher exploits that the items are sorted by decreasing size: by initializing `item` to zero the brancher is trying to pack the largest item first.

```
CDBF ≡
  class CDBF : public Brancher {
  protected:
    ViewArray<Int::IntView> load;
    ViewArray<Int::IntView> bin;
    IntSharedArray size;
    mutable int item;
    ► CDBF CHOICE
  public:
    CDBF(Home home, ViewArray<Int::IntView>& l,
                    ViewArray<Int::IntView>& b,
                    IntSharedArray& s)
      : Brancher(home), load(l), bin(b), size(s), item(0) {
      home.notice(*this,AP_DISPOSE);
    }
    static void post(Home home, ViewArray<Int::IntView>& l,
                                ViewArray<Int::IntView>& b,
                                IntSharedArray& s) {
      (void) new (home) CDBF(home, l, b, s);
    }
    ► STATUS FUNCTION
    ► CHOICE FUNCTION
    ► COMMIT FUNCTION
    ...
    virtual size_t dispose(Space& home) {
      home.ignore(*this,AP_DISPOSE);
      size.~IntSharedArray();
      return sizeof(*this);
    }
  };

  void cdbf(Home home, const IntVarArgs& l, const IntVarArgs& b,
                       const IntArgs& s) {
    if (b.size() != s.size())
      throw Int::ArgumentSizeMismatch("cdbf");
    ViewArray<Int::IntView> load(home, l);
    ViewArray<Int::IntView> bin(home, b);
    IntSharedArray size(s);
    CDBF::post(home, load, bin, size);
  }
```

Figure 19.9: CDBF brancher and branching

By default, the `dispose()` member function of a brancher is not called when the brancher's home space is deleted. However, the `dispose()` function of the `CDBF` brancher must call the destructor of the shared integer array `size`. Hence, the constructor of `CDBF` calls the `notice()` function of the home space so that the brancher's `dispose()` function is called when home is deleted (see also Section 22.9). Likewise, the `dispose()` function calls the `ignore()` function before the brancher is disposed.

Status computation. The `status()` function tries to find a yet unassigned view for branching in the view array `bin`. It starts inspecting the views at position `item` and skips all already assigned views. If there is a not yet assigned view left, `item` is updated to that unassigned view and **`true`** is returned (that is, more branching is needed). Otherwise, the brancher returns **`false`** as no more branching is needed:

STATUS FUNCTION ≡
```
virtual bool status(const Space&) const {
  for (int i = item; i < bin.size(); i++)
    if (!bin[i].assigned()) {
      item = i; return true;
    }
  return false;
}
```

As the items are sorted by decreasing size, the integer `item` refers to the largest not-yet packed item.

Choice computation: initialization. The `choice()` function implements the actual heuristic. The function uses `n` for the number of items, `m` for the number of bins, and initializes a `region` for managing temporary memory (see Section 30.1) as follows:

CHOICE FUNCTION ≡
```
virtual Gecode::Choice* choice(Space& home) {
  int n = bin.size(), m = load.size();
  Region region(home);
  ⟨INITIALIZE FREE SPACE IN BINS⟩
  ⟨INITIALIZE BINS WITH SAME SLACK⟩
  ⟨FIND BEST FIT⟩
  ⟨CREATE CHOICE⟩
}
```

The `choice()` function can rely on the fact that it is immediately executed after the `status()` function has been executed. That entails that `item` refers to the largest not-yet packed item.

The function computes in `free` the free space of each bin. From the maximal load the size of items that have already been packed (that is, the item's `bin` variable is already assigned) is subtracted:

INITIALIZE FREE SPACE IN BINS ≡
```
int* free = region.alloc<int>(m);
for (int j=0; j<m; j++)
  free[j] = load[j].max();
for (int i=0; i<n; i++)
  if (bin[i].assigned())
    free[bin[i].val()] -= size[i];
```

The `choice()` function uses the integer `slack` to track the slack of the so-far best fit (initialized with `INT_MAX` such that any fit will be better). The integer `n_possible` counts the number of possible bins for the item whereas `n_same` counts the number of best fits. The array `same` stores all bins with the same so-far smallest slack.

INITIALIZE BINS WITH SAME SLACK ≡
```
int slack = INT_MAX;
unsigned int n_possible = 0;
unsigned int n_same = 0;
int* same = region.alloc<int>(m+1);
same[n_same++] = -1;
```

The array `same` is initialized to contain the bin `-1`: if no bin has sufficient space for the current item this will guarantee that the `commit()` function (see below) leads to failure.

Choice computation: create choice. In order to find all best fits, all bins are examined. If the current item fits into a bin, the number of possible bins `n_possible` is incremented and all best fits are remembered in the array `same` as follows:

FIND BEST FIT ≡
```
for (Int::ViewValues<Int::IntView> j(bin[item]); j(); ++j)
  if (size[item] <= free[j.val()]) {
    n_possible++;
    if (free[j.val()] - size[item] < slack) {
      slack = free[j.val()] - size[item];
      n_same = 0;
      same[n_same++] = j.val();
    } else if (free[j.val()] - size[item] == slack) {
      same[n_same++] = j.val();
    }
  }
```

Note that finding a better fit updates `slack` and resets the bins stored in `same`.

Now, the `choice()` function determines whether a special case needs to be dealt with:

- Is the best fit a perfect fit: that is, `slack` is zero?
- Are all fits a best fit: that is, `n_same` is equal to `n_possible`?
- Is there no fitting bin: that is, `n_possible` is zero?

In these cases a choice with a single alternative and otherwise a choice with two alternatives is created:

```
CREATE CHOICE ≡
  if ((slack == 0) ||
      (n_same == n_possible) ||
      (n_possible == 0))
    return new Choice(*this, 1, item, same, 1);
  else
    return new Choice(*this, 2, item, same, n_same);
```

The definition of the choice class is shown in Figure 19.10. The choice stores the current item and in the array `same` all bins with the same slack. The choice does not need to store any information regarding items of same size as this information is available from the brancher.

Commit function. The `commit()` function takes a choice of type `CDBF::Choice` and the alternative `a` (either `0` or `1`) as input:

```
COMMIT FUNCTION ≡
  virtual ExecStatus commit(Space& home, const Gecode::Choice& _c,
                            unsigned int a) {
    const Choice& c = static_cast<const Choice&>(_c);
    if (a == 0) {
      ▶ COMMIT TO FIRST ALTERNATIVE
    } else {
      ▶ COMMIT TO SECOND ALTERNATIVE
    }
    return ES_OK;
  }
```

Committing to the first alternative tries to pack `item` into the first bin stored in `same` as follows:

```
COMMIT TO FIRST ALTERNATIVE ≡
  GECODE_ME_CHECK(bin[c.item].eq(home, c.same[0]));
```

Committing to the second alternative removes all `n_same` bins stored in `same` from all items that have the same size as `item` as follows:

CDBF CHOICE ≡

```
class Choice : public Gecode::Choice {
public:
  int  item;
  int* same;
  int  n_same;
  Choice(const Brancher& b, unsigned int a, int i, int* s, int n_s)
    : Gecode::Choice(b,a), item(i),
      same(heap.alloc<int>(n_s)), n_same(n_s) {
    for (int k=0; k<n_same; k++)
      same[k] = s[k];
  }
  virtual size_t size(void) const {
    return sizeof(Choice) + sizeof(int) * n_same;
  }
  virtual ~Choice(void) {
    heap.free<int>(same,n_same);
  }
  virtual void archive(Archive& e) const {
    Gecode::Choice::archive(e);
    e << alternatives() << item << n_same;
    for (int i=n_same; i--;) e << same[i];
  }
};
```

Figure 19.10: CDBF choice

COMMIT TO SECOND ALTERNATIVE ≡

```
int i = c.item;
do {
  Iter::Values::Array same(c.same, c.n_same);
  GECODE_ME_CHECK(bin[i++].minus_v(home, same));
} while ((i < bin.size()) &&
         (size[i] == size[c.item]));
```

The iterator `Iter::Values::Array` iterates over all values stored in an array (they must be in sorted order) and the operation `minus_v()` prunes all values as defined by an iterator from a view, see Section 25.2.

Running the model. Finding a best solution using the model with improved propagation and improved branching takes 84 milliseconds and 3 098 failures. That is, this model runs 352 times faster than the naive model and reduces the number of failures by a factor of 791.

19.5 More information

Bin packing featuring all models presented in this chapter is also available as a Gecode example, see Bin packing. The example also makes use of a more accurate lower bound known as L_2 [29].

20 Kakuro

This chapter studies Kakuro puzzles, a variant of the well-known Sudoku puzzles. Two models are presented: a first and obvious model that suffers from too little propagation to be feasible. This is followed by a model that employs user-defined constraints implemented as `extensional` constraints using tuple set specifications. Interestingly, the tuple set specifications are computed by solving a simple constraint problem.

20.1 Problem

Solving a Kakuro puzzle (see Figure 20.1 for an example) amounts to finding digits between 1 and 9 for the non-hint fields on a board. A hint field can specify a *vertical hint* and/or a *horizontal hint*:

- A vertical hint contains a number s above the diagonal in the hint field. It requires that all digits on the fields extending from the field left of the hint up to the next hint field or to the end of the row are pairwise distinct and sum up to the value s.
- A horizontal hint contains a number s below the diagonal in the hint field. The requirements are analogous.

The number contained in a hint is called its *value* and the number of fields constrained by a hint is called its *length*.

The solution for the Kakuro puzzle from Figure 20.1 is shown in Figure 20.2. Kakuro puzzles are always designed (at least meant to be) to have a unique solution.

20.2 A naive model

A script for the Kakuro model is shown in Figure 20.3. The script stores the width (`w`) and height (`h`) of the board. The fields are stored in an integer variable array `f` which is initialized to have `w·h` elements. Note that none of the fields is initialized in the constructor of `Kakuro`; their initialization is discussed below.

Board specification. The specification of the Kakuro board, as shown in Figure 20.3, stores width and height of the board, followed by a specification of the hints. The hints are provided in two groups: first vertical hints, then horizontal hints, separated by the integer `-1`. A hint

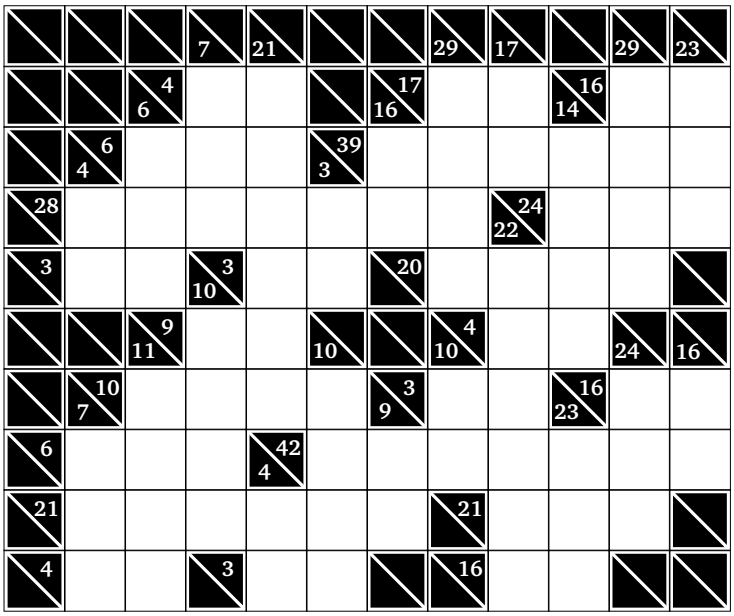


Figure 20.1: A Kakuro puzzle

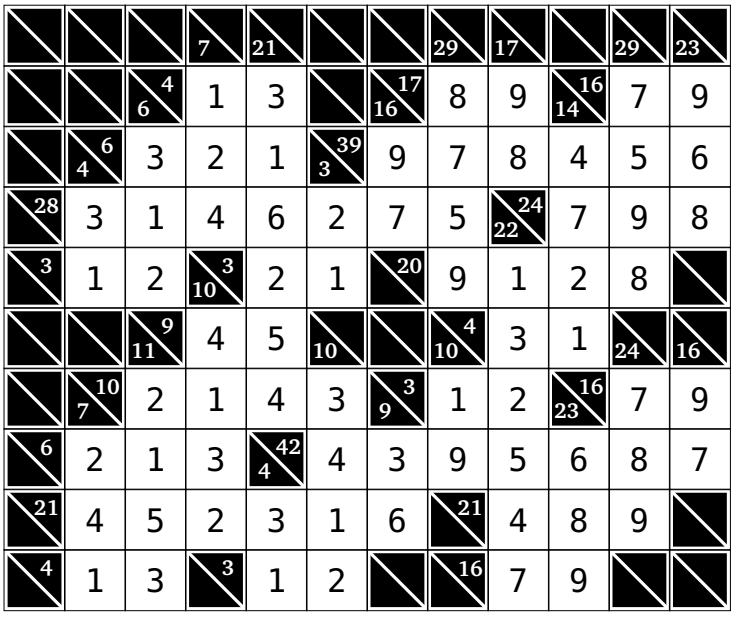


Figure 20.2: Solution for Kakuro puzzle from Figure 20.1

KAKURO NAIVE ≡ [DOWNLOAD]

```
...
► BOARD SPECIFICATION
class Kakuro : public Script {
protected:
  const int w, h;
  IntVarArray f;
public:
  ► INIT FUNCTION
  ► POSTING HINT CONSTRAINTS
  Kakuro(const Options& opt)
    : w(board[0]),  h(board[1]), f(*this,w*h) {
    ► FIELD INITIALIZATION
    ► SETUP
    ► PROCESS VERTICAL HINTS
    ...
    ► BRANCHING
  }
  ...
};
...
```

BOARD SPECIFICATION ≡

```
const int board[] = {
  // Dimension w x h
  12, 10,
  // Vertical hints
   3, 0, 3, 7,     4, 0, 6,21,     7, 0, 4,29,     8, 0, 2,17,
  ...
  -1,
  // Horizontal hints
   2, 1, 2, 4,     6, 1, 2,17,     9, 1, 2,16,     1, 2, 3, 6,
  ...
};
```

Figure 20.3: A naive script and board specification for solving Kakuro puzzles

is described by its coordinates on the board, followed by its length and value. Note that the specification assumes that the field with coordinate $\langle 0,0 \rangle$ is in the left upper corner.

Initializing fields. All fields are initialized to a single shared integer variable `black` that is assigned to zero:

FIELD INITIALIZATION ≡
```
IntVar black(*this,0,0);
for (int i=0; i<w*h; i++)
  f[i] = black;
```

Only if a field is actually used by a hint, the field will be initialized to an integer variable taking digit values by the following `init()` function:

INIT FUNCTION ≡
```
IntVar init(IntVar& x) {
  if (x.min() == 0)
    x = IntVar(*this,1,9);
  return x;
}
```

The test whether the minimum of variable `x` equals zero is true, if and only if `x` still refers to the variable `black`. In this case, a new variable is created with non-zero digits as variable domain. As `x` is passed by reference, assigning `x` to the newly created variable also assigns the corresponding field on the board to the newly created variable. This guarantees that a new variable is created at most once for each field.

Posting hint constraints. Posting hint constraints is done best by using a matrix interface `b` (see Section 7.2) to the fields in `f`. The specification of the hints will be accessed by the variable `k`, where the dimension of the board has already been skipped:

SETUP ≡
```
Matrix<IntVarArray> b(f,w,h);
const int* k = &board[2];
```

Processing the vertical hints is straightforward. After retrieving the coordinates `x` and `y`, the length `n`, and the value `s` for a hint from the board specification, the variables covered by the hint are collected in the integer variable argument array (see Section 4.2.2) `col`. The constraint for the hint on the collected variables is posted by the member function `hint()`:

PROCESS VERTICAL HINTS ≡
```
while (*k >= 0) {
  int x=*k++; int y=*k++; int n=*k++; int s=*k++;
  IntVarArgs col(n);
  for (int i=0; i<n; i++)
    col[i]=init(b(x,y+i+1));
  hint(col,s);
}
```

The `hint()` function must constrain that all variables are distinct (using a `distinct` constraint) and that they sum up to the value of the hint (using a `linear` constraint). To achieve strong propagation, we want to use domain propagation for both `distinct` and `linear`. However, the complexity of domain propagation for `linear` is exponential, hence it is a good idea to avoid posting `linear` constraints as much a possible.

Consider a hint of length 9. Then obviously, the single possible value of the hint is $\sum_{i=1}^{9} i = 9(9+1)/2$ and hence no `linear` constraint needs to be posted. Now consider a hint of length 8 with value s. Then, the fields covered by the hint take on all but one digit. That is, all fields must be different from $\sum_{i=1}^{9} i - s = 9(9+1)/2 - s$. Taking these two observations into account, the constraints for a hint can be posted as follows, where the value `ICL_DOM` requests domain propagation (see Section 4.3.5):

POSTING HINT CONSTRAINTS ≡
```
void hint(const IntVarArgs& x, int s) {
  if (x.size() < 8)
    linear(*this, x, IRT_EQ, s, ICL_DOM);
  else if (x.size() == 8)
    rel(*this, x, IRT_NQ, 9*(9+1)/2 - s);
  distinct(*this, x, ICL_DOM);
}
```

Note that there are other special cases where no `linear` constraint needs to be posted, for example if for a hint of length n and value s it holds that $\sum_{i=1}^{n} i = s$ (that is, only digits from 1 to n are possible). See Section 20.4 for more information.

Vertical hints are of course analogous and are hence omitted.

Branching. We choose a branching that selects the variable where the quotient of AFC and domain size is largest smallest (see Section 8.2). Values are tried by interval bisection:

BRANCHING ≡
```
branch(*this, f, INT_VAR_AFC_SIZE_MAX(), INT_VAL_SPLIT_MIN());
```

Why the model is poor. When running the script, solving even the tiny board of Figure 20.1 requires 19 search nodes. There exist commercially available boards with thousands of hints, which are of course completely out of reach with the naive model. Figure 20.4 shows the possible digits for each field after performing propagation for the Kakuro script but before any search. Consider the two green fields for the hint of length 2 and value 4. The only possible combination for the two fields is $\langle 3, 1 \rangle$. However, propagation does not prune the value 2 for both fields. The reason is that a `hint` constraint is decomposed into a `distinct` constraint and into a `linear` constraint and neither constraint by itself warrants more pruning than shown.

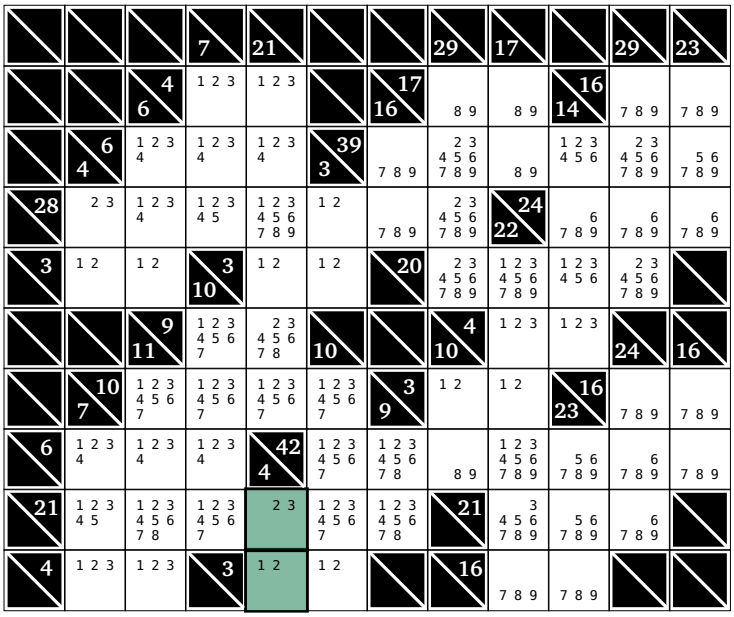


Figure 20.4: Propagation for the Kakuro puzzle

20.3 A working model

The naive model from the previous section suffers from the fact that `hint` constraints are decomposed into a `distinct` constraint and a `linear` constraint. One remedy would be to implement a dedicated propagator for a `distinctlinear` constraint. This is impractical: too complicated and too much effort for such a specialized constraint.

Model idea. This section implements `distinctlinear` constraints as `extensional` constraints using tuple sets as specification of the possible solutions of `distinctlinear` constraints. For example, for a hint of length 3 and value 8, the possible solutions for the corresponding `distinctlinear` constraint are:

$$\begin{array}{llll} \langle 1,2,5\rangle & \langle 1,3,4\rangle & \langle 1,4,3\rangle & \langle 1,5,2\rangle \\ \langle 2,1,5\rangle & \langle 2,5,1\rangle & \langle 3,1,4\rangle & \langle 3,4,1\rangle \\ \langle 4,1,3\rangle & \langle 4,3,1\rangle & \langle 5,1,2\rangle & \langle 5,2,1\rangle \end{array}$$

The model needs a method to compute all solutions of a `distinctlinear` constraint. To simplify matters, we are going to compute all solutions of a `distinctlinear` constraint by computing all solutions of a trivial constraint problem: the decomposition of a `distinctlinear` constraint into a `distinct` and `linear` constraint.

Figure 20.5 shows the outline for a working script for solving Kakuro puzzles. The class `DistinctLinear` defines the script used for computing all solutions of a `distinctlinear` constraint and the function `distinctlinear()` serves as constraint post function. Apart from how `hint` constraints are posted, the `Kakuro` script is the same as in the previous section.

KAKURO ≡ [DOWNLOAD]

```
...
class DistinctLinear : public Space {
protected:
  IntVarArray x;
public:
  ▶ DISTINCT LINEAR SCRIPT
  ▶ RETURNING A SOLUTION
  ...
};

void distinctlinear(Home home, const IntVarArgs& x, int c) {
  ▶ SET UP SEARCH ENGINE
  ▶ COMPUTE TUPLE SET
  ▶ POST EXTENSIONAL CONSTRAINT
}

class Kakuro : public Script {
  ...
  ▶ POSTING HINT CONSTRAINTS
  ...
};
...
```

Figure 20.5: A working script for solving Kakuro puzzles

Computing distinct linear solutions. As mentioned, the script for `DistinctLinear` just posts a `linear` and `distinct` constraint for `n` variables and value `s`. As the search space of the problem is small anyway, we neither need strong propagation for `distinct` and `linear` nor do we need a clever branching:

DISTINCT LINEAR SCRIPT ≡
```
DistinctLinear(int n, int s) : x(*this,n,1,9) {
  distinct(*this, x);
  linear(*this, x, IRT_EQ, s);
  branch(*this, x, INT_VAR_NONE(), INT_VAL_SPLIT_MIN());
}
```

When solving the `DistinctLinear` script, we need its solutions as integer argument arrays for computing a tuple set. The `solution()` member function of `DistinctLinear` returns an integer argument array for a solution as follows:

RETURNING A SOLUTION ≡
```
IntArgs solution(void) const {
  IntArgs s(x.size());
  for (int i=0; i<x.size(); i++)
    s[i]=x[i].val();
  return s;
}
```

Posting `distinctlinear` constraints. The search engine (see Section 9.3) for computing all solutions of a `DistinctLinear` script is initialized as follows:

SET UP SEARCH ENGINE ≡
```
DistinctLinear* e = new DistinctLinear(x.size(),c);
DFS<DistinctLinear> d(e);
delete e;
```

Computing a tuple set (see Section 4.4.13) for all solutions of a `distinctlinear` constraints is straightforward:

COMPUTE TUPLE SET ≡
```
TupleSet ts;
while (DistinctLinear* s = d.next()) {
  ts.add(s->solution()); delete s;
}
ts.finalize();
```

Note that after all solutions have been added to the tuple set `ts`, it must be finalized before it can be used by an `extensional` constraint (see Section 4.4.13).

Finally, posting the `extensional` constraint using the tuple set `ts` is as to be expected:

POST EXTENSIONAL CONSTRAINT ≡
```
extensional(home, x, ts);
```

Posting hint constraints. Posting a `hint` constraint follows a similar line of reasoning as in the previous section. If the length of a hint is 0, no constraint needs to be posted (hints of length 0 are black fields without hints). If the length is 1, the single variable is constrained to `s` directly. For lengths 8 and 9, `distinct` is used as it achieves the same propagation as `distinctlinear`. Note that the case for length 8 continues (as it does not have a **`break`** statement) with the case for length 9 and hence also posts a `distinct` constraint. In all other cases, `distinctlinear` is used:

POSTING HINT CONSTRAINTS ≡

```
void hint(const IntVarArgs& x, int s) {
  switch (x.size()) {
  case 0:
    break;
  case 1:
    rel(*this, x[0], IRT_EQ, s); break;
  case 8:
    rel(*this, x, IRT_NQ, 9*(9+1)/2 - s);
  case 9:
    distinct(*this, x, ICL_DOM); break;
  default:
    distinctlinear(*this, x, s); break;
  }
}
```

There is a further important optimization which we will not show (but see Section 20.4). Each time `distinctlinear` is called, it computes a new tuple set, even though the tuple set is exactly the same for all hints of equal length and value. To guarantee that the same tuple set is computed at most once, one could cache tuple sets: if a tuple set for a certain length and value has already been computed earlier, it is not computed again but taken from a cache (where it had been stored when it was computed for the first time).

This model works. For the example puzzle, propagation alone is sufficient to solve the puzzle. Even puzzles with thousands of hints are solved without search in a fraction of a second (including computing the tuple sets, provided they are cached as sketched above).

20.4 More information

Kakuro puzzles with some more examples are available as a Gecode example, see Kakuro. In particular, the model caches tuple sets such that for each type of hint its corresponding tuple set is computed at most once as discussed in Section 20.3. Furthermore, the example exploits further special cases where posting a `distinct` constraint rather than a complete `hint` constraint is sufficient as discussed in Section 20.2.

More constraint-based techniques for solving Kakuro puzzles are discussed in [54].

21 Crossword puzzle

This chapter studies solving crossword puzzles and presents a simple model using nothing but `distinct` and `element` constraints.

The simple model for this classical problem is shown to work quite well compared to a constraint-based approach to solving crossword puzzles using a dedicated problem-specific constraint solver [1]. This underlines that an efficient general-purpose constraint programming system actually can go a long way.

21.1 Problem

To solve a crossword puzzle problem, a crossword grid (see Figure 21.1 for an example) must be filled with words (from a predefined dictionary) extending both in horizontal and vertical directions such that:

- If words cross at a field of the grid, the words' letters at the crossing field are the same.
- No word is used twice.

Words use lowercase letters only and extend as far as they can. That is, the beginning (and the end) of a word must either be adjacent to a black field on the grid or must be a field on the grid's border.

An example solution for the grid from Figure 21.1 is shown in Figure 21.2.

21.2 Model

The model uses two sets of variables:

- The model uses for each word on the grid a *word variable*. A value for a word variable is a *dictionary index* defining the index of the word chosen from the dictionary of all words.

  The script for the model uses the word variables only as temporary variables for posting constraints. To simplify posting constraints, words are processed in groups of words of the same length. In particular, dictionary indices are also defined with respect to words of the same length in the dictionary.

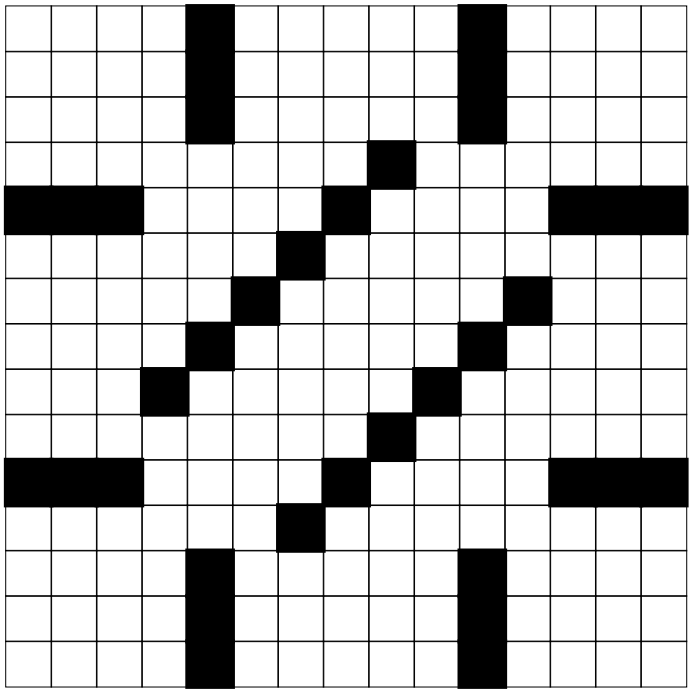


Figure 21.1: A crossword puzzle grid

a	l	b	s	■	r	a	j	a	h	■	s	l	i	p
l	i	r	e	■	a	m	i	g	o	■	h	i	d	e
s	e	a	l	■	v	e	n	o	m	■	a	m	o	k
o	u	t	f	l	a	n	k	■	e	d	i	b	l	e
■	■	■	l	a	g	s	■	f	l	a	k	■	■	■
s	c	h	e	m	e	■	b	i	a	t	h	l	o	n
c	h	a	s	e	■	w	a	n	n	a	■	e	w	e
r	i	d	s	■	h	a	d	e	d	■	h	a	l	e
a	l	e	■	s	i	l	l	s	■	d	a	r	e	d
m	i	s	a	p	p	l	y	■	f	e	i	n	t	s
■	■	■	t	u	b	a	■	s	e	a	r	■	■	■
f	e	d	o	r	a	■	k	i	e	l	b	a	s	a
r	a	i	l	■	t	e	a	e	d	■	a	g	o	g
o	v	a	l	■	h	a	l	v	e	■	n	u	d	e
g	e	l	s	■	s	t	e	e	r	■	d	e	a	d

Figure 21.2: Solution for crossword puzzle grid from Figure 21.1

```
CROSSWORD ≡                                                    [DOWNLOAD]
  ...
  ▶ GRID SPECIFICATION
  ...
  ▶ WORDS SPECIFICATION
  class Crossword : public Script {
  protected:
    const int w, h;
    IntVarArray letters;
  public:
    Crossword(const Options& opt)
      : w(grid[0]), h(grid[1]), letters(*this,w*h,'a','z') {
      ▶ SET UP
      ▶ INITIALIZE BLACK FIELDS
      ▶ PROCESS WORDS BY LENGTH
      ▶ BRANCHING
    }
    ▶ PRINT FUNCTION
    ...
  };
  ...
```

Figure 21.3: Crossword script

- For each field on the grid, the model uses a *letter variable*. The values for a letter variable are either `0` (for a black field on the grid) or a character code between `'a'` and `'z'` for lowercase letters.

Given the sets of variables, the constraints for the model are straightforward:

- All word variables for words of the same length must be `distinct`. Only word variables for words of the same length need to be constrained to be distinct, as words of different length are distinct by definition.

- Assume that w is a word variable for a word of length n on the grid and $x_0, \ldots, x_{n-1}$ are the letter variables that correspond to the word on the grid. Assume further that $0 \leq p < n$ and that an array `w2l` (for `w`ord to `l`etter) maps the dictionary indices of all words of length n to their p-th letter. Then, the letter variable x_p can be linked to the word variable w by posting the constraint that $\texttt{w2l}_w = x_p$ (this is an `element` constraint).

An outline for the script implementing the crossword puzzle is shown in Figure 21.3. The script stores the width `w` and the height `h` of the grid and an integer variable array `letters`

GRID SPECIFICATION ≡

```
const int grid[] = {
  // Width and height of crossword grid
  15, 15,
  // Number of black fields
  36,
  // Black field coordinates
  0,4, 0,10, 1,4, 1,10, 2,4, 2,10, 3,8, 4,0, 4,1,
  ...
  // Length and number of words of that length
  8, 8,
  // Coordinates where words start and direction (0 = horizontal)
  0,3,0, 0,9,0, 3,0,1, 5,7,1, 7,5,0, 7,11,0, 9,0,1, 11,7,1,
  ...
  // End marker
  0
};
```

WORDS SPECIFICATION ≡

```
const int n_words[] = {
  1, 26, 66, 633, 2443, 4763, 7585, 10380, 10974
};
const char** words[] = {
  ...
};
```

Figure 21.4: Grid and words specification

for the letter variables (including the black fields). The values for `letters` range from `'a'` to `'z'` (black fields are discussed below).

Grid and words specification. The specification for the `grid` used in this case study (see Section 21.4 for more information) and the word dictionary are shown in Figure 21.4. The grid specification contains information about the dimension of the grid (as used in Figure 21.3), the number and coordinates of black fields on the grid, and the start coordinates of words and their direction on the grid for each word length permitted by the grid.

For each word length l, the array $\texttt{n_words}_l$ defines how many words of length l exist in the dictionary of words. That is, for a word length l, the set of dictionary indices is $\{0,\ldots,\texttt{n_words}_l-1\}$. The array `words` provides access to the letters of a word of some given length with a given dictionary index. That is, for a given word length l and for a position in the word p with $0 \leq p < l$, $\texttt{words}_{l,i,p}$ (or `words[`l`][`i`][`p`]` in C++) is the p-th letter of the word with dictionary index i among all words of length l (where $0 \leq i < \texttt{n_words}_l$) in the

dictionary. The dictionary of words just contains words of length at most eight as this is sufficient for the example grid used in this case study.

The word list is based on SCOWL-55 (Spell Checking Oriented Word Lists) truncated to words of length at most eight, see wordlist.sourceforge.net. Please check the source file available from Figure 21.3 for copyright information.

Grid initialization. The grid specification is accessed by the pointer g (with the width and height part already skipped). The matrix ml (see Section 7.2) supports access to the letters as a matrix:

```
SET UP ≡
  const int* g = &grid[2];
  Matrix<IntVarArray> ml(letters, w, h);
```

The black fields of the grid are initialized by storing a variable black at the respective coordinates:

```
INITIALIZE BLACK FIELDS ≡
  IntVar black(*this,0,0);
  for (int n = *g++; n--; ) {
    int x=*g++, y=*g++; ml(x,y)=black;
  }
```

At first sight, the treatment of black fields in the grid appears to be inefficient. First, each element in the integer variable array letters is initialized (in the initialization list of the constructor Crossword()) to a new integer variable with values ranging from 'a' to 'z'. Then, some variables become redundant as their fields are overwritten by black. However, this only matters initially when a space of class Crossword is created. As soon as a clone of that space is created, the redundant variables are not copied and hence do not matter any longer. Moreover, all black fields on the grid share a single variable black which saves memory compared to a variable for each black field on the grid.

Processing words by length. As suggested by the grid specification, words are processed in groups of the same length. The loop that processes all words of the same length l has the following structure:

```
PROCESS WORDS BY LENGTH ≡
  while (int l = *g++) {
    int n = *g++;
    ▶ INITIALIZE ARRAY OF WORDS
    ▶ PROCESS WORD ON GRID
  }
```

Here, n is initialized to the number of words with length l in the grid.

To enforce that all n words of the same length l are distinct, an integer argument array wosl (for words of same length) is created (see Section 4.2.1), where each variable takes

the possible dictionary indices for words of length `l` as values. The word variables in `wosl` are constrained to be distinct as follows:

```
INITIALIZE ARRAY OF WORDS ≡
  IntVarArgs wosl(*this,n,0,n_words[l]-1);
  distinct(*this, wosl);
```

Constraining letters by words. The remaining constraints link a word variable to the variables for its letters. All words of length `l` are processed as follows:

```
PROCESS WORD ON GRID ≡
  IntArgs w2l(n_words[l]);
  for (int i=0; i<n; i++) {
    int x = *g++, y = *g++; bool h = (*g++ == 0);
    ▶ PROCESS EACH LETTER POSITION
  }
```

The integer argument array `w2l` is used to map the dictionary indices of all words of length `l` in the dictionary to their letters. The `x`-coordinate and the `y`-coordinate and whether the word extends horizontally (`h` is **`true`**) or vertically (`h` is **`false`**) is retrieved from the grid specification.

Linking a word variable to a single letter is done for all `l` letters in a word, where the integer `p` refers to the position of a letter in a word:

```
PROCESS EACH LETTER POSITION ≡
  for (int p=0; p<l; p++) {
    ▶ CONSTRAIN LETTERS
  }
```

The integer argument array `w2l` is used to map all words in the dictionary of length `l` to their `p`-th letters. Then, for each letter position an `element` constraint (see Section 4.4.12) is posted that links the word variables to the respective letter variable:

```
CONSTRAIN LETTERS ≡
  for (int j=0; j<n_words[l]; j++)
    w2l[j] = words[l][j][p];
  element(*this, w2l, wosl[i], h ? ml(x+p,y) : ml(x,y+p));
```

Branching. We choose a simple branching that selects a variable where the quotient of AFC and domain size is largest (see Section 8.2). The first value for the selected variable to be tried is the smallest:

```
BRANCHING ≡
  branch(*this, letters, INT_VAR_AFC_SIZE_MAX(), INT_VAL_MIN(),
                         NULL, &printletters);
```

Additionally we pass a variable value print function (see Section 8.12) so that additional information about the branching is printed when, for example, using Gist:

PRINT FUNCTION ≡
```
  static void printletters(const Space& home,
                           const BrancherHandle& bh,
                           unsigned int a,
                           IntVar, int i, const int& n,
                           std::ostream& o) {
    const Crossword& c = static_cast<const Crossword&>(home);
    int x = i % c.w, y = i / c.w;
    o << "letters[" << x << "," << y << "] "
      << ((a == 0) ? "=" : "!=") << " "
      << static_cast<char>(n);
  }
```

21.3 An optimized model

The model in the previous section wastes some memory: for all word variables for words of length `l`, the array `w2l` is the same for a given position `p`. However, the very same array is computed `n` times: for each word variable for words of length `l`.

The first optimization is to swap the loops that iterate over the dictionary index `i` and the letter position `p`, as shown in Figure 21.5. However, one can go even further. By default, each time an `element` constraint is posted, a new shared array for the integer argument array is created (these will be still shared among all spaces of the same thread). To just have a single copy of the array, we can create a shared integer array of type `IntSharedArray` instead. Then, for each word length `l` and each position `p` there will be a single shared array only. See Tip 4.9 for more on shared arrays.

The shared integer array is initialized as follows:

INITIALIZE WORD TO LETTER ARRAY ≡
```
  IntSharedArray w2l(n_words[l]);
  for (int j=0; j<n_words[l]; j++)
    w2l[j] = words[l][j][p];
```

The very same shared integer array is used for all words of the same length from the dictionary as follows:

CONSTRAIN LETTERS ≡
```
  for (int i=0; i<n; i++) {
    int x = g[3*i+0], y = g[3*i+1];
    bool h = (g[3*i+2] == 0);
    element(*this, w2l, wosl[i], h ? ml(x+p,y) : ml(x,y+p));
  }
```

CROSSWORD OPTIMIZED ≡ [DOWNLOAD]

```
...
class Crossword : public Script {
  ...
  Crossword(const Options& opt)
    : w(grid[0]), h(grid[1]), letters(*this,w*h,'a','z') {
    ...
    while (int l = *g++) {
      int n = *g++;
      ...
      for (int p=0; p<l; p++) {
        ▶ INITIALIZE WORD TO LETTER ARRAY
        ▶ CONSTRAIN LETTERS
      }
      g += 3*n;
    }
    ...
  }
  ...
};
...
```

Figure 21.5: An optimized crossword script

In summary, the optimization does not offer a better model but a more memory-efficient implementation of the same model.

21.4 More information

The script that is shown in this case study is also available as a Gecode example called Crossword puzzle. The example features a number of different crossword grids and supports branching on the word variables or on the letter variables. In addition, arbitrary dictionaries can be used provided they are available as a list of words in a file.

Related work. Solving crossword puzzles is a classic example for search methods (see for example [19]) and also for constraint programming (see for example [3] and [1]).

Anbulagan and Botea introduce COMBUS in [1]. COMBUS is a constraint-based solver specialized at solving crossword puzzles. Its distinctive feature is that it uses nogood-learning to speed up search.

In the following, we are going to compare COMBUS to the model presented in this chapter. The purpose of the comparison is to shed light on the respective advantages of a problem-specific solver such as COMBUS and a simple model using a modern off-the-shelf constraint programming system such as Gecode.

Used hardware and software platform. All experiments have been run on a laptop with an Intel i7-2860QM processor (2.5 GHz, four cores, hyper-threading), 8 GB of main memory, running Windows 7 x64 and using Gecode 4.2.1. The model has been run with a single thread only. The runtimes are measured as wall-time and are the average of five runs. The coefficient of deviation is less than 5% and typically less than 1%.

The hardware platform used in [1] is an Intel Core Duo 2.4 GHz.

Comparison with COMBUS. The purpose of the comparison is to understand better the relative merits of the two different approaches. An exact comparison of runtimes is therefore not really meaningful, in particular as different hardware platforms are used.

The comparison makes only approximate statements for runtime and number of nodes explored during search. If the runtime for the Gecode model and COMBUS differ by at most a factor of two in either direction, the two approaches are roughly the same, denoted by $\approx$. If the runtime for the Gecode model is two to ten times faster, we use +; if it is ten to 100 times faster, we use ++; if it is more than 100 times faster, we use +++. Analogously, we use -, --, and --- if the Gecode model is slower than COMBUS. We also use the same symbols for comparing the number of nodes explored during search, where fewer nodes are of course better.

Figure 21.6 shows the results for the Gecode model and their comparison to COMBUS for the dictionaries `words` (containing 45 371 words) and `uk` (containing 225 349 words), where

words dictionary												
	15 × 15				19 × 19				21 × 21				23 × 23			
	time		nodes		time		nodes		time		nodes		time		nodes	
01	1.5	++	897	−−	0.4	+++	145	≈	87.1	≈	17 605	−−	0.0	+++	0	≈
02	4.8	+	4 509	−−	2.8	+	1 916	−−	2.0	++	595	−	3.3	++	1 376	−
03	0.4	+++	143	−	8.4	+	3 267	−−	1.1	++	378	−	32.0	++	7 869	≈
04	72.1	−	53 648	−−−	0.6	+++	458	−	32.2	++	9 974	≈	16.2	++	4 230	−
05	1.0	++	382	−	0.3	++	138	≈	6.2	+	3 695	−−	2.1	++	1 258	−
06	42.4	≈	14 895	−	0.3	+++	238	≈	1.5	++	305	−	–	≈	–	
07	34.8	+	8 570	−−	0.6	++	203	≈	2.0	++	671	−	2.7	++	1 104	−
08	0.2	+++	130	≈	0.5	+++	221	≈	1.0	++	173	≈	–	−−−	–	
09	0.2	+++	112	≈	0.6	++	300	−	1.3	++	185	≈	341.4	+	143 715	−
10	–	−−−	–		0.2	++	138	≈	–	≈	–		–	≈	–	

uk dictionary												
	15 × 15				19 × 19				21 × 21				23 × 23			
	time		nodes		time		nodes		time		nodes		time		nodes	
01	0.6	+++	108	≈	1.4	+++	200	≈	2.5	+++	163	≈	2.5	++	241	≈
02	0.6	+++	96	≈	1.3	+++	356	−	2.1	+++	196	≈	3.1	+++	420	−
03	0.6	+++	104	≈	1.6	+++	378	−	2.2	+++	194	≈	5.0	+++	886	−
04	0.4	+++	95	≈	0.7	+++	183	≈	5.0	+++	282	−	23.6	++	2 395	−−
05	0.4	+++	85	≈	0.6	+++	145	≈	3.3	++	304	−	2.9	+++	255	≈
06	1.8	+++	110	≈	0.7	+++	171	≈	1.6	+++	168	≈	23.3	++	3 696	−−
07	1.1	+++	114	≈	0.7	+++	166	≈	1.7	+++	183	≈	3.0	+++	218	≈
08	0.4	+++	122	≈	0.8	+++	154	≈	1.5	+++	188	≈	5.0	+++	379	−
09	0.5	+++	117	≈	–	−−−	–		2.1	+++	193	≈	3.3	+++	212	≈
10	0.7	+++	97	≈	0.8	+++	173	≈	6.6	+++	323	−	30.2	++	2 788	−−

Figure 21.6: Comparison of Gecode model with COMBUS

both dictionaries are the same as in [1]. For each grid size 15 × 15, 19 × 19, 21 × 21, and 23 × 23 ten different grids 01 to 10 are used. The runtime is in seconds.

For the Gecode model a timeout of 10 minutes is used, whereas a timeout of 20 minutes has been used for COMBUS. Giving the Gecode model only half the time is to cater for the difference in the hardware platform used. Orange fields are instances where neither the Gecode model nor COMBUS finds a solution (or proves that there is none) before their respective timeouts. Red fields are instances where COMBUS finds a solution but the Gecode model fails to find a solution.

Just by judging how many instances can be solved by either approach (74 for the Gecode model, 77 for COMBUS), it becomes clear that COMBUS is, as to be expected, the more robust approach. Likewise, considering the number of nodes explored during search, COMBUS shows the clear advantage of the approach taken.

On the other hand, in most cases the Gecode model can explore more than two orders of magnitude more nodes and always at least one order of magnitude more nodes per second than COMBUS. This difference in efficiency explains why the simple Gecode model can solve that many instances at all.

Moreover, one needs to consider the modeling and programming effort. The Gecode model is straightforward, does have considerably less than 100 lines of code (excluding grid specifications and dictionary support), and can be programmed in a few hours. One can expect that designing and programming a powerful problem-specific solver such as COMBUS requires considerably more time and expertise.

Using restarts. To improve the robustness of search, one can use restart-based search and no-goods from restarts with Gecode, see Section 9.4 and Section 9.5. The instances are run using a geometric cutoff sequence with base 1.5 and a scale-factor 500, a decay-factor of 0.99 for AFC (see Section 8.5.2), and no-goods depth limit of 256. The results are shown in Figure 21.7. The number raised to the number of nodes shows how many restarts have been carried out during search. Note that the choice of parameters is standard and in no way optimized for the problem at hand. The reason to use decay for AFC is to gradually change the AFC information for restarts.

Now Gecode can solve exactly the same instances as COMBUS and for all but one with at least the same efficiency. None of the instances that neither COMBUS nor Gecode could solve were helped by restart-based search though, even when the timeout was increased to one hour.

Solve the rest. Even the remaining three `word` instances can be solved within one day of runtime, the results are shown in Figure 21.8. The runtime is in the format hours:minutes:seconds (measured only by a single run).

A solution for `words-21×21-10` is shown in Figure 21.9. A solution for `words-23×23-06` is shown in Figure 21.10. Note that `words-23×23-10` does in fact not have a solution.

words dictionary								
	15 × 15		19 × 19		21 × 21		23 × 23	
	time	nodes	time	nodes	time	nodes	time	nodes
01	1.6 ++	922 --	0.5 +++	145 ≈	70.5 ≈	15 381[5] --	0.0 +++	0 ≈
02	3.3 +	1 327[1] --	1.9 ++	1 043 --	1.3 +++	341 -	2.3 ++	439 -
03	0.4 +++	144 -	68.9 -	22 754[6] --	1.1 ++	319 -	235.8 ≈	69 740[8] -
04	11.7 ≈	4 009[2] --	0.6 +++	353 -	67.9 ++	14 361[5] ≈	316.4 ≈	118 150[10] --
05	0.5 ++	182 -	0.3 ++	138 ≈	27.6 +	10 812[4] --	3.9 ++	1 514[1] -
06	413.0 -	87 391[9] --	0.3 +++	223 ≈	1.0 ++	343 -	– ≈	–
07	17.8 +	3 120[2] --	0.6 ++	245 -	2.0 ++	983 -	7.8 ++	3 238[2] --
08	0.2 +++	130 ≈	0.6 +++	153 ≈	1.1 ++	181 ≈	263.4 ≈	100 418[9] --
09	0.3 ++	114 ≈	0.6 ++	295 -	1.3 ++	184 ≈	379.1 ≈	111 121[9] -
10	45.4 +	14 232[5] --	0.3 ++	138 ≈	– ≈	–	– ≈	–

uk dictionary								
	15 × 15		19 × 19		21 × 21		23 × 23	
	time	nodes	time	nodes	time	nodes	time	nodes
01	0.6 +++	103 ≈	1.5 +++	198 ≈	2.6 +++	162 ≈	2.5 ++	234 ≈
02	0.6 +++	96 ≈	1.4 +++	308 -	2.2 +++	195 ≈	2.9 +++	366 -
03	0.7 +++	104 ≈	1.7 +++	219 -	2.3 +++	194 ≈	2.8 +++	264 ≈
04	0.5 +++	91 ≈	0.8 +++	181 ≈	4.0 +++	479 -	6.5 +++	613 -
05	0.4 +++	85 ≈	0.6 +++	145 ≈	3.4 ++	285 -	3.0 +++	258 ≈
06	1.9 +++	113 ≈	0.8 +++	171 ≈	1.7 +++	168 ≈	123.2 +	11 361[4] --
07	1.2 +++	115 ≈	0.8 +++	166 ≈	1.9 +++	176 ≈	3.1 +++	228 ≈
08	0.5 +++	108 ≈	0.9 +++	154 ≈	1.6 +++	188 ≈	4.9 +++	408 -
09	0.5 +++	116 ≈	13.9 ++	21 655[6] ---	2.3 +++	193 ≈	3.1 +++	210 ≈
10	0.8 +++	94 ≈	0.8 +++	173 ≈	6.6 +++	365 -	367.7 ≈	32 372[6] ---

Figure 21.7: Comparison of Gecode model using restarts with COMBUS

instance	time	nodes	restarts
words-nogoods-21×21-10	13 : 02 : 35	12 248 307	21
words-nogoods-23×23-06	19 : 18 : 53	10 673 073	21
words-nogoods-23×23-10	11 : 54 : 17	4 468 253	19

Figure 21.8: Results for some hard words dictionary instances

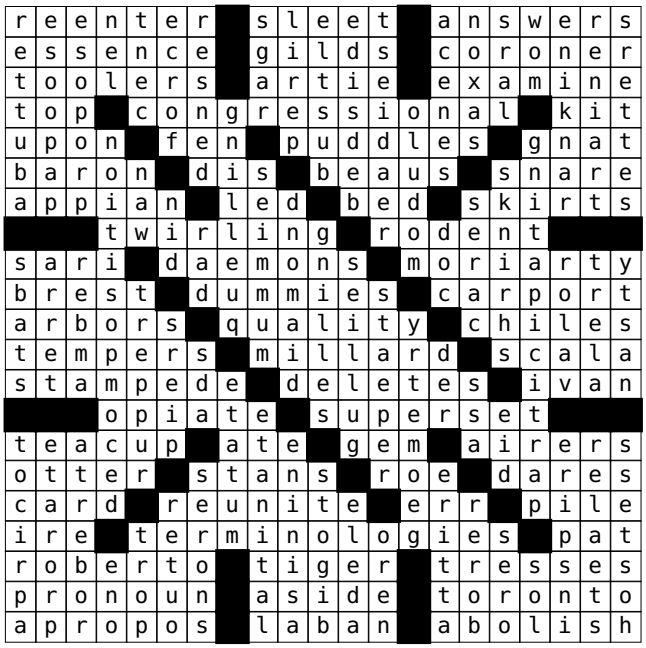


Figure 21.9: Solution for instance `words-21×21-10`

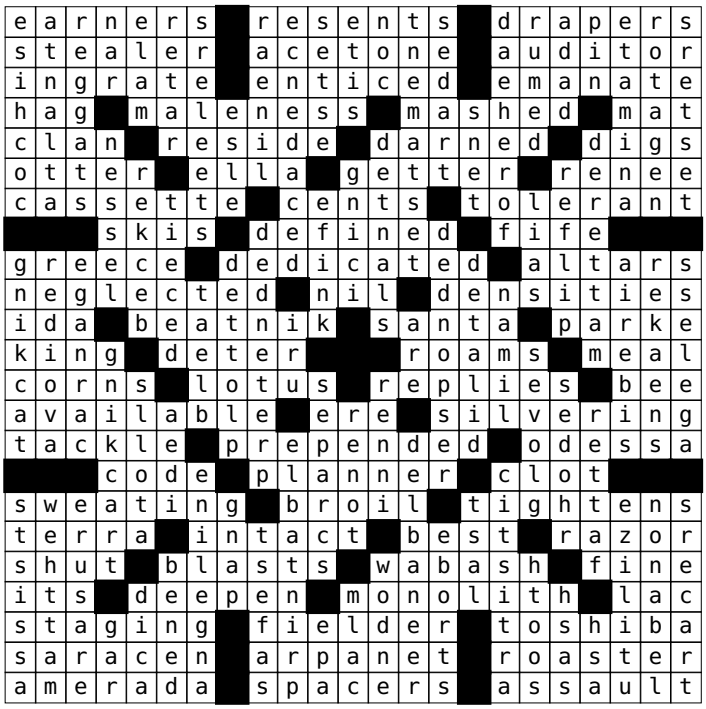
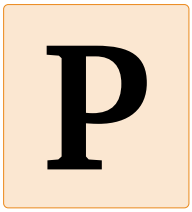


Figure 21.10: Solution for instance `words-23×23-06`

Acknowledgments. We are grateful to Peter Van Beek for access to example grids and to Adi Botea for providing us with the dictionaries `words` and `uk` from [1].



Programming propagators

Christian Schulte, Guido Tack

This part explains how to program propagators as implementations of constraints.

Basic material. Chapter 22 (Getting started) shows how to implement simple propagators for simple constraints over integer variables. It introduces the basic concepts and techniques that are necessary for any propagator.

Programming techniques. The bulk of this part describes a wide range of techniques for programming efficient propagators:

- Chapter 23 (Avoiding execution) discusses techniques for avoiding propagator execution. The examples used in this chapter introduce view arrays for propagators and Boolean views.
- Chapter 24 (Reification and rewriting) discusses how to implement propagators for reified constraints and how to optimize constraint propagation by propagator rewriting.
- Chapter 25 (Domain propagation) explains various programming techniques for propagators that perform domain propagation. The chapter also describes modification event deltas as information available to propagators and staging as a technique for speeding up domain propagation.
- Chapter 26 (Advisors) is concerned with advisors for efficient incremental propagation. Advisors can be used to provide information to a propagator which of its views have changed and how they have changed.
- Chapter 27 (Views) shows how to straightforwardly and efficiently reuse propagators for implementing several different constraints by using views. As it comes to importance, this chapter ranks second after Chapter 22. However, it comes rather late to be able to draw on the example propagators presented in the previous chapters.

Overview material. The following chapters summarize or provide an overview of topics related to programming propagators:

- Chapter 28 (Propagators for set constraints) summarizes how to implement propagators for constraints over set variables.
- Chapter 29 (Propagators for float constraints) summarizes how to implement propagators for constraints over float variables.
- Chapter 30 (Managing memory) provides an overview of memory management for propagators (and branchers, see Part B).

22 Getting started

This chapter shows how to implement simple propagators for simple constraints over integer variables. It introduces the basic concepts and techniques that are necessary for any propagator.

Here, and in the following chapters, the focus is on propagators over integer and Boolean variables. Part of the concepts introduced are specific to integer and Boolean propagators, however the techniques how to program efficient propagators are largely orthogonal to the type of variables. In Chapter 28 and Chapter 29, the corresponding concepts for set and float variables are presented.

Important. This chapter introduces concepts and techniques step-by-step, starting with a naive and inefficient first version of a propagator that then is stepwise refined. Even if you feel compelled to start programming right after you have seen the first, naive variant, you should very definitely read on until having read the entire chapter.

Overview. The first three sections set the stage for programming propagators. Section 22.1 sketches how propagators perform constraint propagation and is followed by an overview of some useful background reading material (Section 22.2). Section 22.3 provides an overview of what needs to be implemented for a constraint followed by the first naive implementation of a simple constraint (Section 22.4). The naive implementation is improved in Section 22.5 by both taking advantage of some predefined abstractions in Gecode and straightforward optimizations. This is followed by a discussion of propagation conditions as a further optimization to avoid redundant propagator executions (Section 22.6). The next section (Section 22.7) presents a first reasonable propagator that takes advantage of predefined patterns to cut down on programming effort. The last two sections discuss the obligations a propagator must meet (Section 22.8) and how some of these obligations can be waived by a propagator (Section 22.9).

22.1 Constraint propagation in a nutshell

Constraints and variables are used for modeling constraint problems. However, the only reason for actually modeling a problem is to be able to solve it with constraint propagation by removing values from variables that are in conflict with a constraint. In order to

implement constraint propagation, a constraint (typically) requires a *propagator* (or several propagators) as its implementation.

Views versus variables. The essence of a propagator is to remove values from variables that are in conflict with the constraint the propagator implements. However, a propagator does not use variables directly as they only offer operations for accessing but not removing values. Instead, a propagator uses *variable views* (or just views) as they offer operations for both value access and removal.

Views and variables have a simple relationship in that they both offer interfaces to variable implementations. When a variable is created for modeling, also a *variable implementation* is created. The variable serves as a read-only interface to the variable implementation. A view can be initialized from a variable: the view becomes just another interface to the variable's variable implementation. For more information on the relationship between variables, views, and variable implementations see Part V.

In the following we often refer to the variables of a propagator as the variable implementations that the propagator refers to through views. That is, we will often not distinguish between variables, views, and variable implementations. There is little risk of confusion as propagators always compute with views and variable implementations are never exposed for programming propagators. Much more on the relationship between variables, views, and variable implementations can be found in Part V.

By the *domain* of a variable (or a view, or a variable implementation) we refer to the set of values the variable still can take. We will often use notation such as $x \in \{1, 2, 5\}$ which means that the domain of x is $\{1, 2, 5\}$.

Executing propagators. A propagator is implemented in Gecode as a subclass of the class `Propagator` where the different tasks a propagator must be able to perform are implemented as virtual member functions. Before we describe these functions and their purpose, we sketch how a propagator actually performs constraint propagation.

As mentioned above, a propagator has several views on which it performs constraint propagation according to the constraint it implements. Like variables in modeling, propagators and views (more precisely, their variable implementations) belong to a *home space* (or just *home*). When a propagator is created, it *subscribes* to some of its views: subscriptions control the execution of a propagator. As soon as a view changes (the only way how a view can change is that some of its values are removed), all propagators that are subscribed to the view are *scheduled* for execution. Actually, subscriptions offer additional control in that only certain types of value removals schedule a propagator, this is discussed in Section 22.6. A propagator that is not scheduled, is called *idle*. Scheduling and execution are sketched in Figure 22.1.

Eventually, the space chooses a scheduled propagator for execution and executes it by running the `propagate()` member function of the propagator. This function possibly removes values and by this might schedule more propagators for eventual execution (possibly re-scheduling the currently executing propagator itself). Besides removing values and

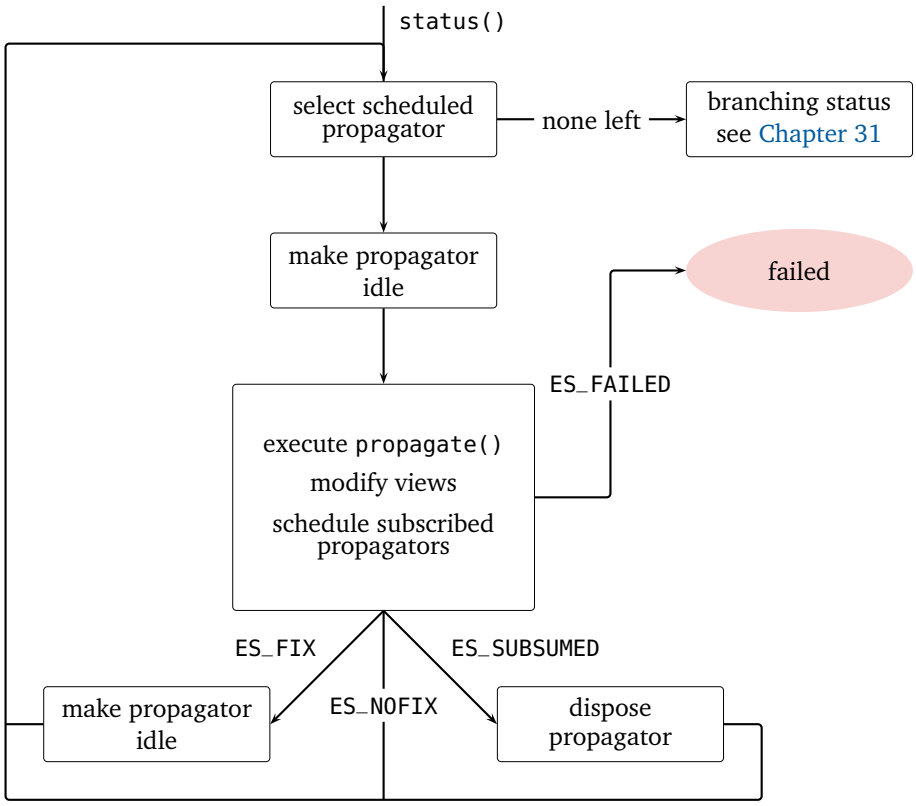


Figure 22.1: Scheduling and executing propagators

scheduling propagators, the `propagate()` member function reports about propagation. The details of what can be reported by a propagator are detailed in the following sections, but a particularly important report is: the propagator reports failure as it found out that the constraint it implements is unsatisfiable with the values left for the propagator's views (this is described by returning the value `ES_FAILED`). The remaining return values are discussed in Section 22.4 and Section 22.5.

To get the entire propagation process started, some but not necessary all propagators are scheduled as soon as they are created. Which propagators are scheduled initially is discussed in detail in Section 22.6.

Propagation is interleaved in that a space executes only one propagator at a time. The process of choosing a scheduled propagator and executing it is repeated by the space until no more propagators are available for execution. If no more propagation is possible, a space is called *stable*. This process implements constraint propagation and must be explicitly triggered by executing the `status()` member function of a space (please consult Tip 2.2). The `status()` function is typically invoked by a search engine (see Part S for details).

Space and propagator fixpoints. We often refer to the fact that no more propagation is possible for a space by saying that the space is at *fixpoint*. Likewise, we say that a propagator that cannot remove any more values is at *fixpoint*. The term fixpoint is intuitive when one looks at typical models for constraint propagation: propagators are modeled as functions that take variables and their values as input and output, often referred to as stores or domains. A domain that is a fixpoint means that input and output for a propagator are the same and hence the propagator did not perform any propagation.

22.2 Background reading

This document does not present any formal or mathematical model of how propagation is organized and which properties propagation has. There are numerous publications that do that in more detail than is possible in this document:

- A general overview of how a constraint programming system works can be found in [45].
- Realistic models that present many ideas that have been developed in the context of Gecode can be found in [48], [46], and [56].
- A truly general model for propagation that is used in Gecode is introduced in [50]. This publication is rather specialized and the generality of the model will be only briefly discussed later (see Section 22.8 and Section 40.2).

It is highly recommended to read about the general setup of constraint propagation in one of these papers. For more advanced ideas, we often refer to certain sections in the above publications or to further publications. Remember, one of the advantages of Gecode

is that many of its key ideas have been introduced by Gecode and are backed by academic publications.

22.3 What to implement?

Our first propagator implements the `less` constraint $x < y$ for two integer variables x and y.

Constraint post functions. Before discussing what a propagator must do in detail, we need to discuss how to implement the function for our `less` constraint that can be used for modeling. As known from Part M, the function should have a declaration such as

```
void less(Space& home, IntVar x0, IntVar x1);
```

We call a function implementing a constraint a *constraint post function*. The constraint post function `less()` takes a space `home` where to post the constraint (actually, where to post the propagator implementing the constraint) and variables `x0` and `x1`.[1] The responsibilities of a constraint post function are straightforward: it checks whether its arguments are valid (and throws an exception otherwise), creates variable views for the variables passed to it, and then posts an appropriate propagator. This is detailed below.

Tip 22.1 (Variables and views are passed by value)**.** As discussed before, variables and views are nothing but different interfaces to variable implementations. Their implementations are hence strikingly simple: they just carry a pointer to a variable implementation and their member functions just call the appropriate member functions of the variable implementation (some views might store an additional integer as well). For that reason, it is sufficient to pass variables and views simply by value rather than by reference. ◀

What a propagator must do. The previous section focused on the execution of a propagator that then prunes variables. While performing propagation is typically the most involved part of a propagator, a propagator must also handle several other tasks: how to post and initialize it, how to dispose it, how to copy it during cloning for search, and when to execute it. How propagation is organized in Gecode is sketched in Figure 22.2, this paragraph will fill in the missing details.

posting A constraint post function typically initiates the posting of one or several propagators. Posting a propagator is organized into two steps. The first step is implemented by a *propagator post function*, while the second is implemented by the propagator's constructor.

Typical tasks in a propagator post function are as follows:

[1] As we will discuss later (and as you might have noticed when modeling with Gecode, see Tip 2.1), a constraint post function takes a value of class `Home` instead of `Space&`. A value of type `Home` includes a reference to a space together with potentially additional information useful for posting. This aspect is discussed in Section 22.5 in more detail.

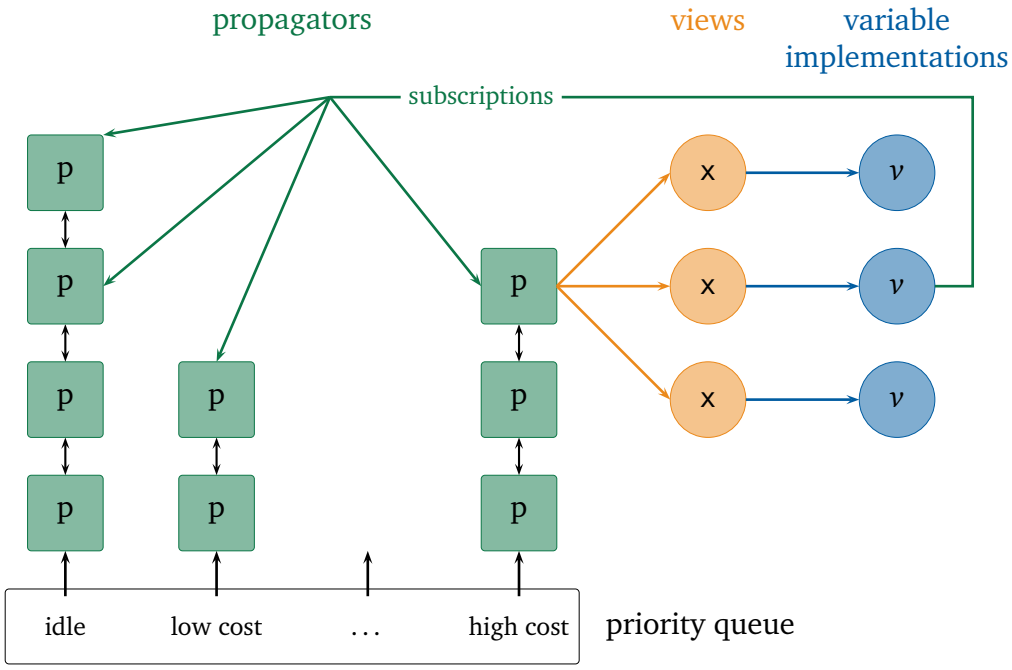


Figure 22.2: Propagators, views, and variable implementations

- Decide whether the propagator really needs to be posted.
- Decide whether a related, simpler, and hence more efficient propagator should be posted instead.
- Enforce certain invariants the propagator might require (for example, restricting the values of variables so that the propagator becomes simpler).
- Perform some initial propagation such that variable domains become reasonably small (for a discussion why small variable domains are useful, see Tip 4.3).
- Last but definitely not least, create an instance of the propagator.

The reason why posting requires a constraint post function as well as a propagator post function is to separate concerns. We will see later that the propagator post function is typically being reused by several constraints and also for propagator rewriting, an important technique which is discussed in Chapter 24. The post function of a propagator is conveniently implemented as a static member function `post()` of the propagator's class.

The propagator's constructor initializes the propagator and performs another essential task: it creates *subscriptions* to views for the propagator. Only if a propagator *subscribes* to a view (together with a propagation condition which is ignored for the moment and discussed later), the propagator is scheduled for execution whenever the domain of the view changes. Subscribing also automatically schedules the propagator for execution if needed (detailed in Section 22.6).

disposal Gecode does not automatically garbage collect propagators.[2] Propagators must be explicitly disposed. When a propagator is disposed it must also explicitly *cancel* its subscriptions (the subscriptions the propagator created in the constructor).

The only exception to this rule is that subscriptions on assigned variables do not need to be canceled. In fact, as an optimization, assigned variables do not maintain subscriptions: subscribing to an assigned variable schedules the propagator and canceling a subscription on an assigned variable does nothing. Disposal must also free other resources currently used by the propagator.

Propagator disposal is implemented by a virtual member function `dispose()`. A propagator has no destructor, the dispose function assumes this role instead. The reason to have a `dispose()` function rather than a destructor is that disposal requires the home space of a propagator which is passed as an argument to the `dispose()` function (destructors cannot take arguments in C++).

Disposal might be triggered by the propagator itself (in fact, this is the most common case) as we will see later. It is important to understand that when the space of a propagator is deleted, its propagators are *not* being disposed by default. A propagator can explicitly request to be disposed when its home is deleted (see Section 22.8). A typical case where this is needed is when the propagator has allocated memory that is not managed by the space being deleted. In Chapter 30, memory management is discussed in detail.

copying Copying for propagators works exactly as does copying for spaces: a virtual `copy()` member function returns a copy of a propagator during cloning and a copy constructor is in charge of copying and updating the propagator's data structures (in particular, its views).

cost computation Scheduling a propagator guarantees that it is executed eventually but it does not specify when. When a propagator is executed, is defined by its *cost*. The cheaper it is to execute the propagator, the earlier the propagator should be executed. This is based on the intuition that a cheaper propagator might either already fail a space and hence no expensive propagators must be executed, or that a cheaper propagator might perform propagation from which a more expensive propagator can take advantage.

As to be expected, every propagator must implement a virtual `cost()` member function. This member function is called when the propagator is scheduled. In fact, the `cost()` function might be called several times when certain information (so-called modification event deltas) for a propagator changes. We postpone a discussion of the details to Section 25.5.

The cost of executing a propagator is just an approximation that helps propagation in Gecode to be fast and also to prevent pathological behavior. Propagators of same cost

[2] One of the key design principles of Gecode is: *no magic!* Whatever needs to be done must be done explicitly.

are executed according to a first scheduled, first executed policy (basically, scheduling is organized fairly into queues of propagators with similar cost). However, Gecode does not give hard guarantees on the order of propagator execution: the cost of a propagator should be understood as a suggestion and not a requirement.

For the interested reader, the design of cost-based scheduling for Gecode together with its evaluation can be found in [48, Section 6].

propagation The core of a propagator is how it performs propagation by removing values from its views that are in conflict with the constraint it implements. This is implemented by a virtual member function `propagate()` that returns an *execution status*.

The execution status returned by the `propagate()` function must capture several important aspects:

- As mentioned before, a propagator must report whether failure occurred by returning an appropriate value for the execution status. The propagator can either find out by some propagation rules that failure must be reported. Or, when it attempts to modify view domains, the *modification operation* reports that it failed (a so-called *domain wipe-out* occurred, as all values would have been removed). Modification operations are also called *tell operations*.
- A propagator must report when the propagator has become *subsumed* (also known as *entailed*): that is, when the propagator can never ever again perform any propagation and should be disposed.

  The requirement to report subsumption is rather weak in that a propagator must at the very latest report subsumption if all of its views are assigned (of course, it might report subsumption earlier, as will be discussed in Section 22.5). With other words, a propagator must *always* report subsumption but can wait until all its views are assigned (unless it reports failure, of course).
- A propagator can characterize what it actually has computed: either a fixpoint for itself or not. We ignore this aspect for the time being and return to it in Section 22.5 and continue this discussion in Section 23.1.

Obligations of a propagator. What becomes quite clear is that a propagator has to meet certain obligations (disposing subscriptions, detecting failure, detecting subsumption, and so on). Some obligations must be met in order to comply with Gecode's requirements of a well-behaved propagator, other obligations must be met so that a propagator becomes a faithful implementation of a constraint. Section 22.8 provides an overview of all obligations a propagator must meet.

22.4 Implementing the `less` constraint

Figure 22.3 shows the class definition `Less` for our less propagator and the definition of the constraint post function. Unsurprisingly, the propagator `Less` uses two views for integer

LESS ≡ [DOWNLOAD]

```
#include <gecode/int.hh>

using namespace Gecode;

class Less : public Propagator {
protected:
  Int::IntView x0, x1;
public:
  ▶ POSTING
  ▶ DISPOSAL
  ▶ COPYING
  ▶ COST COMPUTATION
  ▶ PROPAGATION
};

void less(Space& home, IntVar x0, IntVar x1) {
  ▶ CONSTRAINT POST FUNCTION
}
```

Figure 22.3: A constraint and propagator for `less`

variables of type `Int::IntView` and propagates that the values for `x0` must be less than the values for `x1`.

The `Less` propagator inherits from the class `Propagator` defined by the Gecode kernel (as any propagator must do) and stores two integer views which are defined by Gecode's integer module. Hence, we need to include `<gecode/int.hh>`. Note that only the constraint post functions of the integer module are available in the `Gecode` namespace. All other functionality, including `Int::IntView`, is defined in the namespace `Gecode::Int`.

Constraint post function. The constraint post function is implemented as follows:

CONSTRAINT POST FUNCTION ≡
```
Int::IntView y0(x0), y1(x1);
if (Less::post(home,y0,y1) != ES_OK)
  home.fail();
```

The constraint post function creates two integer variable views `y0` and `y1` for its integer variable arguments and calls the static propagator post function as defined by the `Less` class. A propagator post function also returns an execution status of type `ExecStatus` (see Status of constraint propagation and branching commit) where the only two values that can be returned by a propagator post function are `ES_OK` (posting was successful) and `ES_FAILED` (the post function determined even without actually posting the propagator that the constraint `Less` is unsatisfiable). In case `ES_FAILED` is returned, the constraint post function *must* mark the current space `home` as failed (by using `home.fail()`).

Propagator posting. The posting of the `Less` propagator is defined by a constructor designed for initialization and a static post function returning an execution status as follows:

POSTING ≡
```
Less(Space& home, Int::IntView y0, Int::IntView y1)
  : Propagator(home), x0(y0), x1(y1) {
  x0.subscribe(home,*this,Int::PC_INT_DOM);
  x1.subscribe(home,*this,Int::PC_INT_DOM);
}
static ExecStatus post(Space& home,
                       Int::IntView x0, Int::IntView x1) {
  (void) new (home) Less(home,x0,x1);
  return ES_OK;
}
```

The constructor initializes its integer views and creates subscriptions to both `x0` and `x1`. Subscribing to an integer view takes the home space, the propagator as subscriber (as a reference), and a propagation condition of type `PropCond`. We do not look any further into propagation conditions right here (Section 22.6 does this in detail) but give two hints. First, the values for propagation conditions depend on the variable view as witnessed by the

fact that the value `Int::PC_INT_DOM` is declared in the namespace `Gecode::Int`. Second, `Int::PC_INT_DOM` creates a subscription such that the propagator is executed whenever the domain of `x0` (respectively `x1`) changes.

The propagator post function is entirely naive in that it always creates a `Less` propagator and always succeeds (and hence returns `ES_OK`). Note that propagators can only be created in a space. Hence, the **`new`** operator is used in a placement version with placement argument `(home)` which allocates memory for the `Less` propagator from `home`.

Disposal. The virtual `dispose()` function for the `Less` propagator takes a home space as argument and returns the size of the just disposed propagator (as type `size_t`).[3] Otherwise, the `dispose()` function does exactly what has been described above: it cancels the subscriptions created by the constructor used for posting:

DISPOSAL ≡
```
virtual size_t dispose(Space& home) {
  x0.cancel(home,*this,Int::PC_INT_DOM);
  x1.cancel(home,*this,Int::PC_INT_DOM);
  (void) Propagator::dispose(home);
  return sizeof(*this);
}
```

Note that the arguments for canceling a subscription are (and must be) exactly the same as the arguments for creating a subscription.

Copying. The virtual `copy()` function and the corresponding copy constructor are unsurprising in that they follow exactly the same structure as the corresponding function and constructor for spaces used for modeling:

COPYING ≡
```
Less(Space& home, bool share, Less& p)
  : Propagator(home,share,p) {
  x0.update(home,share,p.x0);
  x1.update(home,share,p.x1);
}
virtual Propagator* copy(Space& home, bool share) {
  return new (home) Less(home,share,*this);
}
```

The only aspect that deserves some attention is that a propagator must be created in a home space and hence a placement **`new`** operator is used (analogous to propagator posting).

[3] Following the discussion from above, the size of a disposed propagator must be returned explicitly, as propagators do not use destructors that implicitly handle size.

static cost functions	
`unary`	propagator with single variable view
`binary`	propagator with two variable views
`ternary`	propagator with three variable view

dynamic cost functions	
`linear`	propagator with ≈ linear complexity (or $O(n \log n)$)
`quadratic`	propagator with ≈ quadratic complexity
`cubic`	propagator with ≈ cubic complexity
`crazy`	propagator with ≈ exponential (or large polynomial) complexity

Figure 22.4: Summary of propagation cost functions

Cost computation. Cost values for propagators are defined by the class `PropCost`. The class `PropCost` defines several static member functions with which cost values can be created. For `Less`, the `cost()` function returns a cost value for a binary propagator with low cost (as we will see below, the `propagate()` function is really cheap to execute):

```
COST COMPUTATION ≡
  virtual PropCost cost(const Space&, const ModEventDelta&) const {
    return PropCost::binary(PropCost::LO);
  }
```

Please ignore the additional argument of type `ModEventDelta` to `cost()` for now, see Section 25.4.

The static member functions provided by `PropCost` are summarized in Figure 22.4. Each function takes either the value `PropCost::LO` (for low cost) or `PropCost::HI` (for high cost). The dynamic cost functions take an additional integer (or unsigned integer) value defining how many views the propagator is computing with.

For example, a propagator with `n` variables and complexity $O(n \log n)$ might return a cost value constructed by

```
  PropCost::linear(PropCost::HI,n);
```

As mentioned before, propagation cost is nothing but an approximation of the real cost of the next execution of the `propagate()` function of the propagator. The only hard fact you can rely on is that a propagator with a cost value using `PropCost::HI` is never given preference over a propagator with a cost value using `PropCost::LO`.

Propagation proper. Before starting with the code for propagation, we have to work out how the propagator should prune. This can be rather involved, leading to specialized *pruning* or *filtering* algorithms. For our `Less` propagator, the filtering rules are simple:

- All values for `x0` must be less than the largest possible value of `x1`.

`eq(home,n)`	assign to value `n`
`nq(home,n)`	remove value `n`
`le(home,n)`	restrict values to be less than `n`
`lq(home,n)`	restrict values to be less or equal than `n`
`gr(home,n)`	restrict values to be greater than `n`
`gq(home,n)`	restrict values to be greater or equal than `n`

Figure 22.5: Value-based modification functions for integer variable views

- All values for `x1` must be larger than the smallest possible value of `x0`.

These two rules can be directly implemented as follows (again, please ignore the additional argument of type `ModEventDelta` to `propagate()` for now):

PROPAGATION ≡
```
virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
  if (x0.le(home,x1.max()) == Int::ME_INT_FAILED)
    return ES_FAILED;
  if (x1.gr(home,x0.min()) == Int::ME_INT_FAILED)
    return ES_FAILED;
  if (x0.assigned() && x1.assigned())
    return home.ES_SUBSUMED(*this);
  else
    return ES_NOFIX;
}
```

The `le()` modification operation applied to an integer view `x` takes a `home` space and an integer value `n` and keeps only those values from the domain of `x` that are smaller than `n` (`gr()` is analogous). A view modification operation returns a modification event of type `ModEvent` (see Generic modification events and propagation conditions and Integer modification events and propagation conditions). A modification event describes how the domain of a view has changed, in particular the value `Int::ME_INT_FAILED` is returned if a domain wipe-out has occurred. In that case, the propagator has found out (just by attempting to perform a view modification operation) that the constraint it implements is unsatisfiable. In this case, a propagator immediately returns with the execution status `ES_FAILED`. Naturally, the member functions `min()` and `max()` of an integer view `x` just return the smallest and largest possible value of the domain of `x`.

If the propagator had not reported failure by returning `ES_FAILED` it would be faulty: it would incorrectly claim that values for the views are solutions of the constraint it implements. Being correct with respect to the constraint a propagator implements is one of the obligations of a propagator we will discuss in Section 22.8.

The modification operations for integer variable views taking a single integer value `n` as argument are listed in Figure 22.5. Integer variable views also support modification opera-

```
LESS BETTER ≡                                                    [DOWNLOAD]
  ...
  class Less : public Propagator {
    ...
  public:
    Less(Home home, Int::IntView y0, Int::IntView y1)
      ...
    }
    ▶ POSTING
    ...
    ▶ PROPAGATION
  };

  ▶ CONSTRAINT POST FUNCTION
```

Figure 22.6: A better constraint and propagator for `less`

tions that simultaneously can operate on sets of values. These operations are discussed in Chapter 25.

The second part of the propagator is concerned with deciding subsumption: if both `x0` and `x1` are assigned, the propagator executes the function `ES_SUBSUMED()`. The function `ES_SUBSUMED()` disposes the propagator by calling its `dispose()` member function and returns a value for `ExecStatus` that signals that the propagator is subsumed. After returning, the memory for the propagator will be reused. It is important to understand that subsumption is not an optimization but a requirement: at the very latest when all of its views are assigned, a propagator must report subsumption! The propagator is of course free to report subsumption earlier as is exploited in Section 22.5.

In case the propagator is neither failed nor subsumed, it reports an execution status `ES_NOFIX`. Returning the value `ES_NOFIX` means that the propagator will be scheduled if one of its views (`x0` and `x1` in our example) have been modified. If none of its views have been modified, the propagator is not scheduled. This rather naive statement of what the propagator has computed is improved in Section 22.5.

Tip 22.2 (Immediately return after subsumption)**.** As mentioned above, the function `ES_SUBSUMED()` disposes a propagator. To not crash Gecode, you must immediately return after calling `ES_SUBSUMED()`. In particular, executing view modification operations or creating subscriptions by a disposed propagator will surely crash Gecode! ◀

22.5 Improving the `Less` propagator

Figure 22.6 shows an improved implementation of the `Less` propagator together with the corresponding constraint post function `less()`. The propagator features improved posting,

improved propagation, and improved readability.

A recurring theme in improving propagators in this and in the next section is not about sophisticated propagation rules (admittedly, `Less` does not have much scope for cleverness). In contrast, all improvements will try to achieve the best of all optimizations: avoid executing the propagator in the first place!

Improving posting. As mentioned above, all functions related to posting (constructor, propagator post function, and constraint post function) should take a value of type `Home` rather than `Space&`. The improved propagator honors this without any further changes (a `Space` is automatically casted to a `Home` if needed, and vice-versa). An example of how passing a `Home` value is actually useful can be found in Section 24.2.

The improved constraint post function is as follows:

CONSTRAINT POST FUNCTION ☰
```
void less(Home home, IntVar x0, IntVar x1) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Less::post(home,x0,x1));
}
```

The constraint post function features three improvements:

- The most obvious improvement: if the `home` space is already failed, no propagator is posted.
- The variable views are initialized implicitly. Note that the propagator post function `Less::post()` is called with integer variables `x0` and `x1` which are automatically coerced to integer views (as `Int::IntView` has a non-explicit constructor with argument type `IntVar`).
- Instead of testing whether `Less::post()` returns `ES_FAILED`, the constraint post function uses the macro `GECODE_ES_FAIL` for convenience (doing exactly the same as shown before). Macros for checking and failing are summarized below.

The improved post function is a little bit more sophisticated:

POSTING ☰
```
static ExecStatus post(Home home,
                       Int::IntView x0, Int::IntView x1) {
  if (same(x0,x1))
    return ES_FAILED;
  GECODE_ME_CHECK(x0.le(home,x1.max()));
  GECODE_ME_CHECK(x1.gr(home,x0.min()));
  if (x0.max() >= x1.min())
    (void) new (home) Less(home,x0,x1);
  return ES_OK;
}
```

It includes the following improvements:

- The propagator is not posted if `x0` and `x1` happen to refer to the very same variable implementation. In that case, of course, `x0` can never be less than `x1` and `post()` can immediately report failure.
- The propagator performs one initial round of propagation already during posting. This yields small variable domains for other propagators to be posted (see Tip 4.3).
- If all values of `x0` are already less than all values of `x1` (that is, `x0.max() < x1.min()`), then the constraint that `x0` is less than `x1` already holds (it is subsumed). Hence, no propagator needs to be posted.

Improving propagation. The `propagate()` function is improved as follows:

PROPAGATION ≡
```
virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
  GECODE_ME_CHECK(x0.le(home,x1.max()));
  GECODE_ME_CHECK(x1.gr(home,x0.min()));
  if (x0.max() < x1.min())
    return home.ES_SUBSUMED(*this);
  else
    return ES_FIX;
}
```

with the following improvements:

- The checking macro `GECODE_ME_CHECK` checks whether a modification operation returns failure and then returns `ES_FAILED`.
- The propagator tries to detect subsumption early, it does not wait until both `x0` and `x1` are assigned. It uses exactly the same criterion that is used for avoiding posting of the propagator in the first place.

  This is entirely legal: a propagator can report subsumption as soon as the propagator will never propagate again (or, with other words, the propagator will always be at fixpoint). The obligation is: it must report subsumption (or failure) at the very latest when all of its views are assigned. Early subsumption means fewer propagator executions.
- If the propagator is not yet subsumed it returns `ES_FIX` instead of `ES_NOFIX`. This means that the propagator tells the Gecode kernel that what it has computed happens to be a fixpoint for itself.

  This is easy enough to see for `Less`: executing the `propagate()` function twice does not perform any propagation in the second call to `propagate()`. After all, only after `x0.min()` or `x1.max()` change, the propagator can prune again. As neither `x0.min()`

check macros	
`GECODE_ME_CHECK(me)`	Checks whether `me` signals failure and returns `ES_FAILED`.
`GECODE_ES_CHECK(es)`	Checks whether `es` signals failure or subsumption and returns `es`.

fail macros	
`GECODE_ME_FAIL(me)`	Checks whether `me` signals failure, fails `home`, and returns.
`GECODE_ES_FAIL(es)`	Checks whether `es` signals failure, fails `home`, and returns.

Figure 22.7: Check and fail macros

nor `x1.max()` change (the propagator might change `x1.min()` or `x0.max()` instead), the propagator should report that it has computed a fixpoint.

The situation where returning `ES_FIX` instead of `ES_NOFIX` differs, is when the propagator actually prunes the values for `x0` or `x1`. If the propagator returns `ES_NOFIX` it will be scheduled in this situation. If it returns `ES_FIX` it will not be scheduled. The difference between `ES_FIX` and `ES_NOFIX` is sketched in Figure 22.1. Again, not scheduling a propagator means fewer propagator executions. Chapter 23 takes a second look at fixpoint reasoning for propagators.

Check and fail macros. Check and fail macros available in Gecode are summarized in Figure 22.7. Note that a check macro can be used in a propagator post function or in a `propagate()` function, whereas a fail macro can only be used in a constraint post function. Note also that both fail macros assume that the identifier `home` refers to the current home space. For an example of how to use `GECODE_ES_CHECK`, see Section 23.3. For an example of how to use `GECODE_ME_FAIL`, see Section 23.2.

In fact, explicitly testing whether a modification event returned by a view modification operation is equal to `Int::ME_INT_FAILED` is highly discouraged. If not using `GECODE_ME_CHECK`, the proper way to test for failure is as in:

```
if (me_failed(x0.le(home,x1.max())))
  return ES_FAILED;
```

Note that both `GECODE_ME_CHECK` as well as `me_failed` work for all variable views and not only for integer variable views.

22.6 Propagation conditions

This section discusses modification events and propagation conditions in more detail. A modification event describes how a modification operation has changed a view. A propaga-

tion condition describes when a propagator is scheduled depending on how the views it is subscribed to are modified.

Modification events. Modification operations on integer views might return the following values for `ModEvent` (we assume that the view `x` has the domain $\{0, 2, 3\}$):

- `Int::ME_INT_NONE`: the view has not been changed. For example, both `x.le(home,5)` and `x.nq(home,1)` return `Int::ME_INT_NONE`.
- `Int::ME_INT_FAILED`: the values of a view have been wiped out. For example, both `x.le(home,0)` and `x.eq(home,1)` return `Int::ME_INT_FAILED`.

  Note that when a modification operation signals failure, the values of a view might change unpredictably, see also Section 4.1.7. For example, the values might entirely remain or only part of the values are removed. This also clarifies that a view (or its variable implementation) does not maintain the information that a modification operation on it failed. This also stresses that it is essential to check the modification event returned by a modification operation for failure.
- `Int::ME_INT_DOM`: an inner value (that is neither the smallest nor the largest value) has been removed. For example, `x.nq(home,2)` ($x \in \{0, 3\}$) returns `Int::ME_INT_DOM`.
- `Int::ME_INT_BND`: the smallest or largest value of a view has changed but the view has not been assigned. For example, both `x.nq(home,0)` ($x \in \{2, 3\}$) and `x.lq(home,2)` ($x \in \{0, 2\}$) return `Int::ME_INT_BND`.
- `Int::ME_INT_VAL`: the view has been assigned to a single value. For example, both `x.le(home,2)` ($x = 0$) and `x.gq(home,3)` ($x = 3$) return `Int::ME_INT_VAL`.

Propagation conditions. The propagation condition used in subscriptions determine, based on modification events, when a propagator is scheduled. Assume that a propagator `p` subscribes to the integer view `x` with one of the following propagation conditions:

- `Int::PC_INT_DOM`: whenever the view `x` is modified (that is, for modification events `Int::ME_INT_DOM`, `Int::ME_INT_BND`, and `ME_INT_VAL` on `x`), schedule `p`.
- `Int::PC_INT_BND`: whenever the bounds of `x` are modified (that is, for modification events `Int::ME_INT_BND` and `Int::ME_INT_VAL` on `x`), schedule `p`.
- `Int::PC_INT_VAL`: whenever `x` becomes assigned (that is, for modification event `Int::ME_INT_VAL` on `x`), schedule `p`.

For our `Less` propagator, the right propagation condition for both views `x0` and `x1` is of course `Int::PC_INT_BND`: the propagator can only propagate if either the lower bound of `x0` or the upper bound of `x1` changes. Otherwise, the propagator is at fixpoint. Again, the

LESS EVEN BETTER ≡ [DOWNLOAD]

```
...
class Less : public Propagator {
  ...
public:
  Less(Home home, Int::IntView y0, Int::IntView y1)
    : Propagator(home), x0(y0), x1(y1) {
    x0.subscribe(home,*this,Int::PC_INT_BND);
    x1.subscribe(home,*this,Int::PC_INT_BND);
  }
  ...
  virtual size_t dispose(Space& home) {
    x0.cancel(home,*this,Int::PC_INT_BND);
    x1.cancel(home,*this,Int::PC_INT_BND);
    (void) Propagator::dispose(home);
    return sizeof(*this);
  }
  ...
};
...
```

Figure 22.8: An even better constraint and propagator for `less`

DISEQUALITY ≡ [DOWNLOAD]

```
...
class Disequal : public Propagator {
  ...
public:
  Disequal(Home home, Int::IntView y0, Int::IntView y1)
    : Propagator(home), x0(y0), x1(y1) {
    x0.subscribe(home,*this,Int::PC_INT_VAL);
    x1.subscribe(home,*this,Int::PC_INT_VAL);
  }
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    if (x0.assigned())
      GECODE_ME_CHECK(x1.nq(home,x0.val()));
    else
      GECODE_ME_CHECK(x0.nq(home,x1.val()));
    return home.ES_SUBSUMED(*this);
  }
};
...
```

Figure 22.9: A propagator for disequality

very point of propagation conditions is to avoid executing a propagator that is known to be at fixpoint. An even better propagator for `less` using the proper propagation conditions is shown in Figure 22.8.

The idea to avoid execution depending on how variables change is well known and typically realized through so-called *events*. For an evaluation, see [48, Section 5]. Distinguishing between modification events and propagation conditions has been introduced by Gecode, for a discussion see [56, Chapter 5].

The `Less` propagator subscribes to both of its views (that is, `x0` and `x1`). For some propagators it is actually sufficient to only subscribe to some but not all of its views. Section 23.3 discusses partial and dynamically changing subscriptions for propagators.

Scheduling when posting. As mentioned earlier, a propagator also needs to be scheduled when it is created to get the process of constraint propagation started. More precisely, a propagator `p` might be scheduled when it subscribes to a view `x`. If `x` is assigned, `p` is always scheduled regardless of the propagation condition used for subscribing. If `x` is not assigned, `p` is only scheduled if the propagation condition is different from `Int::PC_INT_VAL`.

With other words: propagation conditions provide a guarantee a propagator can rely on. In particular, for the propagator condition `Int::PC_INT_VAL`, the guarantee is that the propagator is only executed if a view is assigned.

	single view patterns
`UnaryPropagator`	unary propagator with view `x0`
`BinaryPropagator`	binary propagator with views `x0` and `x1`
`TernaryPropagator`	ternary propagator with views `x0`, `x1`, and `x2`
`NaryPropagator`	*n*-ary propagator with view array `x`
`NaryOnePropagator`	*n*-ary propagator with view array `x` and view `y`
	mixed view patterns
`MixBinaryPropagator`	binary propagator with views `x0` and `x1`
`MixTernaryPropagator`	ternary propagator with views `x0`, `x1`, and `x2`
`MixNaryOnePropagator`	*n*-ary propagator with view array `x` and view `y`

Figure 22.10: Propagator patterns

Consider, for example, the implementation of the `propagate()` method for a disequality propagator shown in Figure 22.9. The propagator waits until at least `x0` or `x1` are assigned (due to the propagation condition `PC_INT_VAL`). When its `propagate()` method is executed, the propagator can exploit that at least one of the views `x0` and `x1` is assigned by testing only `x0` for assignment.

Scheduling when creating subscriptions can be avoided by giving **`false`** as an optional last argument to `subscribe()`. This is typically used when subscriptions are created during propagation, and the propagator does not need to be scheduled.

22.7 Using propagator patterns

Gecode's kernel defines common propagator patterns (see Propagator patterns) which are summarized in Figure 22.10. The single view patterns are templates that require a view as first argument and a propagation condition as second argument. The mixed view patterns require for each view or view array a view argument and a propagation condition. The mixed view patterns are useful when different types of views and/or different propagation conditions for the views are needed (see Section 27.1.2 for an example).

The propagator patterns accept also a value `Int::PC_INT_NONE` for the propagation conditions that avoid creating subscriptions at all. This comes in handy when the propagator patterns are used in situations where no subscriptions are needed, see for example Section 26.3.

The patterns define a constructor for creation (that also creates subscriptions to their views with the defined propagation conditions), a constructor for cloning, a `dispose()` member function, and a `cost()` member function where the cost value is always the `PropCost::LO` variant of the corresponding cost function (that is, `PropCost::unary()` for `UnaryPropagator`, `PropCost::linear()` for `NaryPropagator` and `NaryOnePropagator`, and so on). The integer module additionally defines patterns for reified propagators which

are discussed in Section 24.2.[4]

Figure 22.11 shows how to use the `BinaryPropagator` pattern for the `less` constraint. To give an impression of what needs to be implemented, the code for implementing the `less` constraint is shown in full. Note that one must define a propagator post function and the virtual member functions `copy()` and `propagate()`. Of course, one could also choose to overwrite the virtual member functions `dispose()` and `cost()` if needed.

22.8 Propagator obligations

A propagator has to meet three different kinds of obligations: obligations towards the constraint it implements, obligations towards the amount of propagation it performs, and obligations that are Gecode specific. Some obligations can be waived by notifying the Gecode kernel that a propagator does not comply (see Section 22.9).

Constraint implementation. A propagator must be

correct A propagator must be correct in that it never prunes values that can appear in a solution of the constraint it implements.

checking A propagator must be checking: at the very latest when all views are assigned, the propagator must decide whether the assignment is a solution of the constraint or not (in which case the propagator must report failure).

Amount of propagation. A propagator must be

contracting A propagator is only allowed to remove values. The modification operations we have been discussing so-far naturally satisfy this property. For some operations that are discussed in Section 25.2 extra carefulness is required.

monotonic By default, a propagator must be monotonic: a propagator is not allowed to perform more pruning when executed for views with more values than when executed for views with less values.

Propagators are typically monotonic unless they use randomization, approximation, or something similar. For more details see below. This obligation can be waived.

subscription complete A propagator must create sufficient subscriptions such that it is scheduled for execution when it is not at fixpoint.

fixpoint and subsumption honest A propagator is not allowed to claim that it has computed a fixpoint or is subsumed if it is not (that is, it could still propagate).

[4] As reified propagators require Boolean views, reified patterns are provided by the integer module and not by the kernel. The kernel knows nothing about the different views programmed on top of it.

LESS CONCISE ≡ [DOWNLOAD]

```
...
class Less : public BinaryPropagator<Int::IntView,Int::PC_INT_BND> {
public:
  Less(Home home, Int::IntView x0, Int::IntView x1)
    : BinaryPropagator<Int::IntView,Int::PC_INT_BND>(home,x0,x1) {}
  static ExecStatus post(Home home,
                         Int::IntView x0, Int::IntView x1) {
    if (same(x0,x1))
      return ES_FAILED;
    GECODE_ME_CHECK(x0.le(home,x1.max()));
    GECODE_ME_CHECK(x1.gr(home,x0.min()));
    if (x0.max() >= x1.min())
      (void) new (home) Less(home,x0,x1);
    return ES_OK;
  }
  Less(Space& home, bool share, Less& p)
    : BinaryPropagator<Int::IntView,Int::PC_INT_BND>(home,share,p) {}
  virtual Propagator* copy(Space& home, bool share) {
    return new (home) Less(home,share,*this);
  }
  virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
    GECODE_ME_CHECK(x0.le(home,x1.max()));
    GECODE_ME_CHECK(x1.gr(home,x0.min()));
    if (x0.max() < x1.min())
      return home.ES_SUBSUMED(*this);
    else
      return ES_FIX;
  }
};

void less(Home home, IntVar x0, IntVar x1) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Less::post(home,x0,x1));
}
```

Figure 22.11: A concise constraint and propagator for `less`

Implementation specific obligations. A propagator must be

subsumption complete At the very latest, a propagator must report subsumption if all views it has subscribed to are assigned.

external resource free By default, a propagator cannot allocate memory from any other source (or any other resource) but its home space as the `dispose()` member function is not automatically called when the propagator's home space is deleted. This obligation can be waived.

update complete All views a propagator subscribes to must be updated during cloning (that is, a propagator can only subscribe to views it actually stores).

The reason for this obligation is that a space does not know its views (and variable implementations). It only gets to know them during cloning when they are explicitly updated. Having subscription information maintained by Gecode's kernel without updating the corresponding variable implementations will leave a space after cloning in an inconsistent state (crashes are guaranteed).

cloning conservative When a propagator is copied during cloning, it is not allowed to perform any variable modification operations nor is it allowed to change its subscriptions.

subscription correct Any subscription to a view must eventually be canceled (typically in the `dispose()` member function), unless the view is assigned.

22.9 Waiving obligations

A propagator can notify its home space about some of its properties (as defined by `ActorProperty`, see Programming actors) that relate to some of the obligations mentioned in the previous section.

Weakly monotonic propagators. If a propagator `p` intends to be non-monotonic, it can notify its `home` space `home` by

```
home.notice(p,AP_WEAKLY);
```

It can revoke this notice later (it must revoke this at latest in its `dispose()` function) by

```
home.ignore(p,AP_WEAKLY);
```

This is typically done in the constructor of the propagator and means that the propagator intends to be only *weakly monotonic*: it is sufficient for the propagator to be checking and correct. For a discussion of weak monotonicity together with an example, see [50].

Calling `dispose()` during space deletion. If a propagator `p` needs to use external resources or non-space allocated memory, it must inform its `home` space during posting about this fact by:

```
home.notice(p,AP_DISPOSE);
```

This will ensure that the `dispose()` function of the propagator `p` will be called when its home space is deleted.

In its dispose function the propagator `p` must revoke this notice by

```
home.ignore(p,AP_DISPOSE);
```

For examples, see Section 26.2 and Section 19.4. In Chapter 30, memory management is discussed in detail.

23 Avoiding execution

This chapter serves two purposes. First, it discusses techniques for avoiding propagator execution. Second, the examples used in this chapter introduce view arrays for propagators and Boolean views.

Overview. Fixpoint reasoning as an important technique for avoiding propagator execution is discussed in detail in Section 23.1 (we already briefly touched on this subject in Section 22.5). The following section (Section 23.2) presents an example propagator for Boolean disjunction introducing Boolean variable views and view arrays for propagators with arbitrarily many views. The Boolean disjunction propagator is used in Section 23.3 as an example for dynamic subscriptions (a propagator only subscribes to a subset of its views and the subscriptions change while the propagator is being executed) as another technique to avoid propagator execution.

23.1 Fixpoint reasoning reconsidered

In this section, we develop a propagator for equality $x = y$ that performs bounds reasoning to further discuss how a propagator can do fixpoint reasoning. A stronger domain propagator for equality is discussed in Chapter 25. Background information on fixpoint reasoning can be found in [48, Section 4].

A naive propagator. The `propagate()` member function of an equality propagator `Equal` is shown in Figure 23.1. The remaining functions are as to be expected.

The propagation rules implemented by `Equal` is that the values of both `x0` and `x1` must be greater or equal than `std::max(x0.min(),x1.min())` and less or equal than `std::min(x0.max(),x1.max())`. The above implementation follows these rules even though it avoids the computation of `std::min` and `std::max`. Let us look to the adjustment of the lower bounds of `x0` and `x1` (the upper bounds are analogous):

- If `x0.min()==x1.min()`, nothing is pruned.
- If `x0.min()<x1.min()`, then `x0` is pruned.
- If `x0.min()>x1.min()`, then `x1` is pruned.

EQUAL NAIVE ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_BND> {
public:
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    GECODE_ME_CHECK(x0.gq(home,x1.min()));
    GECODE_ME_CHECK(x1.gq(home,x0.min()));
    GECODE_ME_CHECK(x0.lq(home,x1.max()));
    GECODE_ME_CHECK(x1.lq(home,x0.max()));
    if (x0.assigned() && x1.assigned())
      return home.ES_SUBSUMED(*this);
    else
      return ES_NOFIX;
  }
};
...
```

Figure 23.1: A naive equality bounds propagator

As can be seen, the propagator does not perform any fixpoint reasoning, it always returns `ES_NOFIX` (unless it is subsumed or failed).

The propagator might actually sometimes compute a fixpoint and sometimes not. Consider $\texttt{x0} \in \{1,2,3,4\}$ and $\texttt{x1} \in \{0,1,2,5\}$. Then `Equal` propagates that $\texttt{x0} \in \{1,2,3,4\}$ and $\texttt{x1} \in \{1,2\}$ which happens to be not a fixpoint. The reason (which is common when performing bounds propagation) is that a bounds modification (here `x1.lq(home,4)`) has resulted in an even smaller upper bound (or an even larger lower bound when the lower bound is modified). In a way, the modification operation updating the bound fell into a hole in the variable domain.

Reporting fixpoints. The above example shows that the equality propagator computes a fixpoint if and only if, after propagation, `x0.min()==x1.min()` and `x0.max()==x1.max()`. Figure 23.2 shows the `propagate()` member function of an improved `Equal` propagator that takes advantage of fixpoint reasoning.

An idempotent propagator. The previous propagator reports when it computes a fixpoint. However, we can change any propagator so that it always computes a fixpoint. Propagators which always compute a fixpoint (unless they are subsumed) are known as *idempotent* propagators. The idea to turn an arbitrary propagator into an idempotent propagator is simple: repeat propagation until it has computed a fixpoint. Figure 23.3 shows the `propagate()` member function of an idempotent equality propagator.

EQUAL ≡ [DOWNLOAD]

```
...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    GECODE_ME_CHECK(x0.gq(home,x1.min()));
    GECODE_ME_CHECK(x1.gq(home,x0.min()));
    GECODE_ME_CHECK(x0.lq(home,x1.max()));
    GECODE_ME_CHECK(x1.lq(home,x0.max()));
    if (x0.assigned() && x1.assigned())
      return home.ES_SUBSUMED(*this);
    else if ((x0.min() == x1.min()) &&
             (x0.max() == x1.max()))
      return ES_FIX;
    else
      return ES_NOFIX;
  }
...
```

Figure 23.2: An equality bounds propagator with fixpoint reasoning

EQUAL IDEMPOTENT ≡ [DOWNLOAD]

```
...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    do {
      GECODE_ME_CHECK(x0.gq(home,x1.min()));
      GECODE_ME_CHECK(x1.gq(home,x0.min()));
    } while (x0.min() != x1.min());
    do {
      GECODE_ME_CHECK(x0.lq(home,x1.max()));
      GECODE_ME_CHECK(x1.lq(home,x0.max()));
    } while (x0.max() != x1.max());
    if (x0.assigned() && x1.assigned())
      return home.ES_SUBSUMED(*this);
    else
      return ES_FIX;
  }
...
```

Figure 23.3: An idempotent equality bounds propagator

EQUAL IDEMPOTENT USING MODIFICATION EVENTS ≡ [DOWNLOAD]

```
...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    bool nafp = true;
    while (nafp) {
      nafp = false;
      ModEvent me = x0.gq(home,x1.min());
      if (me_failed(me))
        return ES_FAILED;
      else if (me_modified(me))
        nafp = true;
      ...
    }
    ...
  }
...
```

Figure 23.4: An idempotent equality bounds propagator using modification events

The idempotent propagator always computes a fixpoint. That means that it does not need to use the generic mechanisms provided by the Gecode kernel for scheduling and executing propagators in order to compute a fixpoint. Instead, a tight inner loop inside the `propagate()` function computes the fixpoint. If the propagator is cheap (which it is in our example) it might be better to make the propagator idempotent and hence avoid the overhead of the Gecode kernel (even though the Gecode kernel is very efficient as it comes to scheduling and executing propagators).

An idempotent propagator using modification events. The idempotent propagator shown above tests a propagator-specific criterion to determine whether a fixpoint has been computed. With the help of modification events there is a generic approach to computing a fixpoint within the `propagate()` member function of a propagator. Figure 23.4 shows the `propagate()` member function where the Boolean variable `nafp` (for: not at fixpoint) tracks whether the propagator has computed a fixpoint. The function `me_modified(me)` checks whether the modification event `me` does not signal failure or that the view did change (that is, for integer views, `me` is different from `Int::ME_INT_FAILED` and `Int::ME_INT_NONE`). Whenever a view is modified, `nafp` is accordingly set to **`true`**. The remaining modification operations are omitted as they are analogous.

Tip 23.1 (Understanding `ES_NOFIX`)**.** The above technique for making a propagator idempotent is based on the idea of repeating propagation until no more views are modified.

Unfortunately, we have seen (more than once) a similar but not very good idea in propagators for finding out whether a propagator is at fixpoint. The idea can be sketched as

follows: record in a Boolean flag `modified` whether a modification operation actually modified a view during propagation. For our bounds equality propagator the idea can be sketched as follows:

```
virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
  bool modified = false;

  ModEvent me = x0.gq(home,x1.min());
  if (me_failed(me))
    return ES_FAILED;
  else if (me_modified(me))
    modified = true;
  ...
  if (x0.assigned() && x1.assigned())
    return home.ES_SUBSUMED(*this);
  else
    return modified ? ES_NOFIX : ES_FIX;
}
```

That means, if `modified` is **`true`** the propagator might not be at fixpoint and hence `ES_NOFIX` is returned. This makes not much sense: `ES_NOFIX` means *that a propagator is not considered to be at fixpoint if it has modified a view!* If no view has been modified the propagator must be at fixpoint and just returning `ES_NOFIX` does the trick.

This not-so-hot idea can be summarized as: achieving nothing with quite some effort! ◀

For the equality constraint, checking the propagator-specific fixpoint condition is simple enough. For more involved propagators, the generic approach using modification events always works and is to be preferred. As the generic approach is so useful, a macro `GECODE_ME_CHECK_MODIFIED` is available. With this macro, the loop insided the `propagate()` function can be expressed as:

```
while (nafp) {
  nafp = false;
  GECODE_ME_CHECK_MODIFIED(nafp,x0.gq(home,x1.min()));
  GECODE_ME_CHECK_MODIFIED(nafp,x1.gq(home,x0.min()));
  GECODE_ME_CHECK_MODIFIED(nafp,x0.lq(home,x1.max()));
  GECODE_ME_CHECK_MODIFIED(nafp,x1.lq(home,x0.max()));
}
```

23.2 A Boolean disjunction propagator

Before we demonstrate dynamic subscriptions in the next section, we discuss a propagator for Boolean disjunction. The propagator is rather simple, but we use it as an example for Boolean views, arrays of views, and several other aspects.

OR TRUE ≡ [DOWNLOAD]

```
...
class OrTrue : public Propagator {
protected:
  ViewArray<Int::BoolView> x;
public:
  OrTrue(Home home, ViewArray<Int::BoolView>& y)
    : Propagator(home), x(y) {
    x.subscribe(home,*this,Int::PC_BOOL_VAL);
  }
  ...
  ▶ PROPAGATION
};

void dis(Home home, const BoolVarArgs& x, int n) {
  if ((n != 0) && (n != 1))
    throw Int::NotZeroOne("dis");
  if (home.failed()) return;
  if (n == 0) {
    for (int i=x.size(); i--; )
      GECODE_ME_FAIL(Int::BoolView(x[i]).zero(home));
  } else {
    ViewArray<Int::BoolView> y(home,x);
    GECODE_ES_FAIL(OrTrue::post(home,y));
  }
}
```

Figure 23.5: Naive Boolean disjunction

Figure 23.5 shows the `OrTrue` propagator that propagates that an array of Boolean views `x` is `1` (that is, the Boolean disjunction on `x` is true). That is, at least one of the views in `x` must be `1`. The propagator uses an array `ViewArray` of Boolean views (a `ViewArray` is generic with respect to the views it stores). Similar to views, view arrays have `subscribe()` and `cancel()` functions for subscriptions, where the operations are applied to all views in the view array.

Tip 23.2 (View arrays also provide STL-style iterators)**.** View arrays also support STL-style (Standard Template Library) iterators, similar to other arrays provided by Gecode, see Section 4.2.3. ◀

Constraint post function. The constraint post function `dis()` constrains the disjunction of the Boolean variables in `x` to be equal to the integer `n`:

$$\bigvee_{i=0}^{|x|-1} x_i = n$$

The constraint post function checks that the value for `n` is legal. If `n` is neither `0` nor `1`, the constraint post function throws an exception. Here, we use an appropriate Gecode-defined exception, but any exception of course works.

If `n` is `0`, all variables in `x` must be `0` as well: rather than posting a propagator to do that, we assign the views immediately in the constraint post function. Note that we have to get an integer view to be able to assign `x[i]` to zero as only views provide modification operations.

Otherwise, a view array `y` is created (newly allocated in the space `home`) and its fields are initialized to views that correspond to the respective integer variables in `x`. Then, posting the `OrTrue` propagator is as expected.

A more general case of Boolean disjunction, where `n` is an integer variable instead of an integer value is discussed in Section 26.3.

Propagation. Propagation for Boolean disjunction is straightforward: first, all views are inspected whether they are assigned to `1` (in which case the propagator is subsumed) or to `0` (in which case the assigned view is eliminated from the view array). If no views remain, the propagator is failed. If a single (by elimination, an unassigned view) remains, it is assigned to `1`. This can be implemented as shown in Figure 23.6.

The operation `x.move_lst(i)` of a view array `x` moves the last element of `x` to position `i` and shrinks the array by one element. Shrinking from the end is in particular simple if the array elements are iterated backwards as in our example. The class `ViewArray` provides several operations to shrink view arrays. Note that `x.move_lst(i)` requires that the view at position `i` is actually assigned or has no subscription, otherwise the subscription for `x[i]` needs to be canceled before `x[i]` is overwritten. View arrays also provide operations for simultaneously moving elements and canceling subscriptions.

```
PROPAGATION ≡
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    for (int i=x.size(); i--; )
      if (x[i].one())
        return home.ES_SUBSUMED(*this);
      else if (x[i].zero())
        x.move_lst(i);
    if (x.size() == 0)
      return ES_FAILED;
    if (x.size() == 1) {
      GECODE_ME_CHECK(x[0].one(home));
      return home.ES_SUBSUMED(*this);
    }
    return ES_FIX;
  }
```

Figure 23.6: Propagation for naive Boolean disjunction

Tip 23.3 (View arrays have non-copying copy constructors)**.** The constructor for posting in Figure 23.5 uses the copy constructor of `ViewArray` to initialize the member `x`. The copy constructor does *not* copy a view array. Instead, after its execution both original and copy have shared access to the same views. ◀

Using propagator patterns. How to use a propagator pattern with an array of views is shown in Figure 23.7 for Boolean disjunction.

Boolean views. A Boolean view `Int::BoolView` provides operations for testing whether it is assigned to `1` (`one()`) or `0` (`zero()`), or whether it is not assigned yet (`none()`). As modification operations Boolean views offer `one(home)` and `zero(home)`. Also, Boolean views only support `PC_BOOL_NONE` and `PC_BOOL_VAL` as propagation conditions and `ME_BOOL_NONE`, `ME_BOOL_FAILED`, and `ME_BOOL_VAL` as modification events.

Boolean views (variables and variable implementations likewise) are not related to integer views by design: the very point is that Boolean variable implementations have a specially optimized implementation that is in fact not related to the implementation of integer variables.

Boolean views also implement all operations that integer views implement and integer propagation conditions can also be used with Boolean views. `PC_INT_DOM`, `PC_INT_BND`, and `PC_INT_VAL` are mapped to the single Boolean propagation condition `PC_BOOL_VAL`. Having the same interface for integer and Boolean views is essential: Section 27.3 shows how integer propagators can be reused as Boolean propagators without requiring any modification.

OR TRUE CONCISE ≡ [DOWNLOAD]

```
...
class OrTrue :
  public NaryPropagator<Int::BoolView,Int::PC_BOOL_VAL> {
public:
  OrTrue(Home home, ViewArray<Int::BoolView>& x)
    : NaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>(home,x) {}
  ...
  OrTrue(Space& home, bool share, OrTrue& p)
    : NaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>(home,share,p) {}
  ...
};
...
```

Figure 23.7: Naive Boolean disjunction using a propagator pattern

23.3 Dynamic subscriptions

The previous section has been nothing but a warm-up to the presentation of dynamic subscriptions as another technique to avoid propagator execution.

Watched literals. The naive propagator presented above is disastrous: every time the propagator is executed, it checks all of its views to determine whether a view has been assigned to `0` or `1`. Worse yet, pretty much all of the propagator executions are entirely pointless for propagation (but not for determining subsumption as is discussed below).

One idea would be to only check until two unassigned views have been encountered. In this case, it is clear that the propagator cannot perform any propagation. Of course, this might prevent the propagator from detecting subsumption as early as possible: as it does not scan all views but stops scanning after two unassigned views, it might miss out on a view already assigned to `1`.

The idea to stop scanning after two unassigned views have been encountered can be taken even further. A well-known technique for the efficient implementation of Boolean SAT (satisfiability) solvers are *watched literals* [35]: it is sufficient to subscribe to two Boolean views for propagating a Boolean disjunction to be satisfied. Subscribing to a single Boolean view is not enough: if all views but the subscription view are assigned to `0` the subscription view must be assigned to `1` to perform propagation, however the propagator will not be scheduled as the single subscription view is not assigned. More than two subscriptions are not needed for propagation, as the propagator can only propagate if a single unassigned view remains. It might be the case that a view to which the propagator has not subscribed is assigned to `1`. That means that the propagator is not subsumed as early as possible but that does not affect propagation.

OR TRUE WITH DYNAMIC SUBSCRIPTIONS ≡ [DOWNLOAD]

```
...
class OrTrue :
  public BinaryPropagator<Int::BoolView,Int::PC_BOOL_VAL> {
protected:
  ViewArray<Int::BoolView> x;
public:
  OrTrue(Home home, ViewArray<Int::BoolView>& y)
    : BinaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>
      (home,y[0],y[1]), x(y) {
    x.drop_fst(2);
  }
  ...
  virtual size_t dispose(Space& home) {
    (void) BinaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>
      ::dispose(home);
    return sizeof(*this);
  }
  ...
  ▶ COPY
  ▶ RESUBSCRIBE
  ▶ PROPAGATION
};
...
```

Figure 23.8: Boolean disjunction with dynamic subscriptions

The propagator. The propagator using dynamic subscriptions maintains exactly two subscriptions to its Boolean views. Figure 23.8 shows the `OrTrue` propagator with dynamic subscriptions. It inherits from the `BinaryPropagator` pattern and the views `x0` and `x1` of the pattern are exactly the two views with subscriptions. All remaining views without subscriptions are stored in the view array `x`. The operation `drop_fst(n)` for an integer `n` drops the first `n` elements of a view array and shortens the array by `n` elements accordingly (that is, `drop_fst()` is dual to `drop_lst()` as used in the previous section). Note that the propagator must define a `dispose()` member function: this is needed not because `dispose()` must cancel additional subscriptions (the very point is that `x` has no subscriptions) but that it must return the correct size of `OrTrue`.

Tip 23.4 (`drop_fst()` and `drop_lst()` are efficient)**.** Due to the very special memory allocation policy used for view arrays (their memory is allocated in a space and is never freed as they only can shrink, see Chapter 30), both `drop_fst()` and `drop_lst()` have constant time complexity. ◀

Propagation. The idea of how to perform propagation is fairly simple: if one of the views with subscriptions is assigned to `1`, the propagator is subsumed. If one of the subscription views is assigned to `0`, say `x0`, a function `resubscribe()` tries to find a yet unassigned view to subscribe to it and store it as `x0`. If there is no such view but there is a view assigned to `1`, the propagator is subsumed. If there is no such view, then the propagator tries to assign `1` to `x1`, and, if successful, the propagator is also subsumed. The implementation is as follows:

PROPAGATION ≡
```
virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
  if (x0.one() || x1.one())
    return home.ES_SUBSUMED(*this);
  if (x0.zero())
    GECODE_ES_CHECK(resubscribe(home,x0,x1));
  if (x1.zero())
    GECODE_ES_CHECK(resubscribe(home,x1,x0));
  return ES_FIX;
}
```

Resubscribing. The function `resubscribe()` implements the search for a yet unassigned view for subscription and is shown in Figure 23.9.

Copying. The assigned views in `x` do not really matter much: all views assigned to `0` can be discarded. If there is a view assigned to `1` all other views can be discarded (of course, a single view assigned to `1` must be kept for correctness). Hence a good idea for copying is: copy only those views that still matter. This leads to a smaller view array requiring less memory. We decide to discard assigned views as much as we can in the `copy()` function rather than in the constructor used for copying. By this, also the original and not only the copy profits

```
RESUBSCRIBE ≡
  ExecStatus resubscribe(Space& home,
                         Int::BoolView& y, Int::BoolView z) {
    for (int i=x.size(); i--; )
      if (x[i].one()) {
        return home.ES_SUBSUMED(*this);
      } else if (x[i].zero()) {
        x.move_lst(i);
      } else {
        y=x[i]; x.move_lst(i);
        y.subscribe(home,*this,Int::PC_BOOL_VAL);
        return ES_FIX;
      }
    GECODE_ME_CHECK(z.one(home));
    return home.ES_SUBSUMED(*this);
  }
```

Figure 23.9: Resubscribing for Boolean disjunction with dynamic subscriptions

from fewer views. While the copy benefits because there are less views to be stored, the original propagator benefits because `resubscribe()` does not have to scan assigned views as they already have been eliminated. Following this discussion, the `copy()` function can be implemented as:

```
COPY ≡
  virtual Propagator* copy(Space& home, bool share) {
    for (int i=x.size(); i--; )
      if (x[i].one()) {
        x[0]=x[i]; x.size(1); break;
      } else if (x[i].zero()) {
        x.move_lst(i);
      }
    return new (home) OrTrue(home,share,*this);
  }
```

There is another optimization during copying that looks promising. If all views in `x` are eliminated, a propagator without the view array `x` is sufficient as a copy. If there is a view assigned to 1, then a propagator should be created that is subsumed. How this can be achieved is discussed in Section 24.5.

24 Reification and rewriting

This chapter discusses how to implement propagators for reified constraints and how to optimize constraint propagation by propagator rewriting. Gecode does not offer any special support for reification but uses propagator rewriting as a more general technique.

Overview. Reified constraints and how they can be propagated is reviewed in Section 24.1. The following section (Section 24.2) presents a reified less or equal propagator as an example. How to implement both half and full reification is discussed in the following section. General propagator rewriting during constraint propagation is discussed in Section 24.4, whereas Section 24.5 presents how to rewrite propagators during cloning.

24.1 Reification

Before reading this section, please read about reification in Section 4.3.3 and about half reification in Section 4.3.4.

A propagator for the reified constraint $\mathtt{b} = \mathtt{1} \Leftrightarrow c$ for a Boolean *control variable* `b` propagates as follows:

1. If `b` is `1`, propagate c (reification modes: `RM_EQV` $\Leftrightarrow$, `RM_IMP` $\Rightarrow$).

2. If `b` is `0`, propagate $\neg c$ (reification modes: `RM_EQV` $\Leftrightarrow$, `RM_PMI` $\Leftarrow$). Not that it is not always easy to propagate the negation of a constraint c effectively and efficiently.

3. If c is subsumed, propagate that `b` is `1` (reification modes: `RM_EQV` $\Leftrightarrow$, `RM_PMI` $\Leftarrow$).

4. If $\neg c$ is subsumed, propagate that `b` is `0` (reification modes: `RM_EQV` $\Leftrightarrow$, `RM_IMP` $\Rightarrow$).

The idea how to implement reification in Gecode is quite simple: if the first (or second) case is true, the reified propagator implementing the reified constraint rewrites itself into a propagator for c (or $\neg c$). The remaining two cases are nothing but testing criteria for subsumption.

The advantage of rewriting a reified propagator for $\mathtt{b} = \mathtt{1} \Leftrightarrow c$ into a non-reified propagator for c or $\neg c$ is that the propagators created by rewriting are typically available anyway, and that after rewriting no more overhead for reification is incurred.

24.2 A fully reified less or equal propagator

We will discuss a fully reified less or equal propagator $\texttt{b} = 1 \Leftrightarrow \texttt{x} \leq \texttt{y}$ for integer variables `x` and `y` using full reification (that is, reification mode `RM_EQV` $\Leftrightarrow$) as an example. Taking the above propagation rules for a reified constraint, we obtain:

1. If `b` is `1`, propagate $\texttt{x} \leq \texttt{y}$.
2. If `b` is `0`, propagate $\texttt{x} > \texttt{y}$ (actually, we choose to propagate $\texttt{y} < \texttt{x}$ instead).
3. If $\texttt{x} \leq \texttt{y}$ is subsumed, propagate that `b` is `1`.
4. If $\texttt{x} > \texttt{y}$ is subsumed, propagate that `b` is `0`.
5. If none of the above rules apply, the propagator is at fixpoint.

The propagator `ReLeEq` for reified less or equal relies on propagators `Less` for less and `LeEq` for less or equal (the propagators are not shown, see Section 22.7 instead). The `propagate()` function as shown in Figure 24.1 follows the propagation rules sketched above.

Propagator rewriting. The `GECODE_REWRITE` macro takes the propagator (here, `*this`) to be rewritten and an expression that posts the new propagator as arguments. It relies on the fact that the identifier `home` refers to the current home space (like the fail macros in Section 22.5). The macro expands to something along the following lines:

```
size_t s = this->dispose(home);
GECODE_ES_CHECK(LeEq::post(home(*this),x0,x1));
return home.ES_SUBSUMED_DISPOSED(*this,s);
```

That is, the `ReLeEq` propagator is disposed, the new `LeEq` propagator is posted, and subsumption is reported. The order is essential for performance: by first disposing `ReLeEq`, the data structures that store the subscriptions for `x0` and `x1` will have at least one free slot for the subscriptions that are created by posting `LeEq` and hence avoid (rather inefficient and memory consuming) resizing. Note that it is important to understand that the old propagator is disposed before the new propagator is posted. In case that some data structures that are deleted during disposal are needed for posting, one either has to make sure that the data structures outlive the call to `dispose()` or that one does not use the `GECODE_REWRITE` macro but first posts the new propagator and only then reports subsumption.

Another essential part is that after calling `ES_SUBSUMED()`, a propagator is not allowed to do anything that can schedule propagators (such as performing modification operations or creating subscriptions). That is the reason that first `dispose()` is called and later the special variant `ES_SUBSUMED_DISPOSED()` is used for actually reporting subsumption. Do not use `ES_SUBSUMED_DISPOSED()` unless you really, really have to!

LESS OR EQUAL REIFIED FULL ≡ [DOWNLOAD]

```
...
class Less
...
};
class LeEq
...
};

class ReLeEq
  : public Int::ReBinaryPropagator<Int::IntView,Int::PC_INT_BND,
                                   Int::BoolView> {
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
    if (b.one())
      GECODE_REWRITE(*this,LeEq::post(home(*this),x0,x1));
    if (b.zero())
      GECODE_REWRITE(*this,Less::post(home(*this),x1,x0));
    switch (Int::rtest_lq(x0,x1)) {
    case Int::RT_TRUE:
      GECODE_ME_CHECK(b.one(home)); break;
    case Int::RT_FALSE:
      GECODE_ME_CHECK(b.zero(home)); break;
    case Int::RT_MAYBE:
       return ES_FIX;
    }
    return home.ES_SUBSUMED(*this);
  }
};
...
```

Figure 24.1: A constraint and propagator for fully reified less or equal

Adding information to `Home`. As has been discussed in Tip 2.1 and Section 22.5, posting uses a value of class `Home` instead of a reference to a `Space`. In the expansion of `GECODE_REWRITE` as shown above, the call operator `()` as in

```
home(*this)
```

returns a new value of type `Home` with the additional information added that the propagator to be posted is in fact a rewrite of the propagator `*this`. This information is important, for example, for inheriting the AFC (accumulated failure count, see Section 8.5.2): the newly created propagator will inherit the number of accumulated failures from the propagator being rewritten. There will be other applications of `Home` in future versions of Gecode.

Testing relations between views. Rather than testing whether `x0` $\leq$ `x1` or `x0` $>$ `x1` hold individually, we use the function `Int::rtest_lq()` that tests whether two views are less or equal and returns whether the relation holds (`Int::RT_TRUE`), does not hold (`Int::RT_FALSE`), or might hold or not (`Int::RT_MAYBE`).

Relation tests are available for two views (integer or Boolean) or for a view (again, integer or Boolean) and an integer. Relation tests exist for all inequality relations: `Int::rtest_le()` (for $<$), `Int::rtest_lq()` (for $\leq$), `Int::rtest_gr()` (for $>$), and `Int::rtest_gq()` (for $\geq$). For equality, a bounds test (`Int::rtest_eq_bnd()`) as well as a domain test exists (`Int::rtest_eq_dom()`).

While `Int::rtest_eq_bnd()` only uses the bounds for testing the relation (with constant time complexity), `Int::rtest_eq_dom()` uses the full set of values in the domain of the views to determine the relation (having linear time complexity in the length of the range representation of the views' domains, see Section 25.2 for more information on range representations of view domains). The same holds true for disequality: `Int::rtest_nq_bnd()` performs the bounds-only test, whereas `Int::rtest_nq_dom()` performs the full domain test. For example, `Int::rtest_eq_bnd(x,y)` for `x` $\in \{0,2\}$ and `y` $\in \{1,3\}$ returns `Int::RT_MAYBE`, whereas `Int::rtest_eq_dom()` returns `Int::RT_FALSE`.

Reified propagator patterns. The integer module (as these patterns require Boolean views they are part of the integer module) provides reified propagator patterns for unary propagators (`Int::ReUnaryPropagator`) and binary propagators (`Int::ReBinaryPropagator` and `Int::ReMixBinaryPropagator`). In addition to views `x0` (and `x1` for the binary variants), they define a Boolean control variable `b`. Please note that in Figure 24.1 a reified propagator pattern requires an additional template argument for the Boolean control view used (the mystery why this is useful is lifted in Chapter 27).

24.3 Supporting both full and half reification

There are two different options for implementing all different reification modes: either implement a single propagator that stores its reification mode, or implement three different

LESS OR EQUAL REIFIED HALF ≡ [DOWNLOAD]

```
...
template<ReifyMode rm>
class ReLeEq
  : public Int::ReBinaryPropagator<Int::IntView,Int::PC_INT_BND,
                                   Int::BoolView> {
  ...
  ▶ PROPAGATE FUNCTION
};

void leeq(Home home, IntVar x0, IntVar x1, Reify r) {
  if (home.failed()) return;
  switch (r.mode()) {
  case RM_EQV:
    GECODE_ES_FAIL(ReLeEq<RM_EQV>::post(home,x0,x1,r.var()));
    break;
  case RM_IMP:
    GECODE_ES_FAIL(ReLeEq<RM_IMP>::post(home,x0,x1,r.var()));
    break;
  case RM_PMI:
    GECODE_ES_FAIL(ReLeEq<RM_PMI>::post(home,x0,x1,r.var()));
    break;
  default:
    throw Int::UnknownReifyMode("leeq");
  }
}
```

Figure 24.2: A constraint and propagator for full and half reified less or equal

propagators, one for each reification mode. Due to performance reasons we choose the latter variant here. That is, one needs one propagator for each reification mode: `RM_EQV` for equivalence ($\Leftrightarrow$), `RM_IMP` for implication ($\Rightarrow$), and `RM_PMI` for reverse implication ($\Leftarrow$). Instead of three different implementations it is better to have a single implementation that is parametric with respect to the reification mode.

Figure 24.2 shows the constraint post function `leeq()` for the reified less or equal propagator supporting all three reification modes. The class `ReLeEq` now is parametric with respect to the reification mode the propagator implements. The constraint post function now posts the propagator that corresponds to the reification mode (available via `r.mode()`) passed as argument `r` of type `Reify`. The Boolean control variable can be returned by `r.var()`.

The idea for the reified propagator is straightforward, its `propagate()` function is shown in Figure 24.3: the four different parts of the propagator are employed depending on the reification mode `rm`. Note that this is entirely schematic and any reified propagator can be

PROPAGATE FUNCTION ≡

```
  virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
    if (b.one()) {
      if (rm != RM_PMI)
        GECODE_REWRITE(*this,LeEq::post(home(*this),x0,x1));
    } else if (b.zero()) {
      if (rm != RM_IMP)
        GECODE_REWRITE(*this,Less::post(home(*this),x1,x0));
    } else {
      switch (Int::rtest_lq(x0,x1)) {
      case Int::RT_TRUE:
        if (rm != RM_IMP)
          GECODE_ME_CHECK(b.one(home));
        break;
      case Int::RT_FALSE:
        if (rm != RM_PMI)
          GECODE_ME_CHECK(b.zero(home));
        break;
      case Int::RT_MAYBE:
        return ES_FIX;
      }
    }
    return home.ES_SUBSUMED(*this);
  }
```

Figure 24.3: Propagate function for full and half reified less or equal

programmed supporting all reification modes in that way.

24.4 Rewriting during propagation

Rewriting during propagation is a useful technique that is not limited to implementing reification. We consider a `Max` propagator that propagates $\max(x_0, x_1) = x_2$. The propagation rules (we are giving bounds propagation rules that achieve bounds($\mathbb{Z}$) consistency, see [12]) for `Max` are easy to understand when looking at an equivalent formulation of max [47]:

$$\begin{aligned} \max(x_0, x_1) = x_2 &\iff (x_0 \leq x_2) \wedge (x_1 \leq x_2) \wedge \big((x_0 = x_2) \vee (x_1 = x_2)\big) \\ &\iff (x_0 \leq x_2) \wedge (x_1 \leq x_2) \wedge \big((x_0 \geq x_2) \vee (x_1 \geq x_2)\big) \end{aligned}$$

Then, the propagation rules can be turned into the following C++-code to be executed by the `propagate()` member function of `Max`:

```
GECODE_ME_CHECK(x2.lq(home,std::max(x0.max(),x1.max())));
GECODE_ME_CHECK(x2.gq(home,std::max(x0.min(),x1.min())));
GECODE_ME_CHECK(x0.lq(home,x2.max()));
GECODE_ME_CHECK(x1.lq(home,x2.max()));
if ((x1.max() <= x0.min()) ||
    (x1.max() < x2.min()))
  GECODE_ME_CHECK(x0.gq(home,x2.min()));
if ((x0.max() <= x1.min()) ||
    (x0.max() < x2.min()))
  GECODE_ME_CHECK(x1.gq(home,x2.min()));
```

Please take a second look: the condition of the first **`if`**-statement tests whether x_1 is less or equal than x_0, or whether x_1 is less than x_2. In both cases, $\max(x_0, x_1) = x_2$ simplifies to $x_0 = x_2$. Exactly this simplification can be implemented by propagator rewriting, please check Figure 24.4. Note that the propagator is naive in that it does not implement the propagator to be idempotent (however, this can be done exactly as demonstrated in Section 23.1).

Another idea would be to perform rewriting during cloning: the `copy()` function could return an `Equal` propagator rather than a `Max` propagator in the cases where rewriting is possible. However, this is illegal: it would create a propagator with only two views, hence one view would not be updated even though there is a subscription for it (violating obligation "update complete" from Section 22.8). Also, canceling a subscription is illegal during cloning (violating obligation "cloning conservative" from Section 22.8). The next section shows an example with opposite properties: rewriting during propagation makes no sense (even though it would be legal) whereas rewriting during cloning is legal and useful.

MAX USING REWRITING ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_BND> {
  ...
};
class Max : public TernaryPropagator<Int::IntView,Int::PC_INT_BND> {
public:
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    GECODE_ME_CHECK(x2.lq(home,std::max(x0.max(),x1.max())));
    GECODE_ME_CHECK(x2.gq(home,std::max(x0.min(),x1.min())));
    GECODE_ME_CHECK(x0.lq(home,x2.max()));
    GECODE_ME_CHECK(x1.lq(home,x2.max()));
    if ((x1.max() <= x0.min()) || (x1.max() < x2.min()))
      GECODE_REWRITE(*this,Equal::post(home(*this),x0,x2));
    if ((x0.max() <= x1.min()) || (x0.max() < x2.min()))
      GECODE_REWRITE(*this,Equal::post(home(*this),x1,x2));
    if (x0.assigned() && x1.assigned() && x2.assigned())
      return home.ES_SUBSUMED(*this);
    else
      return ES_NOFIX;
  }
};
...
```

Figure 24.4: A maximum propagator using rewriting

24.5 Rewriting during cloning

In Section 23.3, the propagator `OrTrue` is simplified during copying by eliminating assigned views. Here we show that the propagator created as a copy during cloning can be optimized further by rewriting it during copying.

Copying. The modified `copy()` function is as follows:

```
COPY ≡
  virtual Propagator* copy(Space& home, bool share) {
    for (int i=x.size(); i--; )
      if (x[i].one()) {
        x[0]=x[i]; x.size(1);
        return new (home) SubsumedOrTrue(home,share,*this);
      } else if (x[i].zero()) {
        x.move_lst(i);
      }
    if (x.size() == 0)
      return new (home) BinaryOrTrue(home,share,*this);
    else
      return new (home) OrTrue(home,share,*this);
  }
```

The `copy()` function takes advantage of two cases:

1. If there is a view that is assigned to `1`, the propagator is subsumed. However, during cloning a propagator cannot be deleted by subsumption. Reporting subsumption is only possible during propagation. The `copy()` function does the next best thing: it creates a propagator `SubsumedOrTrue` that next time it is executed will in fact report subsumption.

   Note that in this situation rewriting during cloning is preferable to rewriting during propagation. An important aspect of the `OrTrue` propagator is that it does not inspect all of its views during finding a view for resubscribing (as implemented by the `resubscribe()` member function). Instead, inspection stops as soon as the first unassigned view is found. That entails that a view that is assigned to `1` might be missed during propagation. In other words, rewriting during cloning also optimizes subsumption detection.

2. If all views in the view array `x` are assigned to `0`, the view array is actually not longer needed (it has no elements). In this case, the `copy()` function creates a propagator `BinaryOrTrue` that is a special variant of `OrTrue` limited to just two views. Note that the two views `x0` and `x1` are known to be not yet assigned: otherwise, the propagator would not be at fixpoint. This is impossible as only spaces that are at fixpoint

OR TRUE USING REWRITING ≡ [DOWNLOAD]

```
  ...
  class BinaryOrTrue :
    ...
    BinaryOrTrue(Space& home, bool share,
                 BinaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>& p)
    ...
    virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
      if (x0.zero())
        GECODE_ME_CHECK(x1.one(home));
      if (x1.zero())
        GECODE_ME_CHECK(x0.one(home));
      return home.ES_SUBSUMED(*this);
    }
  };
  class SubsumedOrTrue :
    ...
    SubsumedOrTrue(Space& home, bool share,
                   BinaryPropagator<Int::BoolView,Int::PC_BOOL_VAL>& p)
    ...
    virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
      return home.ES_SUBSUMED(*this);
    }
  };
  class OrTrue :
    ...
    ▶ COPY
    ...
  };
  ...
```

Figure 24.5: A Boolean disjunction propagator using rewriting

(and hence all of its propagators must be at fixpoint) can be cloned (this invariant is discussed in Section 39.1).

Rewriting during propagation would be entirely pointless in this situation: the propagator (be it `OrTrue` or `BinaryOrTrue`) will be executed at most one more time and will become subsumed then (or, due to failure, it is not executed at all). As rewriting incurs the cost of creating and disposing propagators and subscriptions, rewriting in this case would actually slow down execution.

Constructors for copying. The relevant parts of the two special propagators for rewriting `SubsumedOrTrue` and `BinaryOrTrue` are shown in Figure 24.5. Their `propagate()` functions are exactly as sketched above. The only other non-obvious aspect are the constructors used for copying during cloning: they are now called for a propagator of class `OrTrue`. In this example, it is sufficient to have a single constructor for copying as all the propagators `OrTrue`, `SubsumedOrTrue`, and `BinaryOrTrue` inherit from `BinaryPropagator`. In other cases, it might be necessary to have more than a single constructor for copying defined by a propagator class `C`: one for copying a propagator of class `C` and one for creating propagators as rewrites of other propagators.

25 Domain propagation

The propagators in previous chapters use simple modification operations on variable views in that only a single integer defines how a view's domain is changed. When programming propagators that perform more elaborate domain reasoning, these modification operations are typically insufficient as they easily lead to incorrect and inefficient propagators.

To conveniently program efficient propagators that perform domain reasoning, Gecode offers modification operations that take entire sets of integers as arguments. To be efficient, these sets are not implemented as a typical set data structure but as *iterators*: the integers (or entire ranges of integers) can be iterated in increasing order (smallest first).

Overview. This chapters motivates why special domain operations on variable views are needed (Section 25.1) and demonstrates how iterators are used for domain propagation (Section 25.2 and Section 25.3). Section 25.4 and Section 25.5 describe modification event deltas and staging for efficiently combining bounds with domain propagation.

25.1 Why domain operations are needed

Let us consider an attempt to implement domain propagation for an equality constraint on views `x0` and `x1`. The idea for pruning is simple: only keep those values of `x0` and `x1` that are common to the domains of both `x0` and `x1`. This leads to a first version of a domain equality propagator as shown in Figure 25.1. Beware, the "propagator" is not only naive but also incorrect: it will crash Gecode!

The propagator uses iterators of type `Int::ViewValues` to iterate over the values of an integer view in increasing order (the template argument defines the type of the view to iterate on). The increment operator `++` moves the iterator to the next value, the application operator `()` tests whether the iterator has more values to iterate, and the function `val()` returns the iterator's current value. The reason why the propagator will crash is simple: it uses the iterator `i` for iterating over the values of `x0` and modifies the domain of `x0` while iterating! This is illegal: when using an iterator for a view, the view cannot be modified (or, in C++ lingua: modifying the variable invalidates the iterator).

One idea how to fix this problem is to use a temporary data structure in which the newly computed intersection domain is stored. That is not very tempting: allocating and initializing an intermediate data structure is costly.

INCORRECT DOMAIN EQUAL ☰

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
public:
  Equal(Home home, Int::IntView x0, Int::IntView x1)
    : BinaryPropagator<Int::IntView,Int::PC_INT_DOM>(home,x0,x1) {}
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    Int::ViewValues<Int::IntView> i(x0), j(x1);
    while (i() && j())
      if (i.val() < j.val()) {
        GECODE_ME_CHECK(x1.nq(home,i.val())); ++i;
      } else if (j.val() < i.val()) {
        GECODE_ME_CHECK(x0.nq(home,j.val())); ++j;
      } else {
        ++i; ++j;
      }
    while (i()) {
      GECODE_ME_CHECK(x1.nq(home,i.val())); ++i;
    }
    while (j()) {
      GECODE_ME_CHECK(x0.nq(home,j.val())); ++j;
    }
    if (x0.assigned() && x1.assigned())
      return home.ES_SUBSUMED(*this);
    else
      return ES_FIX;
  }
};
...
```

Figure 25.1: An incorrect propagator for domain equal

NAIVE DOMAIN EQUAL ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
  ...
  virtual PropCost cost(const Space&, const ModEventDelta&) const {
    return PropCost::binary(PropCost::HI);
  }
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    Int::ViewRanges<Int::IntView> r0(x0);
    GECODE_ME_CHECK(x1.inter_r(home,r0));
    Int::ViewRanges<Int::IntView> r1(x1);
    GECODE_ME_CHECK(x0.narrow_r(home,r1));
    if (x0.assigned() && x1.assigned())
      return home.ES_SUBSUMED(*this);
    else
      return ES_FIX;
  }
};
...
```

Figure 25.2: A naive propagator for domain equal

But even then, the approach is flawed from the beginning: a single `nq()` operation on a view has linear runtime in the size of the view's domain (actually, in the length of its range sequence, see below). As potentially a linear number of `nq()` operations are executed, the propagator will have quadratic complexity even though it should have linear (as the values of the domains are available in strictly increasing order).

25.2 Iterator-based modification operations

Figure 25.2 shows a working, yet still naive, implementation of an equality propagator performing domain reasoning. Note that the `cost()` function is overridden: even though the propagator is binary, it now incurs a higher cost due to the domain operations to be performed. The `propagate()` function uses two *range iterators* for iterating over the range sequence of the domains of `x0` and `x1`. For the definition of a range sequence, see Section 4.1.6.

A *range iterator* for a range sequence $s = \langle [n_i .. m_i] \rangle_{i=0}^{k}$ is an object that provides iteration over s: each of the $[m_i .. n_i]$ can be obtained in sequential order but only one at a time. A range iterator provides the following operations: the application operator `()` tests whether there are more ranges to iterate, the increment operator `++` moves to the next range, the function `min()` (`max()`) returns the smallest (largest) value of the current range, and the function `width()` returns the *width* of the current range (that is, its number of elements) as

an unsigned integer.

The motivation to iterate over range sequences rather than individual values is efficiency: as there are typically less ranges than indvidual values, iteration over ranges can be more efficient. Moreover, many operations are sufficiently easy to express in terms of ranges rather than values (see Section 25.3 for an example).

Iterator-based modification operations. The propagator uses two modification operations for range iterators: `x1.inter_r()` intersects the current domain of `x1` with the set defined by the range iterator `r0` for `x0`. After this operation (provided no failure occurred), the view `x1` has the correct domain: the intersection of the domains of `x0` and `x1`. The operation `x0.narrow_r()` replaces the current domain of `x0` by the set defined by the range iterator `r1` (which iterates the intersection of the domains of `x0` and `x1`).

A third operation available for range iterators is `minus_r()` which removes the values as described by the range iterator passed as argument.

Instead of using range iterators for modification operations, one can also use value iterators instead. Similarly, a view provides operations `inter_v()`, `narrow_v()`, and `minus_v()` for intersecting, replacing, and removing values.

Tip 25.1 (Narrow is dangerous)**.** The `narrow_r()` operation used in the above example is dangerous (as is the `narrow_v()` operation). As discussed in Section 22.8, a propagator must be contracting: the domain of a view is only allowed to get smaller. If `narrow_r()` is used incorrectly, then one could actually replace the view's domain by a larger domain.

In the above example, the propagator is contracting: `r1` refers to the intersection of `x0` and `x1`, which of course has no more values than `x0`. ◀

Tip 25.2 (Iterators must be increasing)**.** The range sequence iterated by a range iterator *must* be sorted in increasing order and *must* not overlap as described above. Otherwise, domain operations using range iterators become incorrect.

For value iterators, the values must be increasing in value with iteration. But it is okay if the same value is iterated over multiply. That is, the values must be increasing but not necessarily strictly increasing. ◀

Avoiding shared iterators. The problem that made our attempt to implement propagation for equality in Section 25.1 incorrect is to use iterators for iterating over views that are simultaneously modified.

Iterator-based modification operations automatically take care of potential sharing between the iterator they use and the view domain they update. By default, an iterator-based modification operation assumes that iterator and domain are shared. The operation first constructs a new domain and iterates to the end of the iterator. Only then the old domain is replaced by the newly constructed domain. In many cases, however, there is no sharing between iterator and domain and the operations could be performed more efficiently by in-place update operations on the domain.

NON-SHARED DOMAIN EQUAL ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
  ...
  static ExecStatus post(Home home,
                         Int::IntView x0, Int::IntView x1) {
    if (!same(x0,x1))
      (void) new (home) Equal(home,x0,x1);
    return ES_OK;
  }
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    Int::ViewRanges<Int::IntView> r0(x0);
    GECODE_ME_CHECK(x1.inter_r(home,r0,false));
    Int::ViewRanges<Int::IntView> r1(x1);
    GECODE_ME_CHECK(x0.narrow_r(home,r1,false));
    ...
  }
};
...
```

Figure 25.3: A propagator for domain equal without sharing

In our example, there is no sharing if the views `x0` and `x1` do not refer to the very same variable implementation. If they do, the propagator should not even be posted as it is subsumed anyway (views of the same type referring to the same variable implementation are trivially equal). Figure 25.3 shows the propagator post function of an improved propagator for equality: the propagator is posted only if `x0` and `x1` do not refer to the same variable implementation. The `propagate()` function is improved by giving an additional **`false`** argument to both `inter_r()` and `narrow_r()`. Hence, the two operations use more efficient operations performing in-place updates on domains.

25.3 Taking advantage of iterators

This section shows how the combination of simple iterators can help in implementing domain propagation.

Suppose that we want to implement a close variant of the equality constraint, namely $x = y + c$ for integer variables x and y and some integer value c. It is easy to see that the new domain for x must be all values of x intersected with the values $\{n+c \mid n \in y\}$. Likewise, the new domain for y must be all values of y intersected with the values $\{n-c \mid n \in x\}$. But how can we implement these simple propagation rules?

Mapping range sequences. Assume a range sequence $\langle [m_i \,..\, n_i] \rangle_{i=0}^{k}$ for the values in the domain of y. Then, what we want to compute is a range sequence

$$\langle [m_i + c \,..\, n_i + c] \rangle_{i=0}^{k}.$$

With other words, we want to map the first into the second range sequence. Luckily, this is easy. Suppose that `r` is a range iterator for the view `y`. A range iterator for our desired range sequence can be constructed using the `Iter::Range::Map` iterator and an object that describes how to map the ranges of `r` to the desired ranges.

For this, we define the following class:

OFFSET MAP ☰
```
class OffsetMap {
protected:
  int o;
public:
  OffsetMap(int o0) : o(o0) {}
  int min(int m) const {
    return m+o;
  }
  int max(int m) const {
    return m+o;
  }
};
```

DOMAIN EQUAL WITH OFFSET ≡ [DOWNLOAD]

```
...
▶ OFFSET MAP
class EqualOffset :
  public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
protected:
  int c;
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
    Int::ViewRanges<Int::IntView> r0(x0);
    OffsetMap om0(-c);
    Iter::Ranges::Map<Int::ViewRanges<Int::IntView>,OffsetMap>
      mr0(r0,om0);
    GECODE_ME_CHECK(x1.inter_r(home,mr0,false));
    Int::ViewRanges<Int::IntView> r1(x1);
    OffsetMap om1(c);
    Iter::Ranges::Map<Int::ViewRanges<Int::IntView>,OffsetMap>
      mr1(r1,om1);
    GECODE_ME_CHECK(x0.narrow_r(home,mr1,false));
    ...
  }
};
...
```

Figure 25.4: A propagator for domain equal with offset

`OffsetMap` defines to which values the minimum (`min()`) and the maximum (`max()`) of each input range must be mapped. Using `OffsetMap`, we can construct a range iterator `m` for our desired sequence by

```
OffsetMap om(c);
Iter::Ranges::Map<Int::ViewRanges<Int::IntView>,
                  OffsetMap> m(r,om);
```

With the help of the map range iterator, propagation for `OffsetEqual` as shown in Figure 25.4 is straightforward. Note that several functions need to be modified to deal with the additional integer constant `c` used by the propagator (the details can be inspected by downloading the full program code). Further note that the constraint post function is slightly to liberal in that it does not check whether the values with the integer constant `c` added exceed the limits for legal integer values.

While the propagator is reasonably easy to construct using map iterators, Section 27.1.2 shows how to obtain exactly the same propagator without any programming effort but the implementation of an additional constraint post function.

Using and defining iterators. Gecode comes with a multitude of range and value iterators to transform range and value sequences of other iterators. These iterators are defined in the namespace `Iter`. Range iterators are defined in the namespace `Iter::Ranges` and value iterators in the namespace `Iter::Values`. Example iterators include: iterators to convert value into range iterators and vice versa, iterators to compute the union and intersection, iterators to iterate over arrays and bitsets, just to name a few.

But even if the predefined iterators are not sufficient, it is straightforward to implement new iterators: the only requirement is that they implement the appropriate interface mentioned above for range or value iterators. The namespace `Iter` contains a multitude of simple and advanced examples.

Benefits of iterators. The true benefit of using iterators for performing value or range transformations is that the iterator-based domain modification operation with which an iterator is used is automatically specialized at compile time. Typically, no intermediate data structures are created and the modification operations are optimized at compile time for each individual iterator.[1]

A scenario where this in particular matters is when iterators are used as *adaptors* of a propagator-specific data structure. Assume that a propagator uses a specific (typically, quite involved) data structure to perform propagation. Most often this data structure encodes in some form which views should take which values. Then, one can implement simple iterators that inspect the propagator-specific data structure and provide the information about values for views so that they can be used directly with iterator-based modification operations. Again, intermediate data structures are avoided by this approach.

25.4 Modification event deltas

There is a rather obvious approach to improving the efficiency of domain operations performed by propagators: make the domains as small as possible! One typical approach to reduce the size of the domains is to perform bounds propagation first. After bounds propagation, the domains are likely to be smaller and hence the domain operations are likely to be more efficient.

For propagating equality, the simplest idea is to first perform bounds propagation for equality as discussed in Section 23.1, directly followed by domain propagation. However, we can improve further by exploiting an additional token of information about the views of a propagator that is supplied to both the `cost()` and `propagate()` function of a propagator.

The `cost()` and `propagate()` functions of a propagator take an additional *modification event delta* value of type `ModEventDelta` (see Programming actors) as argument. Every propagator maintains a modification event delta that describes how its views have changed since

[1] Some predefined iterators actually have to resort to intermediate data structures. For example, iterators that need to revert the order of its values or ranges (think of a value iterator for values that are the negation of values of some other value iterator).

DOMAIN EQUAL USING BOUNDS PROPAGATION ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
  ...
  virtual ExecStatus propagate(Space& home,
                               const ModEventDelta& med) {
    if (Int::IntView::me(med) != Int::ME_INT_DOM) {
      do {
        GECODE_ME_CHECK(x0.gq(home,x1.min()));
        GECODE_ME_CHECK(x1.gq(home,x0.min()));
      } while (x0.min() != x1.min());
      do {
        GECODE_ME_CHECK(x0.lq(home,x1.max()));
        GECODE_ME_CHECK(x1.lq(home,x0.max()));
      } while (x0.max() != x1.max());
      if (x0.assigned() && x1.assigned())
        return home.ES_SUBSUMED(*this);
      if (x0.range() && x1.range())
        return ES_FIX;
    }
    ...
  }
};
...
```

Figure 25.5: A propagator for domain equal using bounds propagation

the last time the propagator has been executed. For each view type (that is, integer, Boolean, set, ...) a modification event delta stores a modification event that can be extracted from the modification event delta: If `med` is a modification event delta, then `Int::IntView::me(med)` returns the modification event for integer views.

The extracted modification event describes how all views of a certain view type have changed. For example, for integer views, the modification event `Int::ME_INT_VAL` signals that there is at least one integer view that has been assigned since the propagator has been executed last (analogous for `Int::ME_INT_BND` and `Int::ME_INT_DOM`). Even the modification event `Int::ME_INT_NONE` carries some information: none of the propagator's integer views have been modified (which, of course, can only happen if the propagator also uses views of some other type).

Figure 25.5 shows a propagator that combines both bounds and domain propagation for propagating equality. It first extracts the modification event for integer views from the modification event delta `med`. Only if the bounds (that is, modification events `Int::ME_INT_VAL` or `Int::ME_INT_BND`, hence different from `Int::ME_INT_DOM`) have changed for at least one

of the views `x0` and `x1`, the propagator performs bounds propagation.

After performing bounds propagation, the propagator checks whether it is subsumed. Then it does some more fixpoint reasoning: if the domains of `x0` and `x1` are ranges (that is, they do not have holes), the propagator is at fixpoint. Otherwise, domain propagation is done as shown before.

25.5 Staging

Taking the perspective of a single propagator, first performing bounds propagation directly followed by domain propagation seems to be appropriate. However, when taking into account that some other cheap propagators (at least cheaper than performing domain propagation by this propagator) might already be able to take advantage of the results of bounds propagation, it might actually be better to postpone domain propagation and give cheaper propagators the opportunity to execute. This idea is known as *staging* and has been introduced in Gecode [48, Section 7]. Note that staging is not limited to bounds and domain propagation but captures any stages in propagation that differ in cost.

Here, we focus on staging for first performing bounds propagation (stage "bounds") and then domain propagation (stage "domain") for equality. Additionally, a propagator can be idle (stage "idle"). The stage of a propagator is controlled by how its modification event delta changes:

- Initially, the propagator is idle, its modification event delta is empty, and the propagator is in stage "idle".
- When the modification event delta for integer views changes to `Int::ME_INT_DOM` and the propagator is in stage "idle", the propagator is put into stage "domain" with high propagation cost.
- When the modification event delta for integer views changes to `Int::ME_INT_VAL` or `Int::ME_INT_BND`, the propagator is put into stage "bounds" with low propagation cost.

  Note that this in particular includes the case where the modification event delta for integer views has been `Int::ME_INT_DOM` and where the propagator had already been in stage "domain". As soon as the modification event delta changes to `Int::ME_INT_BND` or `Int::ME_INT_VAL` the propagator is put into stage "bounds".

By the very construction of modification event deltas, the modification event delta for integer views can neither change from `Int::ME_INT_VAL` to `Int::ME_INT_BND` nor from `Int::ME_INT_BND` (or `Int::ME_INT_VAL`) to `Int::ME_INT_DOM`. That is, if the equality propagator using staging is in stage "bounds" it stays in that stage until it is executed.

When the propagator is executed, it can be either in stage "bounds" or stage "domain". If it is executed in stage "bounds", it performs bounds propagation and then returns that it wants to be put into stage "domain". The essential point is that the propagator does not

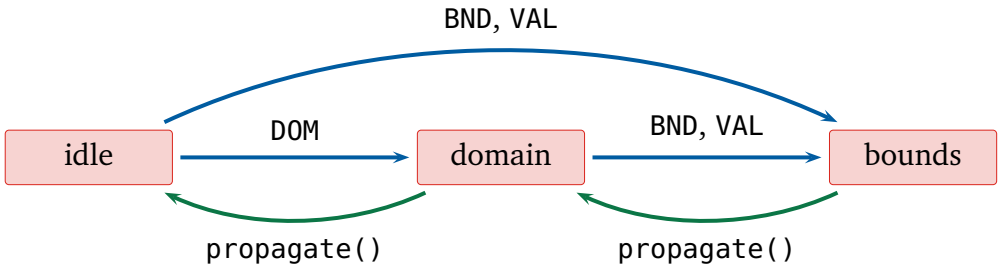


Figure 25.6: Stage transitions for the equality propagator

continue with domain propagation but gives other propagators the opportunity to execute first. If the propagator is executed in stage "domain", it performs domain propagation and returns that it is at fixpoint (and hence the propagator is put into stage "idle"). Figure 25.6 summarizes the stage transitions, where a blue transition is triggered by a change in the modification event delta (`Int::ME_INT_DOM` is abbreviated by `DOM` and so on) and a green transition is performed be executing the propagator.

Rescheduling propagators. The cost of a propagator depends on its modification event delta. This connection goes even further: only if the modification event delta of a propagator changes, a propagator is rescheduled according to its cost by recomputing the `cost()` function.

Not recomputing cost each time a propagator might be scheduled is done for two reasons. First, the number of possibly expensive cost computations is reduced. Second, always rescheduling would also violate the fairness among all propagators already scheduled with same cost. If a propagator is scheduled often, it would be penalized as its execution would be deferred.

Section 26.4 discusses a technique to force rescheduling of a propagator, irrespective of its modification event delta.

Controlling staging by modification event deltas. The `cost()` function and the essential parts of the `propagate()` function of a propagator that uses staging to combine bounds and domain propagation for equality are shown in Figure 25.7.

The `cost()` function returns the cost based on the modification event delta `med`: if `med` only includes `Int::ME_INT_DOM`, then the propagator returns that next time it executes, it executes at high-binary cost (according to stage "domain"). Otherwise, the propagator returns that next time it executes, it executes at low-binary cost (according to stage "bounds").

The `propagate()` function is almost identical to the function shown in the previous section. The only difference is that `propagate()` returns after having performed bounds propagation. The call to `ES_FIX_PARTIAL()` specifies that the current propagator `*this` has computed a partial fixpoint for all modification events but `Int::ME_INT_DOM`. The function `Int::IntView::med` creates a modification event delta from a modification event for integer views. As an effect, the modification event delta of the current propagator is set to

DOMAIN EQUAL USING STAGING ≡ [DOWNLOAD]

```
...
class Equal : public BinaryPropagator<Int::IntView,Int::PC_INT_DOM> {
  ...
  virtual PropCost cost(const Space&,
                        const ModEventDelta& med) const {
    if (Int::IntView::me(med) != Int::ME_INT_DOM)
      return PropCost::binary(PropCost::LO);
    else
      return PropCost::binary(PropCost::HI);
  }
  virtual ExecStatus propagate(Space& home,
                               const ModEventDelta& med) {
    if (Int::IntView::me(med) != Int::ME_INT_DOM) {
      ...
      return home.ES_FIX_PARTIAL
        (*this,Int::IntView::med(Int::ME_INT_DOM));
    }
    ...
  }
};
...
```

Figure 25.7: A propagator for domain equal using staging

include nothing but `Int::ME_INT_DOM` and the propagator is scheduled: as defined by the `cost()` function, the propagator is scheduled for high binary cost. That means that other propagators of lower cost might be executed first.

Constructing modification event deltas. Every view type provides a static `med()` function that translates a modification event of that view type into a modification event delta. Modification event deltas for different view types can be combined with the `|` operator. For example, the following expression combines the modification event `ME_INT_BND` for integer views with the modification event `ME_BOOL_VAL` for Boolean views:

```
Int::IntView::med(ME_INT_BND) | Int::BoolView::med(ME_BOOL_VAL)
```

Note that only modification event deltas for different view types can be combined using `|`.

26 Advisors

This chapter is concerned with advisors for efficient incremental propagation. Advisors can be used to provide information to a propagator which of its views have changed and how they have changed.

Overview. In Section 26.1, a motivation and a model for advisors is presented. The following two sections demonstrate advisors. Section 26.2 shows an example propagator that exploits the information provided by an advisor about which view has changed. Section 26.3 shows an example propagator that exploits information on how the domain of its views have changed. Section 26.4 sketches how advisors can be used for forcing propagators to be rescheduled.

26.1 Advisors for incremental propagation

Consider the following, rather simple, example constraint. The constraint `samedom` for an array of integer variables `x` and an integer set `d` holds, if and only if:

- either all variables in `x` take values from `d` (that is, $\mathtt{x}_i \in \mathtt{d}$ for $0 \leq i < |\mathtt{x}|$),
- or none of the `x` take values in `d` (that is, $\mathtt{x}_i \notin \mathtt{d}$ for $0 \leq i < |\mathtt{x}|$).

More knowledge is needed. Obviously, there are two different approaches to realize `samedom`:

decomposition We decompose the `samedom` constraint as follows. We create a Boolean variable `b` and post reified `dom` constraints (see Domain constraints) such that $\mathtt{b} = 1 \Leftrightarrow \mathtt{x}_i \in \mathtt{d}$ (for $0 \leq i < |\mathtt{x}|$). As the single Boolean variable `b` is the same for all reified `dom` constraints, `samedom` is automatically enforced.

implementation A different approach is to implement a dedicated propagator for `samedom`. Propagation is quite simple: whenever the propagator is executed, try to find a view among the $\mathtt{x}_i$ such that either $\mathtt{x}_i \in \mathtt{d}$ or $\mathtt{x}_i \notin \mathtt{d}$. If there is no such $\mathtt{x}_i$, the propagator is at fixpoint. Otherwise, the propagator constrains all $\mathtt{x}_i$ accordingly.

Let us compare the individual merits of the two approaches (note that both achieve domain consistency). Decomposition requires $O(|x|)$ propagators consuming quite some memory. Implementation requires a single propagator only and hence has a lower overhead both for runtime (only a single propagator needs to be scheduled) and memory consumption.

However, the implementation approach is dreadful and is in fact less efficient than decomposition! The problem is the "try to find a view" approach: each time the propagator is executed, it only knows that some of its views have changed since last time the propagator has been executed. It just does not know which views changed! That is, each time the propagator is executed, it needs to scan all its views. For `samedom`, the propagator takes $O(|x| \cdot |d|)$ runtime as scanning needs to inspect the entire domain of each view. This is considerably worse than decomposition: when the domain of a x_i changes, only a single reified propagator for `dom` is executed with cost $O(|d|)$.

The problem is the little amount of information available to a propagator when it is executed. The propagator only knows that some of its views changed but not which views. Hence, just finding out which views have changed always incurs linear cost in the number of views.

For a simple constraint such as `samedom`, the linear overhead is prohibitive. For more involved constraints, decomposition might be infeasible as it would compromise propagation (think of domain consistent `distinct` as an example).

Advisors. Gecode provides *advisors* to inform propagators about view changes. An advisor belongs to a propagator and can be defined (by inheriting from `Advisor`) to store information as needed by its propagator. The sole purpose of an advisor is to subscribe to a view of its propagator: each time the view changes, an `advise()` function of the advisor's propagator is executed with the advisor as argument (sometimes we will be a little sloppy by saying that the advisor is executed itself).

In more detail:

- An advisor must inherit from the class `Advisor`.

- When an advisor is created, it is created with respect to its propagator and a *council* of advisors. Each advisor belongs to a council and a propagator can have at most one council. The sole purpose of a council is to manage its advisors for cloning and disposal. In particular, when the propagator is disposed, the council must be disposed as well.

  A council also provides access to all of its advisors (we will exploit this in the following sections).

- An advisor can subscribe to views (and, hence, an advisor subscription like any other subscription must eventually be canceled). Unlike subscriptions of propagators to views, subscriptions of advisors do not use propagation conditions: an advisor is always executed when its subscription view changes.

Also, an advisor is never executed when the subscription is created, only when the subscription view changes.

- An advisor is executed as follows: the `advise()` function of its propagator is executed. The function takes the advisor as argument and an additional argument of type `Delta`. The *delta* describes how the domain of the view has changed. Clearly, which kind of information a delta provides depends on the type of the view. Deltas, in particular, provide access to the modification event of the operation that triggered the advisor's execution.

  For integer and Boolean views, deltas provide some approximate information about which values have been removed. For an example, see Section 26.3.

- The `advise()` function is *not* allowed to perform propagation by executing modification operations on views. It can change the state of the propagator and must return an execution status: `ES_FAILED` means that the propagator is failed; `ES_FIX` means that the propagator is at fixpoint (hence, it is not scheduled); `ES_NOFIX` means that the propagator is not any longer at fixpoint (hence, it is scheduled). That is, an advisor does exactly what its name suggests as it provides advice to its propagator: when should the propagator be executed and the advisor can also provide information for the next propagator execution.

  The `advise()` function can also return after calling the functions `ES_FIX_DISPOSE()` or `ES_NOFIX_DISPOSE()` which are analogous to `ES_FIX` and `ES_NOFIX` but also dispose the advisor.

  There are two more functions that force the advisor's propagator to be rescheduled. They are discussed in Section 26.4.

Note that a propagator using advisors must make sure that its advisors schedule the propagator when it is not at fixpoint. Otherwise, it would not meet its "subscription complete" obligation (make sure to read Tip 26.2). A propagator is free to mix subscriptions using advisors and subscriptions using propagation conditions, see Section 26.3.

For more information on advisors including design and evaluation, see [23].

26.2 The `samedom` constraint

The idea how the `SameDom` propagator implements the `samedom` constraint is straightforward. The propagator creates an advisor for each of its views and as soon as the `advise()` function decides for one of the views `x` that either

$$x \subseteq d \qquad \text{or} \qquad x \cap d = \emptyset$$

holds, the propagator is informed what it has to do and is scheduled. Then, the propagator performs the required propagation and becomes subsumed. That is, a `SameDom` propagator is executed at most once.

Todo information. A `SameDom` propagator stores in `todo` what it has to do when it is executed:

TODO INFORMATION ≡
```
enum ToDo { INCLUDE, EXCLUDE, NOTHING };
ToDo todo;
```

Initially, `todo` is `NOTHING`. When the propagator is scheduled on behalf of an advisor `todo` is either `INCLUDE` (that is, the propagator must propagate that $x_i \subseteq d$ for $0 \leq i < |x|$) or `EXCLUDE` (that is, the propagator must propagate that $x_i \cap d = \emptyset$ for $0 \leq i < |x|$).

View advisors. Each advisor used by the `SameDom` propagator stores the view it is subscribed to. By this, the `advise()` function can use the view stored with an advisor to decide what the propagator needs to do. A view advisor is defined as follows:

ADVISOR ≡
```
class ViewAdvisor : public Advisor {
public:
  Int::IntView x;
  ViewAdvisor(Space& home, Propagator& p,
              Council<ViewAdvisor>& c, Int::IntView x0)
    : Advisor(home,p,c), x(x0) {
    x.subscribe(home,*this);
  }
  ViewAdvisor(Space& home, bool share, ViewAdvisor& a)
    : Advisor(home,share,a) {
    x.update(home,share,a.x);
  }
  void dispose(Space& home, Council<ViewAdvisor>& c) {
    x.cancel(home,*this);
    Advisor::dispose(home,c);
  }
};
Council<ViewAdvisor> c;
```

An advisor must implement a constructor for creation which takes the home space, the advisor's propagator, and the council of advisors as arguments. Additionally, a view advisor also takes the view as input and subscribes to the view.

An advisor does neither have an `update()` nor a `copy()` function, a constructor for cloning with the typical arguments is sufficient. The `dispose()` function of an advisor does not have to report the advisor's size (in contrast to a propagator's `dispose()` function).

The propagator maintains a council of view advisors `c`. A council controls disposal and copying during cloning. Moreover, a council provides access to all advisors in the council (where already disposed advisors are excluded). The `SameDom` propagator does not store its

SAMEDOM ≡ [DOWNLOAD]

```
...
class SameDom : public Propagator {
protected:
  ▶ TODO INFORMATION
  ▶ ADVISOR
  IntSet d;
  static ModEvent include(Space& home, Int::IntView x,
                          const IntSet& d) {
  ...
  static ModEvent exclude(Space& home, Int::IntView x,
                          const IntSet& d) {
  ...
  static ToDo dom(Int::IntView x, const IntSet& d) {
  ...
public:
  ▶ CONSTRUCTOR FOR POSTING
  ...
  ▶ DISPOSAL
  ...
  ▶ ADVISE FUNCTION
  ▶ PROPAGATION
};
...
```

Figure 26.1: A `samedom` propagator using advisors

views explicitly in a view array. As the council provides access to its advisors, all views can be accessed from the council's advisors.

Tip 26.1 (Different types of advisors for the same propagator)**.** Any propagator that uses advisors must have exactly one council. As the council depends on the advisor type, only one advisor type per propagator is possible.

This is not a real restriction. If several different types of advisors are needed, one can either use advisors with virtual member functions or encode the advisor's type by some member stored by the advisor. ◀

The propagator proper. The `SameDom` propagator is shown in Figure 26.1. The function `include(home,x,d)` constrains the view `x` to only take values from `d`, whereas `exclude(home,x,d)` constrains the view `x` to not take any values from `d`. The function `dom(x,d)` returns whether the values for `x` are included in `d` (`INCLUDE` is returned) or whether values for `x` are excluded from `d` (`EXCLUDE` is returned). All these functions are

implemented with range iterators as discussed in Chapter 25.

Posting the propagator. The propagator post function (not shown) makes sure that the propagator is only posted if for all views x_i, it cannot be decided whether $x_i \in d$ or $x_i \notin d$. If this is not the case, the post function already performs the necessary propagation. Note that the propagator post function by performing some propagation ensures the central invariant of the `SameDom` propagator: the value of `todo` (which is `NOTHING` initially) corresponds to the current domains of the propagator's views.

The constructor for posting creates the required advisors as follows:

CONSTRUCTOR FOR POSTING ≡
```
SameDom(Home home, const IntVarArgs& x, const IntSet& d0)
  : Propagator(home), todo(NOTHING), c(home), d(d0) {
  for (int i=x.size(); i--; )
    (void) new (home) ViewAdvisor(home,*this,c,x[i]);
  home.notice(*this,AP_DISPOSE);
}
```

Tip 26.2 (Getting a propagator started with advisors)**.** The `SameDom` propagator has the property that when it is posted, it is known to be at fixpoint (the `post()` function ensures this by checking for each view x_i whether $x_i \in d$ or $x_i \notin d$).

In general, it might not be true that a propagator using advisors is at fixpoint when it is posted. In that case, the constructor of the propagator must not only create advisors but also make sure that the propagator is scheduled for execution.

A propagator can be scheduled by using the static `schedule()` function of a view. For example, assume that the propagator to be scheduled should be scheduled because one of its integer views of type `IntView` is assigned. This can be achieved by:

```
IntView::schedule(home, *this, ME_INT_VAL);
```

where `*this` refers to the current propagator.

Likewise,

```
IntView::schedule(home, *this, ME_INT_BND);
```

schedules the propagator with the information that the bounds of some of its integer views have changed. ◀

Mandatory propagator disposal. The constructor also puts a notice on the propagator that it must always be disposed, even if the home space is deleted (as discussed in Section 22.8). Putting a notice is required because the integer set `d` of type `IntSet` is a proper data structure and must hence be deleted when a `SameDom` propagator is disposed.

Accordingly, the `dispose()` function deletes the integer set `d` as follows:

DISPOSAL ≡
```
virtual size_t dispose(Space& home) {
  home.ignore(*this,AP_DISPOSE);
  c.dispose(home);
  d.~IntSet();
  (void) Propagator::dispose(home);
  return sizeof(*this);
}
```

Disposing the council `c` also disposes all view advisors. Hence, also all subscriptions are canceled (as the `dispose()` function of a `ViewAdvisor` cancels its subscription).

It is essential to ignore the notice in `dispose()` by calling `home.notice()`: otherwise Gecode might attempt to dispose a now already disposed propagator just over and over again!

Propagation with advice. The `advise()` function is straightforward:

ADVISE FUNCTION ≡
```
virtual ExecStatus advise(Space&, Advisor& a, const Delta&) {
  if (todo != NOTHING)
    return ES_FIX;
  todo = dom(static_cast<ViewAdvisor&>(a).x,d);
  return (todo == NOTHING) ? ES_FIX : ES_NOFIX;
}
```

If `todo` is already different from `NOTHING`, the propagator has already been scheduled, and the `advise()` function returns `ES_FIX` to signal that the propagator does not need to be scheduled again.[1] Otherwise, depending on the result of `dom()`, the `advise()` function updates the `todo` value of the propagator and returns `ES_NOFIX` if the propagator needs scheduling (as `todo` is different from `NOTHING`).

Note that the `advise()` function of `SameDom` ignores its `Delta` argument. In the next section we will see a complementary design: the advisor does not carry any information, but only uses the information provided by the `Delta`.

The `propagate()` member function is exactly as to be expected:

PROPAGATION ≡
```
virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
  if (todo == INCLUDE)
    for (Advisors<ViewAdvisor> a(c); a(); ++a)
      GECODE_ME_CHECK(include(home,a.advisor().x,d));
  ...
}
```

[1] Actually, it would also be okay to return `ES_NOFIX`. Scheduling an already scheduled propagator is okay.

The `Advisors` class provides an iterator over all advisors in the council `c`. As mentioned earlier, the iterator (and hence the council) provides sufficient information to retrieve all views of interest for propagation.

Tip 26.3 (Advisors and propagator obligations)**.** The astute reader might have wondered whether a `SameDom` propagator is actually "update complete" in the sense of Section 22.8. The answer is yes, because the council copies all of its view advisors and each view advisor updates its view in turn.

Likewise, the obligations "subscription complete" and "subscription correct" need to be satisfied regardless of whether a propagator uses propagator condition-based or advisor-based subscriptions to views. ◀

Advisor disposal. The `SameDom` propagator leaves the disposal of its advisors to its own `dispose()` function. However, it could also request disposal of an advisor in the `advise()` function itself. A different implementation of the advise function would be:

```
if (todo != NOTHING)
  return home.ES_FIX_DISPOSE(c,static_cast<ViewAdvisor&>(a));
todo = dom(static_cast<ViewAdvisor&>(a).x,d);
return (todo == NOTHING) ? ES_FIX :
  home.ES_NOFIX_DISPOSE(c,static_cast<ViewAdvisor&>(a));
```

With this design, all advisors would be disposed on behalf of the `advise()` function and not by the propagator's `dispose()` function. This is due to the following two facts:

- A single advisor finds out that `todo` must be either `EXCLUDE` or `INCLUDE`. This advisor returns `ES_NOFIX_DISPOSE()` and hence is disposed.
- All other advisors will be executed at the very latest when the propagator performs propagation and are disposed as well.

Using predefined view advisors. Unsurprisingly, view advisors are a common abstraction for propagators using advisors. Hence, Gecode provides view advisors as predefined abstractions that are parametric with respect to their view type. Figure 26.2 shows how `SameDom` can be implemented using predefined view advisors.

26.3 General Boolean disjunction

Let us consider an efficient propagator `Or` for implementing the Boolean disjunction

$$\bigvee_{i=0}^{|x|-1} x_i = y$$

SAMEDOM USING PREDEFINED VIEW ADVISORS ≡ [DOWNLOAD]

```
...
class SameDom : public Propagator {
protected:
  ...
  Council<ViewAdvisor<Int::IntView> > c;
  IntSet d;
  ...
public:
  ...
  virtual ExecStatus advise(Space&, Advisor& a, const Delta&) {
    if (todo != NOTHING)
      return ES_FIX;
    todo = dom(static_cast<ViewAdvisor<Int::IntView>&>(a).view(),d);
    return (todo == NOTHING) ? ES_FIX : ES_NOFIX;
  }
  ...
};
...
```

Figure 26.2: A `samedom` propagator using predefined view advisors

OR ≡ [DOWNLOAD]

```
...
typedef MixNaryOnePropagator<Int::BoolView,Int::PC_BOOL_NONE,
                             Int::BoolView,Int::PC_BOOL_VAL>
        OrBase;
class Or : public OrBase {
protected:
  int n_zero;
  Council<Advisor> c;
public:
  Or(Home home, ViewArray<Int::BoolView>& x, Int::BoolView y)
    : OrBase(home,x,y), n_zero(0), c(home) {
    x.subscribe(home,*new (home) Advisor(home,*this,c));
  }
  ...
  ▶ ADVISE
  ▶ PROPAGATION
};
...
```

Figure 26.3: A Boolean disjunction propagator using advisors

where all x_i and y are Boolean views. When y becomes assigned, propagation is straightforward. If y is assigned to 0, all x_i must be assigned to 0 as well. If y is assigned to 1, Or is rewritten into the propagator OrTrue from Section 23.3.

As it comes to the x_i, the Or propagator uses a similar technique to the SameDom propagator. It can use advisors to find out which view has been assigned instead of inspecting all x_i. However, the propagator requires very little information: it does not need to know which view has changed, it only needs to know whether a view among the x_i has been assigned to 0 or 1. Our Or propagator uses the delta information passed to the advise() function to determine the value to which a view has been assigned. The advantage is that the propagator only needs a single advisor instead of one advisor per view.

Tip 26.4 (Advisor space requirements)**.** A subscription requires one pointer, regardless of whether a propagator or an advisor is subscribed to a view. Without any additional information stored by an advisor, an advisor requires two pointers. Hence, it pays off to use as few advisors as possible. ◀

The Or propagator. The Or propagator inherits from the MixNaryOnePropagator template (to increase readability, a base class OrBase is defined as a type) and uses PC_BOOL_NONE as propagation condition for the views in the view array x. That actually means that no subscriptions are created for the views in x. A subscription with propagation condition

PC_BOOL_VAL is created for the single Boolean view y. The constructor for posting creates a single advisor which subscribes to all views in x. In fact, the Or propagator mixes advisors with normal subscriptions.

The advise() function uses the delta information to decide whether one of the views the advisor has subscribed to is assigned to 0 (then Int::BoolView::zero() returns true):

ADVISE ≡
```
virtual ExecStatus advise(Space&, Advisor&, const Delta& d) {
  if (Int::BoolView::zero(d) && (++n_zero < x.size()))
    return ES_FIX;
  else
    return ES_NOFIX;
}
```

The advise function counts the number of views assigned to zero in n_zero. It reports that the propagator must be scheduled, if all views have been assigned to zero (that is, n_zero equals the number of views in x) or if a view has been assigned to one. Note that the propagator post function makes sure that the propagator is only posted when none of the views in x are assigned.

The propagate() function is straightforward:

PROPAGATION ≡
```
virtual ExecStatus propagate(Space& home, const ModEventDelta&) {
  if (y.one())
    GECODE_REWRITE(*this,OrTrue::post(home(*this),x));
  if (y.zero()) {
    for (int i = x.size(); i--; )
      GECODE_ME_CHECK(x[i].zero(home));
  } else if (n_zero == x.size()) {
    GECODE_ME_CHECK(y.zero(home));
  } else {
    GECODE_ME_CHECK(y.one(home));
  }
  return home.ES_SUBSUMED(*this);
}
```

It first checks whether the propagator has been executed because y has been assigned and propagates as sketched before. Then it uses the value of n_zero to determine how to propagate.

Delta information for integer views. The delta information provided to the advise() function can be interpreted through the view the advisor has subscribed to. Boolean views provide static member functions zero() and one() to find out whether the view has been assigned to 0 or 1.

For integer views (and set views, see Section 28.2), one must use the view to which the advisor has subscribed to for accessing the delta.

For example, suppose that the advisor `a` has subscribed to the view `x` by

```
x.subscribe(home,a);
```

When the `advise()` function is executed for the advisor `a` with delta `d`, the view `x` provides access to the delta information (typically, this will mean that `a` in fact stores the view `x`).

The modification event of the operation that modified the view `x` is available by `x.modevent(d)`. If `x.any(d)` returns true, no further information about how the domain of `x` has changed is available.

Only if `x.any(d)` is false, `x.min(d)` returns the smallest value and `x.max(d)` returns the largest value of the values that have been removed from `x`. With other words, the delta `d` only provides approximate information on how the old view domain has been changed.

For example, if `x` has the domain $\{0,\ldots,10\}$ and the value 4 is removed from `x`, then a delta `d` is passed where `x.any(d)` is true. If the values $\{0,\ldots,6\}$ are removed and the new domain is $\{7,\ldots,10\}$ then a delta `d` is passed where `x.any(d)` is false and `x.min(d)` returns 0 and `x.max(d)` returns 6.

For Boolean views, one can rely on the fact that the delta information is always accurate. For integer views, it might be the case that `x.any(d)` returns true even though only the bounds of `x` have changed.

26.4 Forced propagator rescheduling

An advisor can force its propagator to be rescheduled even though the propagator's modification event delta has not changed. As discussed in Section 25.5, the `cost()` function of a propagator is only recomputed when its modification event delta changes.

When the `advise()` of a propagator returns `ES_NOFIX_FORCE` (or, the `advise()` function calls `ES_NOFIX_DISPOSE_FORCE()`), the propagator is rescheduled regardless of its current modification event delta. See also Status of constraint propagation and branching commit.

27 Views

This chapter should come as a welcome diversion from the previous chapters in this part. Instead of introducing more concepts and techniques for programming propagators, it shows how to straightforwardly and efficiently reuse propagators for implementing several different constraints. In a way, the chapter tells you how to cache in on all the effort that goes into developing a propagator.

The idea is to make a propagator generic with respect to the views the propagator computes with. As we are talking C++, generic propagators will be nothing but templates where the template arguments happen to be view types. Then, by instantiating the template propagator, one can obtain implementations for several constraints from a single propagator. More on views (a concept introduced by Gecode) can be found in [51] and [49].

As it comes to importance, this chapter should be the second in this part. However, the chapter comes rather late to be able to draw on the example propagators presented in the previous chapters.

Overview. Integer variable views are discussed in Section 27.1 and Boolean variable views are discussed in Section 27.2. How integer propagators can be reused for Boolean views is presented in Section 27.3.

27.1 Integer views

Assume that we need an implementation for the `min` constraint. Of course, we could implement a `Min` propagator analogous to the `Max` propagator from Section 24.4. But let us assume that we need to be lazy in that we do not have the time to implement `Min` (after all, there are more interesting constraints out there that we want to implement).

What we could do to implement $\min(x, y) = z$ is to introduce three new variables x', y', and z', post three constraints such that $x = -x'$, $y = -y'$, and $z = -z'$, and finally post a `max` constraint instead: $\max(x', y') = z'$. While the strength of propagation is uncompromised, efficiency is poor: three additional variables and three additional propagators are needed.

27.1.1 Minus views

Minus views can do exactly what we discussed above but without creating additional variables or propagators. Assume that we have an integer view `x` that serves as an interface to

MIN AND MAX ≡ [DOWNLOAD]

```
...
template<class View>
class Max : public TernaryPropagator<View,Int::PC_INT_BND> {
protected:
    using TernaryPropagator<View,Int::PC_INT_BND>::x0;
    using TernaryPropagator<View,Int::PC_INT_BND>::x1;
    using TernaryPropagator<View,Int::PC_INT_BND>::x2;
  ...
};

void min(Home home, IntVar x0, IntVar x1, IntVar x2) {
  if (home.failed()) return;
  Int::MinusView y0(x0), y1(x1), y2(x2);
  GECODE_ES_FAIL(Max<Int::MinusView>::post(home,y0,y1,y2));
}
void max(Home home, IntVar x0, IntVar x1, IntVar x2) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Max<Int::IntView>::post(home,x0,x1,x2));
}
```

Figure 27.1: Minimum and maximum constraints implemented by a `Max` propagator

a variable implementation v. Then, a *minus integer view* `m` for v is also an interface to v, however the interface implements operations such that `m` is an interface to $-v$.

For example, assume that the domain of `x` is $\{-1,1,3,4,6\}$ (which also means that $v \in \{-1,1,3,4,6\}$). Then, the domain for `m` is $\{-6,-4,-3,-1,1\}$. For example, `m.min()` returns -6 (which, of course, is nothing but `-x.max()`) and the modification operation `m.gq(home,-3)` results in domains `m` $\in \{-3,-1,1\}$ and `x` $\in \{-1,1,3\}$ (which, of course, is the same as `x.lq(home,-(-3))` and hence as `x.lq(home,3)`).

The very point of this exercise is: a minus view is just a different interface to an *existing* variable implementation and does not require a *new* variable implementation. Moreover, the operations performed by the minus view interface are optimized away at compile time.

Figure 27.1 shows how to obtain both `min` and `max` constraints from the very same `Max` propagator using `Int::IntView` and `Int::MinusView` views. The only change needed compared to the `Max` propagator from Section 24.4 is that the propagator does not hardwire its view type. Instead, the propagator is generic by being implemented as a template over the view type `View` it uses. The constraint post functions then just instantiate the `Max` propagator with the appropriate view types.

Tip 27.1 (Using **`using`** clauses)**.** Note the **`using`** clauses in Figure 27.1. They make `x0`, `x1`, and `x2` visible for the `Max` propagator. This is necessary in C++ as `Max` inherits from a base

DOMAIN EQUAL WITH AND WITHOUT OFFSET ≡ [DOWNLOAD]

```
...
template<class View0, class View1>
class Equal
  : public MixBinaryPropagator<View0,Int::PC_INT_DOM,
                               View1,Int::PC_INT_DOM> {
  ...
};

void equal(Home home, IntVar x0, IntVar x1) {
  if (home.failed()) return;
  GECODE_ES_FAIL((Equal<Int::IntView,Int::IntView>
                  ::post(home,x0,x1)));
}
void equal(Home home, IntVar x0, IntVar x1, int c) {
  if (home.failed()) return;
  GECODE_ES_FAIL((Equal<Int::IntView,Int::OffsetView>
                  ::post(home,x0,Int::OffsetView(x1,c))));
}
```

Figure 27.2: Domain equality with and without offset

class that itself depends on the template argument `View` of `Max`. There are other possibilities to refer to members such as `x0` that also work, for example by writing **`this`**`->x0` instead of just `x0`. We choose the variant with **`using`** clauses to keep the code of the propagator uncluttered. ◀

.

27.1.2 Offset views

An *offset view* `o` with offset `c` (an integer value) for a variable implementation v provides operations such that `o` behaves as $v + c$.

Figure 27.2 shows how a domain equality constraint (see Section 25.2) and a domain equality constraint with offset (see Section 25.3) can be obtained from the same domain equality propagator `Equal`. `Equal` has two template arguments `View0` and `View1` for its views `x0` and `x1` respectively. With two view template arguments, the propagator can be instantiated with different view types for `x0` and `x1`. Therefore, the propagator uses `MixBinaryPropagator` as base class as it supports different view types as well.

`shared` versus `same`. The domain modification operations `inter_r` and `narrow_r` used in the `Equal` propagator from Section 25.2 are used such that the operations perform a more

efficient in-place update of the view domain (with an additional Boolean value **`false`** as last and optional argument). This is only legal because the range iterator passed as argument to the modification operations does not depend on the view being modified. The post function of `Equal` ensures this by only posting the propagator if the two views `x0` and `x1` are not referring to the very same variable implementation (that is, `same(x0,x1)` is false).

With arbitrary views, the situation becomes a little bit more involved. Assume that `x0` is an integer view referring to the variable implementation v and that `x1` is an offset integer view for the same variable implementation v and an integer value $c \neq 0$. In this case, the views `x0` and `x1` share the same variable implementation v but are *not* the same.

The function `shared()` tests whether two views share the same variable implementation. Hence, the use of domain modification operations in `Equal` have to be modified as follows:

DOMAIN PROPAGATION ≡

```
Int::ViewRanges<View0> r0(x0);
GECODE_ME_CHECK(x1.inter_r(home,r0,shared(x0,x1)));
Int::ViewRanges<View1> r1(x1);
GECODE_ME_CHECK(x0.narrow_r(home,r1,shared(x0,x1)));
```

Now, the more efficient in-place operations are used only if `x0` and `x1` do not share the same variable implementation.

27.1.3 Constant and scale views

In addition to minus and offset views, Gecode offers *scale views* and *constant views* for integer variable implementations.

A scale view for a variable implementation v with an integer scale factor a where $a > 0$ implements operations for $a \cdot v$. Scale views exist in two variants differing in the precision of multiplication: `IntScaleView` performs multiplication over integers, whereas `LLongScaleView` performs multiplication over long long integers (see Integer views and `Int::ScaleView`).

An integer constant view `Int::ConstIntView` provides an integer view interface to an integer constant `c`. With other words, an integer constant view for the integer `c` behaves as an integer view assigned to the value `c`.

27.2 Boolean views

For Boolean views, the view resembling a minus view over integers is a view for negation. For example, with Boolean negation views `Int::NegBoolView` both disjunction and conjunction constraints can be obtained from a propagator for disjunction (see Figure 27.3).

OR AND AND FROM OR ≡ [DOWNLOAD]

```
  ...
  template<class View>
  class OrTrue :
    public BinaryPropagator<View,Int::PC_BOOL_VAL> {
    ...
  };

  void dis(Home home, const BoolVarArgs& x, int n) {
    ...
  }

  void con(Home home, const BoolVarArgs& x, int n) {
    ...
    } else {
      ViewArray<Int::NegBoolView> y(home,x.size());
      for (int i=x.size(); i--; )
        y[i]=Int::NegBoolView(x[i]);
      GECODE_ES_FAIL(OrTrue<Int::NegBoolView>::post(home,y));
    }
  }
```

Figure 27.3: Disjunction and conjunction from same propagator

LESS FOR INTEGER AND BOOLEAN VARIABLES ≡ [DOWNLOAD]

```
...
template<class View>
class Less : public BinaryPropagator<View,Int::PC_INT_BND> {
protected:
  ...
};

void less(Home home, IntVar x0, IntVar x1) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Less<Int::IntView>::post(home,x0,x1));
}
void less(Home home, BoolVar x0, BoolVar x1) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Less<Int::BoolView>::post(home,x0,x1));
}
```

Figure 27.4: Less constraints for both integer and Boolean variables

27.3 Integer propagators on Boolean views

As has been discussed in Section 23.2, Boolean views feature all operations available on integer views (such as `lq()` or `gr()`) in addition to the dedicated Boolean operations (such as `one()` or `zero()`). Due to the availability of integer operations on Boolean views, integer propagators can be used to implement Boolean constraints.

Figure 27.4 shows how the propagator `Less` can be used to implement the `less` constraint for both integer and Boolean variables.

Tip 27.2 (Boolean variables are not integer variables)**.** The above example uses a template to obtain an implementation of a constraint on both integer and Boolean variables. This is necessary as Boolean variables are not integer variables (in the sense that `BoolVar` is not a subclass of `IntVar`). The same holds true for their views and variable implementations.

This is by design: Boolean variables are not integer variables as they have a specially optimized implementation (taking advantage of the limited possible variable domains and that only `PC_BOOL_VAL` as propagation condition is needed). ◀

28 Propagators for set constraints

This chapter shows how to implement propagators for constraints over set variables. We assume that you have worked through the chapters on implementing integer propagators, as most of the techniques readily carry over and are not explained again here.

We also assume a basic knowledge of propagation for set constraints. To read more about this topic, please refer to [18, 56].

Overview. Section 28.1 demonstrates a propagator that implements set interesection. Set views and their related concepts are summarized in Section 28.2.

28.1 A simple example

Figure 28.1 shows a propagator for the ternary intersection constraint $x_0 \cap x_1 = x_2$ for three set variables x_0, x_1, and x_2.

As you can see, propagators for set constraints follow exactly the same structure as propagators for integer or Boolean constraints. The same propagator patterns can be used (see Section 22.7). The appropriate views and propagation conditions are defined in the namespace `Gecode::Set`.

In order to understand the `propagate()` function, we have to look at how set variable domains are represented.

The set bounds approximation. We already saw in Chapter 5 that set variable domains are represented as intervals in order to avoid an exponential representation. For example, recall that

$$\{\ \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}\ \}$$

cannot be captured exactly by an interval, but is instead approximated by the smallest enclosing interval $[\{\} .. \{1,2,3\}]$.

Set propagators therefore access and modify the interval bounds. Naturally, set-valued domain operations similar to the ones for integer variables (see Chapter 25) play an important role for set propagators.

For each set view, `Set::GlbRanges` provides a range iterator for its lower bound, and `Set::LubRanges` iterates the upper bound. The main iterator-based modification operations

INTERSECTION ≡ [DOWNLOAD]

```
#include <gecode/set.hh>
using namespace Gecode;

class Intersection
  : public TernaryPropagator<Set::SetView,Set::PC_SET_ANY> {
public:
  Intersection(Home home, Set::SetView x0, Set::SetView x1,
               Set::SetView x2)
    : TernaryPropagator<Set::SetView,Set::PC_SET_ANY>(home,
                                                      x0,x1,x2) {}
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
    using namespace Iter::Ranges; using namespace Set;
    bool assigned = x0.assigned() && x1.assigned() && x2.assigned();
    ▶ RULE 1
    ▶ RULE 2
    ▶ RULE 3
    ▶ RULE 4
    ▶ RULE 5
    ▶ RULE 6
    ▶ CARDINALITY
    return assigned ? home.ES_SUBSUMED(*this) : ES_NOFIX;
  }
};

void intersection(Home home, SetVar x0, SetVar x1, SetVar x2) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Intersection::post(home,x0,x1,x2));
}
```

Figure 28.1: A constraint and propagator for set intersection

on set views are `includeI` (adding a set to the lower bound), `excludeI` (removing a set from the upper bound), and `intersectI` (intersecting the upper bound with a set).

Filtering rules. Coming back to the example propagator for ternary intersection, we have to devise filtering rules that express the constraint in terms of the interval bounds. In the following, we write $\underline{x}$ and $\overline{x}$ for the lower bound resp. upper bound of a view x. Then, ternary intersection can be propagated with the following rules and implemented with set domain operations:

1. $\underline{x_0} \cap \underline{x_1} \subseteq x_2$

```
RULE 1 ≡
  {
    GlbRanges<SetView> x0lb(x0), x1lb(x1);
    Inter<GlbRanges<SetView>, GlbRanges<SetView> > i(x0lb,x1lb);
    GECODE_ME_CHECK(x2.includeI(home,i));
  }
```

2. $\overline{x_0} \cap \overline{x_1} \supseteq x_2$

```
RULE 2 ≡
  {
    LubRanges<SetView> x0ub(x0), x1ub(x1);
    Inter<LubRanges<SetView>, LubRanges<SetView> > i1(x0ub,x1ub);
    GECODE_ME_CHECK(x2.intersectI(home,i1));
  }
```

3. $\underline{x_2} \subseteq x_0$

```
RULE 3 ≡
  {
    GlbRanges<SetView> x2lb(x2);
    GECODE_ME_CHECK(x0.includeI(home,x2lb));
  }
```

4. $\underline{x_2} \subseteq x_1$

```
RULE 4 ≡
  {
    GlbRanges<SetView> x2lb(x2);
    GECODE_ME_CHECK(x1.includeI(home,x2lb));
  }
```

	integer-valued bounds operations
`glbMin()` / `glbMax()`	return minimum/maximum of lower bound
`lubMin()` / `lubMax()`	return minimum/maximum of upper bound
`glbSize()` / `lubSize()`	return size of lower/upper bound
`unknownSize()`	return number of elements in upper but not in lower bound
`contains()`	test whether lower bound contains element
`notContains()`	test whether upper bound does not contain element
`include()`	add element (or range) to lower bound
`exclude()`	remove element (or range) from upper bound
`intersect()`	intersect upper bound with element or range
	set-valued bounds modifications
`includeI()`	add elements to lower bound
`excludeI()`	remove elements from upper bound
`intersectI()`	intersect upper bound with given set
	cardinality operations
`cardMin()`	return/modify minimum cardinality
`cardMax()`	return/modify maximum cardinality

Figure 28.2: Set view operations

5. $\underline{x_0} \setminus \overline{x_2} \not\subseteq x_1$

```
RULE 5 ≡
  {
    GlbRanges<SetView> x0lb(x0); LubRanges<SetView> x2ub(x2);
    Diff<GlbRanges<SetView>, LubRanges<SetView> > diff(x0lb, x2ub);
    GECODE_ME_CHECK(x1.excludeI(home,diff));
  }
```

6. $\underline{x_1} \setminus \overline{x_2} \not\subseteq x_0$

```
RULE 6 ≡
  {
    GlbRanges<SetView> x1lb(x1); LubRanges<SetView> x2ub(x2);
    Diff<GlbRanges<SetView>, LubRanges<SetView> > diff(x1lb, x2ub);
    GECODE_ME_CHECK(x0.excludeI(home,diff));
  }
```

The first four rules should be self-explanatory. The last two rules state that anything that is in x_0 but not in x_2 cannot be in x_1 (and the same for x_0 and x_1 swapped). The full list of operations on set views appears in Figure 28.2.

Fixpoint. Note how the propagator determines which execution status to return. Before applying any of the filtering rules, it checks whether all of the variables are already assigned. If they are, then propagation will compute a fixpoint and the propagator can return that it is subsumed after applying the filtering rules. Otherwise, it has not necessarily computed a fixpoint (e.g. rule 6 may modify the upper bound of x_0, making it necessary to apply rule 2 again).

Cardinality. In addition to the interval bounds, set variables store *cardinality bounds*, that is, the minimum and maximum cardinality of the set variable. These bounds are stored and modified independently of the interval bounds, but of course modifications to these different bounds affect each other.

For example, consider a set variable with a domain represented by the interval $[\{\} .. \{1,2\}]$ and the cardinality $\#[1 .. 2]$. Adding 1 to the lower bound would result in the cardinality lower bound being increased to 1. Removing 1 from the upper bound would result in 2 being added to the lower bound to satisfy the minimum cardinality of 1.

Using cardinality information, propagation for some set constraints can be strengthened. For the ternary intersection example, we can for instance add the following filtering rules:

CARDINALITY ≡
```
LubRanges<SetView> x0ub(x0), x1ub(x1);
Union<LubRanges<SetView>, LubRanges<SetView> > u_lub(x0ub,x1ub);
unsigned int s_lub = size(u_lub);
if (x0.cardMin() + x1.cardMin() > s_lub)
  GECODE_ME_CHECK(x2.cardMin(home, x0.cardMin()+x1.cardMin()-s_lub));
GlbRanges<SetView> x0lb(x0), x1lb(x1);
Union<GlbRanges<SetView>, GlbRanges<SetView> > u_glb(x0lb,x1lb);
unsigned int s_glb = size(u_glb);
GECODE_ME_CHECK(x2.cardMax(home,x0.cardMax()+x1.cardMax()-s_glb));
GECODE_ME_CHECK(x0.cardMin(home,x2.cardMin()));
GECODE_ME_CHECK(x1.cardMin(home,x2.cardMin()));
```

When dealing with cardinality, it is important to handle overflow or signedness issues. In the above example, we have to check whether `x0.cardMin()+x1.cardMin()>s`, because otherwise the expression `x0.cardMin()+x1.cardMin()-s` may underflow (as we are dealing with unsigned integers here). This is not the case for the second cardinality rule. Here, we can be sure that the size of the union of the lower bounds is always greater than the sum of the maximum cardinalities.

28.2 Modification events, propagation conditions, views, and advisors

This section summarizes how these concepts are specialized for set variables and propagators.

Modification events and propagation conditions. The modification events and propagation conditions for set propagators (see Figure 28.3) capture the parts of a set variable domain that can change.

One could imagine a richer set, for example distinguishing between lower and upper bound changes of the cardinality, or separating the cardinality changes from the interval bound changes. However, the number of propagation conditions has a direct influence on the size of a variable, see Section 34.1. Just like for integer views, this set of modification events and propagation conditions has been chosen as a compromise between expressiveness on the one hand, and keeping the set small on the other.

Set variable views. In addition to the basic `Set::SetView` class, there are five other set views: `Set::ConstSetView`, `Set::EmptyView`, `Set::UniverseView`, `Set::SingletonView`, and `Set::ComplementView`.

The first three are constant views. A `SingletonView` wraps an integer view x in the interface of a set view, so that it acts like the singleton set $\{x\}$. A `ComplementView` is like Boolean negation, it provides the set complement with respect to the global Gecode universe for set variables (defined as [`Set` :: `Limits` :: `min` .. `Set` :: `Limits` :: `max`], see `Set::Limits`).

Advisors for set propagators. Advisors for set constraints get informed about the domain modifications using a `Set::SetDelta`. The set delta provides only information about the minimum and maximum values that were added to the lower bound and/or removed from the upper bound.

set modification events	
`Set::ME_SET_NONE`	the view has not been changed
`Set::ME_SET_FAILED`	the domain has become empty
`Set::ME_SET_VAL`	the view has been assigned to a single set
`Set::ME_SET_CARD`	the view has been assigned to a single set
`Set::ME_SET_LUB`	the upper bound has been changed
`Set::ME_SET_GLB`	the lower bound has been changed
`Set::ME_SET_BB`	both bounds have been changed
`Set::ME_SET_CLUB`	cardinality and upper bound have changed
`Set::ME_SET_CGLB`	cardinality and lower bound have changed
`Set::ME_SET_CBB`	cardinality and both bounds have changed

set propagation conditions	
`Set::PC_SET_VAL`	schedule when the view is assigned
`Set::PC_SET_CARD`	schedule when the cardinality changes
`Set::PC_SET_CLUB`	schedule when the cardinality or the upper bound changes
`Set::PC_SET_CGLB`	schedule when the cardinality or the lower bound changes
`Set::PC_SET_ANY`	schedule at any change
`Set::PC_SET_NONE`	do not schedule

Figure 28.3: Set modification events and propagation conditions

29 Propagators for float constraints

This chapter shows how to implement propagators for constraints over float variables. We assume that you have worked through the chapters on implementing integer propagators, as most of the techniques readily carry over and are not explained again here.

Overview. Section 29.1 demonstrates a propagator that implements a ternary linear constraint. Float views and their related concepts are summarized in Section 29.2.

29.1 A simple example

Figure 29.1 shows a propagator for the ternary linear constraint $x_0 + x_1 + x_2 = 0$ for three float variables x_0, x_1, and x_2.

As you can see, propagators for float constraints follow exactly the same structure as propagators for integer or Boolean constraints. The same propagator patterns can be used (see Section 22.7). The appropriate views and propagation conditions are defined in the namespace `Gecode::Float`.

Operations on float views. The most important operations on float views for programming propagators are summarized in Figure 29.2, the full information can be found in `Float::FloatView`. The lack of operations such as `gr()` (for greater), `le()` (for less), and `nq()` (for disequality) is due to the fact that domains are closed intervals, see Section 6.1 and Tip 6.4.

Creating a rounding object. The propagation rules of the `Linear` propagator will require that it can be controlled whether to round downwards or upwards in a floating point operation on a float number. Access to operations with explicit rounding control is provided by an object of class `Float::Rounding`. The creation of an object of this class initializes the underlying floating point unit such that it performs exact rounding in the required direction. Note, that explicit rounding is only required if rounding provided by operations on float values is not sufficient.

Figure 29.3 lists the supported operations with explicit rounding, where the `_down()` variants round downwards and the `_up()` variants round upwards. The functions marked

LINEAR ≡ [DOWNLOAD]

```
#include <gecode/float.hh>
using namespace Gecode;

class Linear
  : public TernaryPropagator<Float::FloatView,Float::PC_FLOAT_BND> {
public:
  Linear(Home home, Float::FloatView x0, Float::FloatView x1,
                    Float::FloatView x2)
    : TernaryPropagator<Float::FloatView,Float::PC_FLOAT_BND>
        (home,x0,x1,x2) {}
  ...
  virtual ExecStatus propagate(Space& home, const ModEventDelta&)  {
    using namespace Float;
    ▶ CREATE ROUNDING OBJECT
    ▶ PRUNE UPPER BOUNDS
    ▶ PRUNE LOWER BOUNDS
    return (x0.assigned() && x1.assigned()) ?
      home.ES_SUBSUMED(*this) : ES_NOFIX;
  }
};

void linear(Home home, FloatVar x0, FloatVar x1, FloatVar x2) {
  if (home.failed()) return;
  GECODE_ES_FAIL(Linear::post(home,x0,x1,x2));
}
```

Figure 29.1: A constraint and propagator for ternary linear

access operations

`min()`	return lower bound (a float number)
`max()`	return upper bound (a float number)
`size()`	return width of domain (a float number)
`assigned()`	whether view is assigned
`in(n)`	whether float number `n` is contained in domain
`in(n)`	whether float value `n` is contained in domain

modification operations

`eq(home,n)`	restrict values to be equal to `n`
`lq(home,n)`	restrict values to be less or equal than `n`
`gq(home,n)`	restrict values to be greater or equal than `n`

Figure 29.2: Most important float view operations

function	meaning	default
`add_down(x,y)`, `add_up(x,y)`	l/u bound of $x+y$	✓
`sub_down(x,y)`, `sub_up(x,y)`	l/u bound of $x-y$	✓
`mul_down(x,y)`, `mul_up(x,y)`	l/u bound of $x \times y$	✓
`div_down(x,y)`, `div_up(x,y)`	l/u bound of x/y	✓
`sqrt_down(x)`, `sqrt_up(x)`	l/u bound of $\sqrt{x}$	✓
`int_down(x)`	next downward-rounded integer of `x`	✓
`int_up(x)`	next upward-rounded integer of `x`	✓
`exp_down(x)`, `exp_up(x)`	l/u bound of exp(x)	
`log_down(x)`, `log_up(x)`	l/u bound of log(x)	
`sin_down(x)`, `sin_up(x)`	l/u bound of sin(x)	
`cos_down(x)`, `cos_up(x)`	l/u bound of cos(x)	
`tan_down(x)`, `tan_up(x)`	l/u bound of tan(x)	
`asin_down(x)`, `asin_up(x)`	l/u bound of arcsin(x)	
`acos_down(x)`, `acos_up(x)`	l/u bound of arccos(x)	
`atan_down(x)`, `atan_up(x)`	l/u bound of arctan(x)	
`sinh_down(x)`, `sinh_up(x)`	l/u bound of sinh(x)	
`cosh_down(x)`, `cosh_up(x)`	l/u bound of cosh(x)	
`tanh_down(x)`, `tanh_up(x)`	l/u bound of tanh(x)	
`asinh_down(x)`, `asinh_up(x)`	l/u bound of arcsinh(x)	
`acosh_down(x)`, `acosh_up(x)`	l/u bound of arccosh(x)	
`atanh_down(x)`, `atanh_up(x)`	l/u bound of arctanh(x)	

Figure 29.3: Rounding operations on float numbers (`x` and `y` are float numbers)

as default are always supported, the others are only supported if Gecode has been built accordingly, see Tip 6.1.

Hence, the first thing that the `propagate()` function of the `Linear` propagator does is to create a rounding object `r` as follows:

```
CREATE ROUNDING OBJECT ≡
  Rounding r;
```

Pruning lower and upper bounds. The propagation rules for `Linear` are quite straightforward. As $x_0 + x_1 + x_2 = 0$ we can isolate x_0 (x_1 and x_2 are of course analogous):

$$x_0 = -x_1 - x_2$$

The upper bound of x_0 can be constrained following:

$$\begin{aligned} x_0 &\leq \max\left(-x_1 - x_2\right) \\ &= -\min\left(x_1 + x_2\right) \\ &= -\min\left(\min(x_1) + \min(x_2)\right) \end{aligned}$$

and, accordingly, the lower bound of x_0 can be constrained following:

$$\begin{aligned} x_0 &\geq \min\left(-x_1 - x_2\right) \\ &= -\max\left(x_1 + x_2\right) \\ &= -\max\left(\max(x_1) + \max(x_2)\right) \end{aligned}$$

The equations can be translated directly into update operations, where min corresponds to rounding downwards:

```
PRUNE UPPER BOUNDS ≡
  GECODE_ME_CHECK(x0.lq(home,-r.add_down(x1.min(),x2.min())));
  GECODE_ME_CHECK(x1.lq(home,-r.add_down(x0.min(),x2.min())));
  GECODE_ME_CHECK(x2.lq(home,-r.add_down(x0.min(),x1.min())));
```

and max corresponds to rounding upwards:

```
PRUNE LOWER BOUNDS ≡
  GECODE_ME_CHECK(x0.gq(home,-r.add_up(x1.max(),x2.max())));
  GECODE_ME_CHECK(x1.gq(home,-r.add_up(x0.max(),x2.max())));
  GECODE_ME_CHECK(x2.gq(home,-r.add_up(x0.max(),x1.max())));
```

29.2 Modification events, propagation conditions, views, and advisors

This section summarizes how these concepts are specialized for float variables and propagators.

float modification events	
`Float::ME_FLOAT_NONE`	the view has not been changed
`Float::ME_FLOAT_FAILED`	the domain has become empty
`Float::ME_FLOAT_VAL`	the view has been assigned
`Float::ME_FLOAT_BND`	the bounds have changed (the domain has changed)

float propagation conditions	
`Float::PC_FLOAT_VAL`	schedule when the view is assigned
`Float::PC_FLOAT_BND`	schedule when the domain changes
`Float::PC_FLOAT_NONE`	do not schedule

Figure 29.4: Float modification events and propagation conditions

Modification events and propagation conditions. The modification events and propagation conditions for float propagators (see Figure 29.4) capture how the variable domain of a float view can change.

Float variable views. In addition to the basic `Float::FloatView` class, there are two other float views: `Float::MinusView`, and `Float::ScaleView`. The two latter views are defined similarly to minus view for integers (see Section 27.1.1) and scale views for integers (see Section 27.1.3).

Advisors for float propagators. Advisors for float constraints get informed about the domain modifications using a float delta of class `Float::FloatDelta`.

Float deltas are also represented by a minimum and maximum float number and hence also constitute a closed interval (like float values and float variables). That means that a float delta cannot describe exactly which values have been removed. For example, assume that `x` is a float view and that the domain of `x` is $[-1.0 .. 1.0]$. Then, executing

```
GECODE_ME_CHECK(x.lq(home,0.0));
```

will generate a float delta `d` such that `x.min(d)` returns `0.0` and `x.max(d)` returns `1.0` even though the domain of `x` is now $[-1.0 .. 0.0]$ and still includes 0.0.

30 Managing memory

This chapter provides an overview of memory management for propagators. In fact, the memory management aspects discussed here are also valid for branchers (Part B) and to some extent even for variables (Part V).

Overview. Section 30.1 describes the different memory areas available in Gecode together with their allocation policies. The following section (Section 30.2) discusses how a propagator can efficiently manage its own state. Section 30.3 discusses an abstraction for sharing data structures among all spaces for one thread, whereas Section 30.4 discusses an abstraction for sharing data structures among several propagators that belong to the same space.

30.1 Memory areas

Gecode manages several different memory areas with different allocation policies: *spaces* provide access to space-allocated memory, *regions* provide access to temporary memory with implicit deallocation, and *space-allocated freelists* provide efficient access to small chunks of memory which require frequent allocation and deallocation.

All memory areas but freelists provide operations `alloc()`, `realloc()`, and `free()` for allocation, reallocation, and deallocation of memory from the respective area. To provide a uniform interface, Gecode also defines a `heap` object (see Heap memory management) implementing the very same allocation functions. All memory-management related operations are documented in Memory management.

Memory management functions. Let us consider allocation from the `heap` as an example. By

```
int* i = heap.alloc<int>(n);
```

a memory chunk for `n` integers is allocated. Likewise, by

```
heap.free<int>(i,n);
```

the memory is freed (for `heap`, the memory is returned to the operating system). By

```
int* j = heap.realloc<int>(i,n,m);
```

a memory chunk is allocated for `m` integers. If possible, `j` will refer to the same memory chunk as `i` (but there is no guarantee, of course).

The memory management functions implement C++ semantics: `alloc()` calls the default constructor for each element allocated; `realloc()` calls the destructor, default constructor, or copy constructor (depending on whether the memory area must shrink or grow); `free()` calls the destructor for each element freed.

Space. Space-allocated memory (see Space-memory management) is managed per each individual space. All space-allocated memory is returned to the operating system if a space is deleted. Freeing memory (via the `free()` operation) enables the memory to be reused by later `alloc()` operations.

Spaces manage several blocks of memory allocated from the operating system, the block sizes are automatically chosen based on the recent memory allocations of a space (and, if a space has been created by cloning, the memory allocation history of the original space). The exact policies can be configured, see the namespace `MemoryConfig`.

Memory chunks allocated from the operating system for spaces are cached among all spaces for a single thread. This cache for the current thread can be flushed by invoking the `flush()` member function of `Space`.

Variable implementations, propagators, branchers, view arrays, for example, are allocated from their `home` space. An important characteristic is that all these data structures have fixed size throughout their lifetimes or even shrink. As space-allocated memory is managed for later reusal if it is freed, frequent allocation/reallocation/deallocation leads to highly fragmented memory with little chance of reusal. Hence, space-allocated memory is *absolutely unsuited* for data structures that require frequent allocation/deallocation and/or resizing. For these data structures it is better to use the heap or freelists, if possible. See Section 30.2 for more information.

Note that space-allocated memory is available with allocators compatible with the C++ STL, see Using allocators with Gecode.

Region. All spaces for one thread share a chunk of memory for temporary data structures with very efficient allocation and deallocation (again, its exact size is defined in the namespace `MemoryConfig`). Assume that `home` refers to a space (say, during the execution of the `propagate()` function of a propagator), then

```
Region r(home);
```

creates a new region `r` for memory management (see Region memory management). A region does not impose any size limit on the memory blocks allocated from it. If necessary, a region transparently falls back to heap-allocated memory.

Several regions for the same space can exist simultaneously. For example,

```
Region r1(home);
int* i = r1.alloc<int>(n);
{
  Region r2(home);
  int* j = r2.alloc<int>(m);
  ...
}
```

However, it is essential that regions are used in a stack fashion: while region `r2` is in use, no allocation from any previous region (`r1` in our example) is allowed. That is, the following code will result in a crash (if we are lucky):

```
Region r1(home), r2(home);
int* j = r2.alloc<int>(m);
int* i = r1.alloc<int>(n);
```

The speciality of a region is that it does not require `free()` operations. If a region is destructed, all memory allocated from it is freed (unless several regions for the same space exist). If several regions for the same space are in use, the memory is entirely freed when all regions are destructed.

Even though a region does not require `free()` operations, it can profit from it: if the memory allocated last is freed first, the freed memory becomes immediately available for future allocation.

Tip 30.1 (Keep the scope of a region small)**.** In order to make best use of the memory provided by a region it is good practice to keep the scope of regions as small as possible. For example, suppose that in the following example

```
Region r(home);
{
  int* i = r.alloc<int>(n);
  ...
}
{
  int* i = r.alloc<int>(n);
  ...
}
```

the memory allocated for `i` is not used outside the two blocks. Then it is in fact better to rewrite the code to:

```
{
  Region r(home);
  int* i = r.alloc<int>(n);
  ...
}
{
  Region r(home);
  int* i = r.alloc<int>(n);
  ...
}
```

Then both blocks will have access to the full memory of the region. Furthermore, the creation of a region is very efficient and the code might even benefit from better cache behavior. ◀

Heap. The `heap` (a global variable, see Heap memory management) is nothing but a C++-wrapper around `malloc()` and `free()` as provided by the underlying operating system. In case memory is exhausted, an exception of type `MemoryExhausted` is thrown.

Space-allocated freelists. Freelists are allocated from space memory. Any object to be managed by a freelist must inherit from the class `FreeList` which already defines a pointer to a next freelist element. The sizes for freelist objects are quite constrained, check the values `fl_size_min` and `fl_size_max` as defined in the namespace `MemoryConfig`.

Allocation and deallocation is available through the member functions `fl_alloc()` and `fl_dispose()` of the class `Space`.

30.2 Managing propagator state

Many propagators require sophisticated data structures to perform propagation. The data structures are typically kept between successive executions of a propagator. There are two main issues for these data structures: *where* to allocate them and *when* to allocate them.

Where to allocate. Typically, the data structures used by a propagator are of dynamic size and hence cannot be stored in a simple member of the propagator. This means that the propagator is free to allocate the memory from either its own space or from the heap. Allocation from the heap could also mean to use other operations to allocate and free memory, such as `malloc()` and `free()` provided by the operating system or **`new`** and **`delete`** provided by C++.

In case the data structure does not change its size often, it is best to allocate from a space: allocation is more efficient and deallocation is automatic when the space is deleted.

In case the data structure requires frequent reallocation operations, it is better to allocate from the heap. Then, the memory will not be automatically freed when a space is deleted. The memory must be freed by the propagator's `dispose()` function. Furthermore,

the Gecode kernel must be informed that the `dispose()` function must be called when the propagator's home space is deleted, see Section 22.9.

When to allocate. An important fact on which search engines in Gecode rely (see Part S) is that they always store a space created by cloning for backtracking and never a space that has already been used for propagation. The reason is that in order to perform propagation, the propagators, variables, and branchers might require some additional memory. Hence, a space that has performed propagation is likely to require more memory than a pristine clone. As the spaces stored by a search engine define the total amount of memory allocated for solving a problem with Gecode, it pays to save memory by storing pristine clones.

The most obvious policy is to allocate *eagerly*: the data structures are allocated when the propagator is created and they are copied exactly when the propagator is copied.

However, very often it is better to *lazily* recompute the data structures as follows. The data structures are initialized and allocated the first time a propagator is executed. Likewise, the data structures are not copied and construction is again postponed until the propagator is executed again. Creating and allocating data structures lazily is often as simple as creating them eagerly. The benefit is that clones of spaces on which no propagation has been performed yet require considerably less memory.

Most propagators in Gecode that require involved data structures construct their data structures lazily (for example, the propagator for domain consistent `distinct` using the algorithm from [41], see `Int::Distinct::Dom`). Some use a hybrid approach where some data structures are created eagerly and others lazily (for example the propagator for `extensional` for finite automata using the algorithm from [36], see `Int::Extensional::LayeredGraph`).

30.3 Shared objects and handles

A common request for managing a data structure (we will refer to data structure here just as object) is that it is used and shared by several propagators from different spaces with the restriction that all spaces are used by a single thread. Gecode provides *shared objects* and *shared handles* for this form of sharing. Shared handles and objects are used to implement proper data structures as introduced in Section 4.2.

A shared object is heap-allocated and is referred to by several shared handles. If the last shared handle is deleted, also the shared object is deleted (implemented by reference counting). In addition, shared handles provide an `update()` function that creates a new copy of the shared object in case the copy is to be used by a different thread (that is, the Boolean `share` argument is true).

Figure 30.1 shows a simple example of a shared object and the shared handle using the object. A shared object `SIO` (for `S`hared `I`nteger `O`bject) stores a single integer `data` to be shared (normally, this will be an interesting data structure). A shared object must inherit from `SharedHandle::Object` and must implement a virtual `copy()` function that returns a new shared object. If needed, a virtual destructor can be defined.

SHARED OBJECT AND HANDLE ≡ [DOWNLOAD]

```
...
class SI : public SharedHandle {
protected:
  class SIO : public SharedHandle::Object {
  public:
    int data;
    SIO(int d)
      : data(d) {}
    SIO(const SIO& sio)
      : data(sio.data) {}
    virtual Object* copy(void) const {
      return new SIO(*this);
    }
    virtual ~SIO(void) {}
  };
public:
  SI(int d)
    : SharedHandle(new SIO(d)) {}
  SI(const SI& si)
    : SharedHandle(si) {}
  SI& operator =(const SI& si) {
    return static_cast<SI&>(SharedHandle::operator =(si));
  }
  int get(void) const {
    return static_cast<SIO*>(object())->data;
  }
  void set(int d) {
    static_cast<SIO*>(object())->data = d;
  }
  ▶ SOME INHERITED MEMBERS
};
```

Figure 30.1: A simple shared object and handle

The shared handle `SI` (for Shared Integer) is the only class that has access to a shared object of type `SIO`. A shared handle must inherit from `SharedHandle` and can use the `object()` member function to access and update its shared object.

The most interesting members of `SI` that are inherited are the following:

SOME INHERITED MEMBERS ≡
```
void update(Space& home, bool share, SharedHandle& sh) {
  ...
}
~SI(void) {}
```

Other inherited members include a copy constructor and an assignment operator.

Shared integer arrays are an example for shared objects, see Tip 4.9.

30.4 Local objects and handles

For some applications, it is necessary to share data structures between different propagators (and/or branchers, see Part B) that belong to the same space. For example, several scheduling propagators might record precedences between tasks in a shared data structure. *Local objects* and *local handles* implement this form of sharing.

A local object is space-allocated and is referred to by several local handles. A local object is deleted when its `home` space is deleted. Local handles provide an `update()` function that creates a new copy of the local object when the space is cloned (while maintaining the sharing of local objects within a space).

Figure 30.2 shows a simple local object and handle. Similar to the shared integer objects from Figure 30.1, the local integer objects here provide shared access to a single integer. However, this integer object is copied whenever the space is copied, so changes to the object are kept within the same space.

If a local object additionally allocates external resources or non-space allocated memory, it must inform its `home` space about this fact, very much like propagators (see Section 22.9). Figure 30.3 shows a local object and handle that implement an array of integers. In the constructor and the `dispose` function, the local object registers and de-registers itself for disposal using `AP_DISPOSE`.

LOCAL OBJECT AND HANDLE ≡ [DOWNLOAD]

```
...
class LI : public LocalHandle {
protected:
  class LIO : public LocalObject {
  public:
    int data;
    LIO(Space& home, int d) : LocalObject(home), data(d) {}
    LIO(Space& home, bool share, LIO& l)
      : LocalObject(home,share,l), data(l.data) {}
    virtual LocalObject* copy(Space& home, bool share) {
      return new (home) LIO(home,share,*this);
    }
    virtual size_t dispose(Space&) { return sizeof(*this); }
  };
public:
  LI(Space& home, int d)
    : LocalHandle(new (home) LIO(home,d)) {}
  LI(const LI& li)
    : LocalHandle(li) {}
  LI& operator =(const LI& li) {
    return static_cast<LI&>(LocalHandle::operator =(li));
  }
  int get(void) const {
    return static_cast<LIO*>(object())->data;
  }
  void set(int d) {
    static_cast<LIO*>(object())->data = d;
  }
};
```

Figure 30.2: A simple local object and handle

LOCAL OBJECT WITH EXTERNAL RESOURCES ≡ [DOWNLOAD]

```
...
class LI : public LocalHandle {
protected:
  class LIO : public LocalObject {
  public:
    int* data;
    int n;
    LIO(Space& home, int n0)
     : LocalObject(home), data(heap.alloc<int>(n0)), n(n0) {
      home.notice(*this,AP_DISPOSE);
    }
    LIO(Space& home, bool share, LIO& l)
      : LocalObject(home,share,l), data(l.data) {}
    virtual LocalObject* copy(Space& home, bool share) {
      return new (home) LIO(home,share,*this);
    }
    virtual size_t dispose(Space& home) {
      home.ignore(*this,AP_DISPOSE);
      heap.free<int>(data,n);
      return sizeof(*this);
    }
  };
public:
  ...
  int operator [](int i) const {
    return static_cast<const LIO*>(object())->data[i];
  }
  int& operator [](int i) {
    return static_cast<LIO*>(object())->data[i];
  }
};
```

Figure 30.3: A local object and handle with external resources

Programming branchers

Christian Schulte

This part presents how to program branchers as implementations of branchings.

Chapter 31 (Getting started) shows how to implement simple branchers. More advanced topics are discussed in Chapter 32 (Advanced topics).

How to implement a whole array of variable-value branchings (a need that typically arises when implementing new variables) is discussed in Chapter 37 of Part B.

31 Getting started

This chapters presents how to program branchers as implementations of branchings. It focuses on the basic concepts for programming branchers, more advanced topics are discussed in Chapter 32.

Important. You need to read about search and programming propagators before reading this chapter. For search, you need to read Chapter 9 first, in particular Section 9.1. For programming propagators, the initial Chapter 22 is sufficient.

Overview. Section 31.1 sets the stage for programming branchers by explaining how search engines, spaces, and branchers together implement the branching process during search. A simple brancher is used as an initial example in Section 31.2. A brancher that chooses its views for branching according to some criterion is demonstrated in Section 31.3.

31.1 What to implement?

A *branching* is used in modeling for describing the shape of the search tree. A *brancher* implements a branching. That is, a branching is similar to a constraint, whereas a brancher is similar to a propagator. Branchings take variables as arguments and branchers compute with variable views.

Brancher order. Creating a brancher registers it with its home space. A space maintains a queue of its branchers in that the brancher that is registered first is also used first for branching. The first brancher in the queue of branchers is referred to as the *current brancher*.

Executing branchers. A brancher in Gecode is implemented as a subclass of the class `Brancher`. Similar to a propagator, a brancher must implement several virtual member functions defining the brancher's behavior.

The most important functions of a brancher can be sketched as follows:

- The `status()` function tests whether the current brancher has anything left to do.
- The `choice()` function creates a *choice* that describes the next alternatives for search. The choice must be independent of the brancher's home space, and must contain all necessary information for the alternatives.

- The `commit()` function takes a previously created choice and commits to one of the choice's alternatives. Note that `commit()` must use only the information in the choice, the domains of the brancher's views might be weaker, stronger, or incomparable to the domains when the choice was created (due to recomputation, see Section 40.2.1).

- The `print()` function takes a previously created choice and prints some information of one of the choice's alternatives.

- The `ngl()` function takes a previously created choice and returns a *no-good literal* which then can be used as part of a no-good (see also Section 9.5). The no-good literal is an implementation of a typically simple constraint that corresponds to an alternative as described by the choice. No-good literals are mostly orthogonal to the other functions and are hence discussed in the next chapter in section Section 32.2.

Branchers and propagators are both actors, they both inherit from the class `Actor`. That entails that copying and disposal of branchers is the same as it is for propagators and hence does not require any further discussion (see Section 22.3). Also memory management is identical, see Chapter 30.

How branchers execute is best understood when studying the operations that are performed by a search engine on a space.

Status computation. A search engine calls the `status()` function of a space to determine whether a space is failed, solved, or must be branched on. When the `status()` function of a space is executed, constraint propagation is performed as described in Section 22.1. If the space is failed, `status()` returns the value `SS_FAILED` (a value of type `SpaceStatus`, see Programming search engines).

Assume that constraint propagation has finished (hence, the space is stable) and the space is not failed. Then, `status()` computes the status of the current brancher. The status of a brancher is computed by the virtual `status()` member function a brancher implements. If the brancher's `status()` function returns **`false`**, the space's `status()` function tries to select the next brancher in the queue of branchers as the current brancher. If there are no branchers left in the queue of branchers, the space's `status()` function terminates and reports that the space is solved (by returning `SS_SOLVED`). Otherwise the process is repeated until the space has a current brancher such that its `status()` function has returned **`true`**. In this case, the space's `status()` function returns `SS_BRANCH` to signal to the search engine that the space requires branching.

The `status()` function of a brancher does not do anything to actually perform branching, it just checks whether there is something left to do for the brancher (for example, whether there are unassigned views left in an array of views to be branched on).

Choice creation. In case the space's `status()` function has returned `SS_BRANCH`, the search engine typically branches. In order to actually branch, the search engine must know how to branch. To keep search engines orthogonal to the space type they compute with

(that is, the problem a search engine tries to find solutions for), a search engine can request a *choice* as a description of how to branch. With a choice, a search engine can *commit* a space to a particular alternative as defined by the choice. Alternatives are numbered from zero to the number of alternatives minus one, where the number of alternatives is defined by the choice. A choice is specific to a brancher and must provide sufficient information such that the brancher together with the choice can commit to all possible alternatives.

A search engine requests the computation of a choice by executing the `choice()` function of a space. The space's `choice()` function does nothing but returning the choice created by the `choice()` function of the current brancher. Let us postpone the discussion of how a choice can store the information needed for branching. A choice always provides two important pieces of information:

- The number of its alternatives (an unsigned integer greater than zero) is available through the function `alternatives()`.
- A unique identity inherited from the brancher that has created the choice. This information is not available to a programmer but makes it possible for a space to find the corresponding brancher for a given choice (to be detailed below).

A search engine must execute the `choice()` function of a space *immediately* after executing the space's `status()` function. Any other action on the space (such as adding propagators or modifying views) might change the current brancher of the space and hence `choice()` might fail to compute the correct choice.

Committing to alternatives. Assume that the engine has a choice `ch` with two alternatives and that the engine has created a clone `c` of the current space `s` by[1]

```
Space* c = s->clone();
```

The search engine typically commits `s` to the first alternative (the alternative with number `0`) as defined by the space and later it might commit `c` to the second alternative (the alternative with number `1`). Let us first consider committing the space `s` to the first alternative by:

```
s->commit(*ch,0);
```

The space's `commit()` function attempts to find a brancher that corresponds to the choice `ch` (based on the identity that is stored by the choice). Assume, the space's `commit()` function finds a brancher `b`. Then it invokes the `commit()` function of the brancher `b` as follows:

```
b.commit(*s,*ch,0);
```

[1]Note that search engines compute with pointers to spaces and choices rather than references, hence `s`, `c`, and `ch` are pointers.

Now the brancher `b` executes its `commit()` function. The function typically modifies a view as defined by the choice (`*ch` in our example) and the number of the alternative (`0` in our example) passed as arguments. The brancher's `commit()` function must return an executions status that captures whether the operations performed by `commit()` have failed (`ES_FAILED` is returned) or not (`ES_OK` is returned).

If the space's `commit()` function does not find a brancher that corresponds to the choice `ch`, an exception of type `SpaceNoBrancher` is thrown. The reason for being unable to find a corresponding brancher is that the search engine is implemented incorrectly (see Part S).

At some later time, the search engine might decide to explore the second alternative of the choice `ch` by using the clone `c`:

```
c->commit(*ch,1);
```

Then, the `commit()` function of the space `c` tries to find the brancher corresponding to `ch`. This of course will be a different brancher in a different space: however, the brancher found will be a copy of the brancher `b`.

More on choices. A consequence of the discussion of `commit()` is that choices cannot store information that belongs to a particular space: the very idea of a choice is that it can be used with different spaces (above: original and clone)! That also entails that a choice is allocated independently from any space and must be deleted explicitly by a search engine.

Consider as an example a brancher that wants to create the choice $(x_i = n) \vee (x_i \neq n)$ where `x` is an array of integer views (that is, $x_i = n$ is alternative `0` and $x_i \neq n$ is alternative `1`). The choice is not allowed to store x_i directly (as x_i belongs to a space), instead it stores the position i in the array `x` and the integer value `n`. When the brancher's `commit()` function is executed, the brancher provides the view array `x` (the part that is specific to a space) and the choice provides the position i and the value `n` (the parts that are space-independent).

A choice must inherit from the class `Choice` and needs to implement a single virtual member function `size()` which returns the size (of type `size_t`) of the choice. This function can be used by a search engine to compute the amount of memory allocated by a choice (this can be useful for statistics).

Even more on (archives of) choices. Branchers and choices must support one more operation, the *(un-)archiving* of choices. Archiving enables choices to be used not only in the same search engine with a different space, but transferred to a search engine in a different process, possibly on a different computer. The main application for this are distributed search engines.

An `Archive` is simply an array of unsigned integers, which is easy to transmit e.g. over a network. Every choice must implement a virtual `archive()` member function, which in turn calls `archive()` on the super class and then writes the contents of the choice into the archive. Conversely, branchers must implement a virtual `choice()` member function that gets an archive as the argument and returns a new choice.

Brancher invariants and life cycle. The very idea of a choice is that it is a space-independent description of how to perform commits on a space. Consider the following example scenario: a search engine has a space `s` and a clone `c` of `s`. Then, the search engine explores part of the search tree starting from `s` and stores a sequence of choices while exploring. Then, the engine wants to recompute a node in the search tree and uses the clone `c` for it: it can recompute by performing commits with the choices it has stored.

In this scenario, the brancher to which the choices correspond *must* be able to perform the commits as described by the choices. That in particular entails that a brancher cannot modify its state arbitrarily: it must always guarantee that choices it created earlier (actually, that a copy of it created earlier) can be interpreted correctly.

The `status()` and `choice()` functions of a brancher can be seen as being orthogonal to its `commit()` function. The former functions compute new choices. The latter function must be capable to commit according to other choices that are unrelated to the choice that might be computed by the `choice()` function. Even if the `status()` function of a brancher has already returned **`false`**, the brancher must still be capable of performing commits. Hence, a brancher cannot be disposed immediately when its `status()` returns **`false`**.

Spaces impose an important invariant on the use of its `commit()` and `choice()` function. After having executed the `choice()` function, no calls to `commit()` are allowed with choices created earlier (that is, `choice()` invalidates all previously created choices for `commit()`). That also clarifies when branchers are disposed: the `choice()` function of a space disposes all branchers for which their `status()` functions have returned **`false`** before. This invariant is essential for recomputation, see Section 40.2.1.

Garbage collection of branchers. As the `choice()` function of a space performs garbage collection of branchers, it can also be called in case the space's `status()` function returned `SS_SOLVED` (signaling that no more branchers are left). In this case, the branchers are garbage collected but no choice is returned (instead, `NULL` is returned). See Section 39.2 for an example and Section 39.1 for a discussion.

31.2 Implementing a `nonemin` branching

This section presents an example for a branching and its implementing brancher.

The branching

```
void nonemin(Home home, const IntVarArgs& x);
```

branches by selecting the first unassigned variable x_i in `x` and tries to assign x_i to the smallest possible value of x_i.

That is, `nonemin` is equivalent to the predefined branching (see Section 8.2) used as

```
branch(home, x, INT_VAR_NONE(), INT_VAL_MIN());
```

For examples of problem-specific branchers see Chapter 18 and Chapter 19.

NONE MIN ☰ [DOWNLOAD]

```
...
class NoneMin : public Brancher {
protected:
  ViewArray<Int::IntView> x;
  ▶ CHOICE DEFINITION
public:
  NoneMin(Home home, ViewArray<Int::IntView>& x0)
    : Brancher(home), x(x0) {}
  static void post(Home home, ViewArray<Int::IntView>& x) {
    (void) new (home) NoneMin(home,x);
  }
  ...
  ▶ STATUS
  ▶ CHOICE
  ▶ COMMIT
  ▶ PRINT
};
void nonemin(Home home, const IntVarArgs& x) {
  if (home.failed()) return;
  ViewArray<Int::IntView> y(home,x);
  NoneMin::post(home,y);
}
```

Figure 31.1: A branching and brancher for `nonemin`

31.2.1 A naive brancher

Figure 31.1 shows the class `NoneMin` implementing the brancher for the branching `nonemin`. `NoneMin` inherits from the class `Brancher`. The branching post function creates a view array and posts a brancher of class `NoneMin`. The brancher post function does not return an execution status as posting a brancher does not fail (in contrast to a propagator post function, see Section 22.4). The constructor as well as the brancher post function of `NoneMin` are exactly as to be expected from our knowledge on implementing propagators (of course, branchers do not use subscriptions).

Status computation. The status computation of a `NoneMin` brancher is straightforward. The `status()` function scans all views in the view array `x` and returns **`true`** if there is an unassigned view left:

STATUS ≡
```
virtual bool status(const Space& home) const {
  for (int i=0; i<x.size(); i++)
    if (!x[i].assigned())
      return true;
  return false;
}
```

Choice computation. Computing the choice for a brancher involves two aspects: the definition of the class for the choice object and the `choice()` member function of a brancher that creates choice objects.

The choice object `PosVal` inherits from the class `Choice`. The information that is required for committing is which view and which value should be used. As discussed above, the `PosVal` choice does not store the view directly, but the position `pos` of the view in the view array `x`. A `PosVal` choice stores both position and value as integers as follows:

CHOICE DEFINITION ≡
```
class PosVal : public Choice {
public:
  int pos; int val;
  PosVal(const NoneMin& b, int p, int v)
    : Choice(b,2), pos(p), val(v) {}
  virtual size_t size(void) const {
    return sizeof(*this);
  }
  virtual void archive(Archive& e) const {
    Choice::archive(e);
    e << pos << val;
  }
};
```

The constructor of `PosVal` uses the constructor of `Choice` for initialization, where both the brancher `b` and the number of alternatives of the choice (`2` for `PosVal`) are passed as arguments. The `size()` function just returns the size of a `PosVal` object. The `archive()` function calls `Choice::archive()` and then writes the position and value to the archive.

The `choice(Space& home)` member function of the brancher is called directly (by the `choice(Space& home)` function of a space) after the `status()` function of the brancher in case the `status()` of the brancher has returned **`true`**. That is, the brancher is the current brancher of the space and still needs to create choices for branching. For a `NoneMin` brancher that means that there is definitely a not yet assigned view in the view array `x`, and that the brancher must choose the first unassigned view:

CHOICE ≡
```
virtual Choice* choice(Space& home) {
  for (int i=0; true; i++)
    if (!x[i].assigned())
      return new PosVal(*this,i,x[i].min());
  GECODE_NEVER;
  return NULL;
}
virtual Choice* choice(const Space&, Archive& e) {
  int pos, val;
  e >> pos >> val;
  return new PosVal(*this, pos, val);
}
```

The `choice(Space& home)` function returns a new `PosVal` object for position `i` and the smallest value of `x[i]`. The `choice(const Space&, Archive& e)` function creates a choice from the information contained in the archive.

Tip 31.1 (Never execute)**.** The macro `GECODE_NEVER` states that it will never be executed. If compiled in debug mode, `GECODE_NEVER` corresponds to the assertion `assert(false)`. If compiled in release mode, the macro informs the compiler (depending on the platform) that `GECODE_NEVER` is never executed so that the compiler can take advantage of this information for optimization. ◀

Committing to alternatives. The `commit()` function of `NoneMin` can safely assume that the choice `c` passed as argument is in fact an object `pv` of class `PosVal`. This is due to the fact that the space `commit()` function uses the identity stored in a choice object to find the corresponding brancher.

COMMIT ≡
```
virtual ExecStatus commit(Space& home,
                          const Choice& c,
                          unsigned int a) {
  const PosVal& pv = static_cast<const PosVal&>(c);
  int pos=pv.pos, val=pv.val;
  if (a == 0)
    return me_failed(x[pos].eq(home,val)) ? ES_FAILED : ES_OK;
  else
    return me_failed(x[pos].nq(home,val)) ? ES_FAILED : ES_OK;
}
```

From the `PosVal` object the position of the view `pos` and the value `val` are extracted. Then, if the commit is for the first alternative (`a` is `0`), the brancher assigns the view `x[pos]` to `val` using the view modification operation `eq` (see Figure 22.5). If the modification operation

returns a modification event signaling failure, `me_failed` is true and hence the execution status `ES_FAILED` is returned. Otherwise, `ES_OK` is returned as execution status. If the commit is for the second alternative (`a` is `1`), the value `val` is excluded from the view `x[pos]`.

Printing information on alternatives. The `print()` function of `NoneMin` is straightforward, it prints on the standard output stream `o` information about an alternative (it prints what `commit()` does):

PRINT ≡
```
virtual void print(const Space& home, const Choice& c,
                   unsigned int a,
                   std::ostream& o) const {
  const PosVal& pv = static_cast<const PosVal&>(c);
  int pos=pv.pos, val=pv.val;
  if (a == 0)
    o << "x[" << pos << "] = " << val;
  else
    o << "x[" << pos << "] != " << val;
}
```

The `print()` function is used for displaying information about alternatives explored during search. For example, it is used in Gist (see Section 10.3.4) or can be used in other search engines (see Tip 39.1).

31.2.2 Improving status and choice

The `status()` and `choice()` functions of `NoneMin` as defined in Figure 31.1 are inefficient as they always inspect the entire view array for finding the first unassigned view.

The brancher shown in Figure 31.2 stores an integer `start` to remember which of the views in `x` are already assigned. The brancher maintains the invariant that all x_i are assigned for $0 \leq i < \texttt{start}$. The value of `start` will be updated by the `status()` function which must be declared as **`const`**. As `start` is updated by a **`const`**-function, it is declared as **`mutable`**.

The `choice()` function of an improved `NoneMin` brancher can rely on the fact that `x[start]` is the first unassigned view in the view array `x`.

31.3 Implementing a `sizemin` branching

This section presents an example for a brancher that implements a more interesting selection of the view to branch on. The branching

```
void sizemin(Home home, const IntVarArgs& x);
```

NONE MIN IMPROVED ≡ [DOWNLOAD]

```
...
class NoneMin : public Brancher {
protected:
  ViewArray<Int::IntView> x;
  mutable int start;
  ...
  virtual bool status(const Space& home) const {
    for (int i=start; i<x.size(); i++)
      if (!x[i].assigned()) {
        start = i; return true;
      }
    return false;
  }
  virtual Choice* choice(Space& home) {
    return new PosVal(*this,start,x[start].min());
  }
  ...
};
...
```

Figure 31.2: An improved brancher for `nonemin`

SIZE MIN ☰ [DOWNLOAD]

```
...
class SizeMin : public Brancher {
  ...
  virtual Choice* choice(Space& home) {
    int p = start;
    unsigned int s = x[p].size();
    for (int i=start+1; i<x.size(); i++)
      if (!x[i].assigned() && (x[i].size() < s)) {
        p = i; s = x[p].size();
      }
    return new PosVal(*this,p,x[p].min());
  }
  ...
};
...
```

Figure 31.3: A brancher for `sizemin`

branches by selecting the variable in `x` with smallest domain size first and tries to assign the selected variable to its smallest possible value. That is, `sizemin` is equivalent to the predefined branching (see Section 8.2) if used as

```
branch(home, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

Figure 31.3 shows the brancher `SizeMin` implementing the `sizemin` branching. The `status()` function is not shown as it is exactly the same function as for `NoneMin` from the previous section: after all, the brancher is done only after all of its views are assigned. Likewise, the `PosVal` choice and the `commit()` and `print()` functions are unchanged.

The `choice()` function uses the integer `p` for the position of the unassigned view with the so-far smallest domain size and the unsigned integer `s` for the so-far smallest size. Then, all views that are not known to be assigned are scanned to find the view with smallest domain size. Keep in mind that due to the invariant enforced by the brancher's `status()` function, `x[start]` is not assigned.

It is absolutely essential to skip already assigned views. If assigned views were not skipped, then the same choice would be created over and over again leading to an infinite search tree.

32 Advanced topics

This chapters presents advanced topics for programming branchers as implementations of branchings.

Overview. Section 32.1 presents a specialized brancher for assigning views rather than branching on them. How branchers support no-goods is discussed in Section 32.2. Variable views for branchers are discussed in Section 32.3.

32.1 Assignment branchers

This section presents an example for a brancher that assigns all of its views rather than branches on its views. The branching

```
void assignmin(Home home, const IntVarArgs& x);
```

assigns all variables in `x` to their smallest possible value. That is, `assignmin` is equivalent to the predefined branching (see Section 8.13) used as

```
assign(home, x, INT_ASSIGN_MIN());
```

Figure 32.1 shows the relevant parts of the brancher `AssignMin`. Unsurprisingly, both the `status()` and `choice()` function are identical to those shown in Figure 31.2 (and hence are omitted).

The changes concern the created choice of type `PosVal`: the constructor now initializes a choice with a single alternative only (the second argument to the call of the constructor `Choice`). The `commit()` function is a specialized version of the `commit()` function defined in Figure 31.2. It only needs to be capable of handling a single alternative. The same holds true for the `print()` function.

32.2 Supporting no-goods

Supporting no-goods by a brancher is straightforward: every brancher has a virtual member function `ngl()` (for no-good literal) that takes the same arguments as the `commit()` member function: a space, a choice, and the number of the alternative and returns a pointer to a no-good literal of class `NGL`. The `ngl()` function is called during no-good generation

ASSIGN MIN ≡ [DOWNLOAD]

```
...
class AssignMin : public Brancher {
  ...
  class PosVal : public Choice {
  public:
    int pos; int val;
    PosVal(const AssignMin& b, int p, int v)
      : Choice(b,1), pos(p), val(v) {}
    ...
  };
  ...
  virtual ExecStatus commit(Space& home,
                            const Choice& c,
                            unsigned int a) {
    const PosVal& pv = static_cast<const PosVal&>(c);
    int pos=pv.pos, val=pv.val;
    return me_failed(x[pos].eq(home,val)) ? ES_FAILED : ES_OK;
  }
  ...
};
...
```

Figure 32.1: A brancher for `assignmin`

(see Section 9.5) and the returned no-good literal is then used by a no-good propagator that propagates the no-goods (if you are curious, the propagator is implemented by `Search::Meta::NoGoodsProp`).

By default, the `ngl()` function of a brancher returns `NULL`, which means that the brancher does not support no-goods. In order to support no-goods, a brancher must redefine the `ngl()` function and must define a class (or several classes) for the no-good literals to be returned. The class `EqNGL` implementing a no-good literal for equality and the `ngl()` function is shown in Figure 32.2. Otherwise, the brancher is the same as the `nonemin` brancher shown in Figure 31.2.

32.2.1 Returning no-good literals

The `ngl()` function of a brancher has the following options:

- As mentioned above, it can always return `NULL` and hence the brancher does not support no-goods.

- It returns for each alternative of a choice a no-good literal. For our `NoneMin` brancher this would entail that when `ngl(home,c,0)` is called, it returns a no-good literal implementing equality between a view and an integer that corresponds to the first alternative (for a given space `home` and a choice `c`).

  For the second alternative `ngl(home,c,1)` a no-good literal implementing disequality should be returned. This would work, but the brancher can do better than that as is explained below.

- When `ngl(home,c,a)` is called for an alternative where $a > 0$ with a space `home` and a choice `c` and `a` is the last alternative (for `NoneMin`, $a = 1$) and the last alternative is the logical negation of all other alternatives, then the `ngl()` function can return `NULL` as an optimization.

  Assume that the alternatives of a choice are

  $$l_0 \vee \ldots \vee l_{n-1}$$

  where `n` is the arity of the choice and it holds that

  $$(l_0 \vee \ldots \vee l_{n-2}) \Leftrightarrow \neg l_{n-1}$$

  is true, then the `ngl()` function can return `NULL` for the last alternative (that is, for the alternative $n-1$).

  Note that this optimization implements the very same idea as discussed at the beginning of Section 9.5. Note also that this property is typically only true for branchers with binary choices such as in our example.

NONE MIN WITH NO-GOOD SUPPORT ≡ [DOWNLOAD]

```
...
class EqNGL : public NGL {
protected:
  Int::IntView x; int n;
public:
  EqNGL(Space& home, Int::IntView x0, int n0)
    : NGL(home), x(x0), n(n0) {}
  EqNGL(Space& home, bool share, EqNGL& ngl)
    : NGL(home, share, ngl), n(ngl.n) {
    x.update(home, share, ngl.x);
  }
  ▶ STATUS
  ▶ PRUNE
  ▶ SUBSCRIBE AND CANCEL
  virtual NGL* copy(Space& home, bool share) {
    return new (home) EqNGL(home, share, *this);
  }
  virtual size_t dispose(Space& home) {
    (void) NGL::dispose(home);
    return sizeof(*this);
  }
};

class NoneMin : public Brancher {
  ...
public:
  ...
  ▶ NO-GOOD LITERAL CREATION
};
...
```

Figure 32.2: Branching for `nonemin` with no-good support

In our example, the second alternative is indeed the negation of the first alternative. Hence the following `ngl()` function implements no-good literal creation and only requires a single class `EqNGL` for no-good literals implementing equality:

NO-GOOD LITERAL CREATION ≡
```
virtual NGL* ngl(Space& home, const Choice& c,
                 unsigned int a) const {
  const PosVal& pv = static_cast<const PosVal&>(c);
  int pos=pv.pos, val=pv.val;
  if (a == 0)
    return new (home) EqNGL(home, x[pos], val);
  else
    return NULL;
}
```

32.2.2 Implementing no-good literals

No-good literals implement constraints that correspond to alternatives of choices. But instead of implementing these constraints by a propagator, they have a specialized implementation that is used by a no-good propagator. All concepts needed to implement a no-good literal are concepts that are familiar from implementing a propagator.

A no-good literal inherits from the class `NGL` and must implement the following constructors and functions:

- Unsurprisingly, a no-good literal must implement constructors for creation and cloning and member functions `copy()` for copying and `dispose()` for disposal. They are straightforward and are shown in Figure 32.2.

- It must implement a `status()` function that checks whether the no-good literal is subsumed (the function returns `NGL::SUBSUMED`), failed (the function returns `NGL::FAILED`), or neither (the function returns `NGL::NONE`). The return type is `NGL::Status` as defined in the `NGL` class.

  It is important to understand that the no-good propagator using no-good literals can only perform propagation if some of its no-good literals become subsumed. Hence, the test for subsumption used in `status()` should try to detect subsumption as early as possible.

  Testing subsumption for our `EqNGL` no-good literal is straightforward:

  STATUS ≡
  ```
  virtual NGL::Status status(const Space& home) const {
    if (x.assigned())
      return (x.val() == n) ? NGL::SUBSUMED : NGL::FAILED;
    else
      return x.in(n) ? NGL::NONE : NGL::FAILED;
  }
  ```

- The `prune()` function propagates the negation of the constraint that the no-good literal implements.

  Again, the `prune()` function for `NoneMin` is straightforward:

```
PRUNE ≡
  virtual ExecStatus prune(Space& home) {
    return me_failed(x.nq(home,n)) ? ES_FAILED : ES_OK;
  }
```

- A no-good literal must implement a `subscribe()` and a `cancel()` function that subscribe and cancel the no-good propagator to the no-good literal's views.

  As mentioned above, the earlier the `status()` function detects subsumption, the more constraint propagation can be expected from the no-good propagator. Hence, it is important to choose the propagation condition for subscriptions such that whenever there is a modification to the view that could result in subsumption the no-good propagator is executed.

  For the `EqNGL` no-good literal, we choose the propagation condition for subscriptions to be `Int::PC_INT_VAL` (see Section 22.6 for a discussion of propagation conditions). This choice reflects the fact that subsumption can only be decided after the view `x` has been assigned:

```
SUBSCRIBE AND CANCEL ≡
  virtual void subscribe(Space& home, Propagator& p) {
    x.subscribe(home, p, Int::PC_INT_VAL);
  }
  virtual void cancel(Space& home, Propagator& p) {
    x.cancel(home, p, Int::PC_INT_VAL);
  }
```

In case a no-good literal uses members that must be deallocated when the home-space is deleted, the no-good literal's class must redefine the virtual member functions `notice()` and `dispose()`, for example by:

```
virtual bool notice(void) const {
  return true;
}
virtual size_t dispose(Space& home) {
  ...
}
```

If `notice()` returns **`true`**, the no-good propagator ensures that the no-good literal's `dispose()` function is called whenever the propagator's home-space is deleted.

NONE MIN AND NONE MAX ≡ [DOWNLOAD]

```
...
template<class View>
class NoneMin : public Brancher {
  ...
};
void nonemin(Home home, const IntVarArgs& x) {
  ...
}
void nonemax(Home home, const IntVarArgs& x) {
  if (home.failed()) return;
  ViewArray<Int::MinusView> y(home,x.size());
  for (int i=x.size(); i--; )
    y[i]=Int::MinusView(x[i]);
  NoneMin<Int::MinusView>::post(home,y);
}
```

Figure 32.3: Branchings for `nonemin` and `nonemax`

32.3 Using variable views

Variable views can also be used for reusing branchers to obtain several branchings, similar to reusing propagators for several constraints, see Chapter 27.

While in principle all different variable views introduced in Chapter 27 can be used for branchers, the only meaningful variable view for branchers is the minus integer view (see Section 27.1.1).

As an example consider the branchings `nonemin` (see Section 31.2) and `nonemax` where the latter tries to assign the maximal value of a view first. The corresponding program fragment is shown in Figure 32.3. The class `NoneMin` is made generic with respect to the type of view it uses. Then, both `nonemin` and `nonemax` can be obtained by instantiating `NoneMin` with integer views or integer minus views.

Programming variables

Christian Schulte

This part explains how new variable types can be programmed with Gecode.

In order to be able to program variables, all chapters in this part should be read. The chapters capture the following:

- Chapter 33 (Getting started) outlines how a new variable type can be programmed for Gecode.
- Chapter 34 (Variable implementations) shows how the kernel-specific and variable domain-specific aspects of a variable implementation are specified and programmed.
- Chapter 35 (Variables and variable arrays) shows how variables and variable arrays for modeling are programmed for a given variable implementation.
- Chapter 36 (Views) shows how views for programming propagators and branchers are programmed for a given variable implementation.
- Chapter 37 (Variable-value branchings) explains how variable-value branchings can be implemented from functionality provided by Gecode.
- Chapter 38 (Putting everything together) finally explains how a new variable type can be used with Gecode.

Important. Programming variables requires to configure and recompile Gecode from its source code. Using one of the pre-compiled packages is not sufficient. More details can be found in Chapter 38.

33 Getting started

This chapter outlines how a new variable type can be programmed with Gecode. The chapter (and the entire part on programming variables) chooses integer interval variables as its running example.

Overview. An overview of what needs to be designed and programmed is presented in Section 33.1. The structure of how the implementation of a variable type is organized is presented in Section 33.2.

Important. Programming variables requires to configure and recompile Gecode from its source code. More details can be found in Chapter 38.

33.1 Overview

We are going to use *integer interval variables* as the running example for programming variables. Integer interval variables take integer values (like the integer variables that come pre-defined with Gecode do) but their domain is defined by a lower and an upper bound only. That is, the domain is always an interval. This is in contrast to the integer variables that come with Gecode, where their domain can be any finite set of integer values.

The focus of this part is on understanding what needs to be done for implementing new variables, so we deliberately choose very simple variables together with very few operations on them as an example.

Even though integer interval variables seem not very interesting, variants of them could in fact be interesting. For example, the integer values used for the lower and upper bound could be integers of arbitrary precision, or instead of integer values one could choose floating point values.

What must be programmed. Programming variables includes the following tasks:

- *Variable implementations* (Chapter 34): Programming a variable implementation consists of two tasks:

  1. The first task is to specify domain-independent aspects of a variable implementation. This includes specifying a name for the variable implementation type, scope

information, modification events, and propagation conditions. From a simple specification file containing this information a domain-independent base class for a variable implementation and the corresponding C++ definitions of modification events and propagation conditions is generated.

The generated base class together with the definition of modification events and propagation conditions actually become part of Gecode's kernel. The kernel needs these definitions to schedule propagators that have subscribed to a variable implementation and to maintain these subscriptions during cloning.

2. The second task consists of programming the domain-dependent operations of a variable implementation. This is achieved by defining a class for a variable implementation that inherits from the generated, domain-independent base class and defines the respective domain operations.

- *Variables* and *variable arrays* (Chapter 35): As a variable is nothing but a simple and read-only interface to a variable implementation, a variable is obtained by inheriting from a base class for variables that depends on the variable implementation type. The actual programming amounts to defining read-only variable operations that invoke the corresponding operations on the variable's variable implementation.

  Variable arrays and variable argument arrays are, as variables, needed for modeling. Their programming requires the definition of several traits classes so that the defined arrays can be used with Gecode-provided functionality (for example, with the matrix interface for arrays, see Section 7.2).

- *Views* (Chapter 36): Programming a view depends on the type of the view: whether the view is a direct interface to a variable implementation (a *variable implementation view*), whether it is a *constant view*, or whether it is derived (a *derived view*) from some other view. As examples, we are going to implement an integer view as a variable implementation view, minus and offset views as derived views, and an integer constant view as a constant view.

  In addition to the classes for views, some additional functions on views must be defined for testing in which order views are and whether two views are shared or the same (see Section 27.1.2).

- *Constraints* and *branchings*: typically, when implementing variables one also needs to implement constraints and branchings for them. The implementation of constraints for a new variable type is not in any way different from what is described in Part P.

  The situation for implementing branchings is quite different: here one would want to offer at least a common set of variable-value branchings similar to those for integer variables (see Section 8.2). Gecode offers substantial support for implementing variable-value branchings, including support for the specification of variable and value selection strategies, random and activity-based selection of variables, tie-breaking, filter functions, no-goods, and much more. How to use Gecode's support for implementing variable-value branchings is detailed in Chapter 37.

Putting everything together. Even though we are presenting the implementation of integer interval variables only as an example, Chapter 38 shows how everything is put together. This includes examples of propagators, post functions using various views, and a simple script (Golomb rulers, see Chapter 12) using integer interval variables.

More importantly, it also shows how Gecode must be configured and compiled such that integer interval variables are supported by Gecode's kernel.

33.2 Structure

The implementation of integer interval variables is contained in a single header file `int.hh`, which is shown in Figure 33.1.

Namespaces. The implementation is contained in the namespace `MPG` (for `M`odeling and `P`rogramming with `G`ecode) to avoid name-clashes with functionality provided by Gecode. To keep the implementation of integer interval variables concise, some important definitions in the `Gecode` namespace are made available by **`using`** declarations (see Figure 33.1).

Similar to the organization of namespaces in Gecode, definitions that are used for modeling (variables and variable arrays) are contained in the namespace `MPG`, while definitions that are used for programming (variable implementations, views, branchers, and additional support) are in the namespace `MPG::Int`. As the structure of namespaces matters (part of the support for variable arrays must be defined inside the `Gecode` namespace), each program fragment is shown in its appropriate namespace.

As an example, consider the definition of exceptions. Two are thrown by the constructor of the integer interval variable, in case the variable domain is ill-specified. The third exception is thrown when the variable or value selection for a branching is unknown:

EXCEPTIONS ≡

```
namespace MPG { namespace Int {
  class OutOfLimits : public Exception {
  public:
    OutOfLimits(const char* l)
      : Exception(l,"Number out of limits") {}
  };
  class VariableEmptyDomain : public Exception {
    ...
  };
  class UnknownBranching : public Exception {
    ...
  };
}}
```

As discussed above, `Exception` is `Gecode::Exception` (see `Exception`) and has been introduced by a **`using`** declaration.

INT.HH ≡ [DOWNLOAD]

```
#ifndef __MPG_INT_HH__
#define __MPG_INT_HH__
...
#include <gecode/kernel.hh>

using Gecode::Advisor;
...

► EXCEPTIONS

► VARIABLE IMPLEMENTATION

► VARIABLE
► ARRAY TRAITS
► VARIABLE ARRAYS

► INTEGER VIEW
► CONSTANT INTEGER VIEW
► MINUS VIEW
► OFFSET VIEW

► BRANCHING

#endif
```

Figure 33.1: The header file for integer interval variables

Naming scheme. The naming scheme follows the same naming scheme for integer variables as defined by Gecode (albeit defined in the namespace `MPG` instead of `Gecode`):

- Variable implementations: The base class is named `IntVarImpBase` whereas the variable implementation class is named `IntVarImp`. The names of modification events start with `ME_INT_` whereas the names of propagation conditions start with `PC_INT_`. As mentioned above, these classes and identifiers are defined within the namespace `MPG::Int`.
- Variables and variable arrays: Integer interval variables are implemented by the class `IntVar`. Variable arrays of integer interval variables are implemented by the class `IntVarArray`, whereas the corresponding variable argument array is implemented by the class `IntVarArgs`. These classes are defined in the namespace `MPG`.
- Views: the respective views are implemented by classes `IntView`, `ConstIntView`, `MinusView`, and `OffsetView`. They are all defined within the namespace `MPG::Int`.
- Branchings: how variables and values are selected is implemented by functions such as `INT_VAR_NONE()` or `INT_VAL_MIN()` and the actual branching is implemented by a single `branch()` function. The good news is that no actual brancher must in fact be implemented, even though a number of rather straightforward support definitions must be implemented (which are contained in the namespace `MPG::Int`).

Inline functions as simplification. All functions, be they member or non-member functions are defined as **`inline`**. The reason for this is to make it easier to follow the example, as only the single header file `int.hh` is needed. In a real implementation one would move the definitions of some functions to a source file and only leave the declaration of the functions in the header file. This is in particular true for many of the functions defined in Chapter 37.

34 Variable implementations

This chapter describes how variable implementations can be programmed with Gecode. The chapter uses integer interval variables as introduced in Chapter 33 as its running example.

Overview. The design of integer interval variables is detailed in Section 34.1. After having finalized the design, Section 34.2 explains how the domain-independent base class for the variable implementation together with definitions of modification events and propagation conditions can be generated from a simple specification. Section 34.3 shows how the actual variable implementation is programmed from the generated base class for a variable implementation. Section 34.4 provides an overview of additional options for generating a variable implementation base class from its specification.

34.1 Design decisions

Before starting with the description of the implementation of integer interval variables, let us detail their design. This includes the design of the variable domain including access and modification operations, deltas for advisors (see Chapter 26), modification events, and propagation conditions.

Variable domain and operations. Unsurprisingly, the variable domain of an integer variable implementation is represented by two integers `l` (lower bound) and `u` (upper bound). The variable implementation provides access operations `min()` and `max()` that return these integers.

To modify an integer interval variable implementation, the operation `gq(home,n)` modifies the domain such that its values must be greater or equal to `n`, whereas the operation `lq(home,n)` modifies the domain such that its values must be less or equal to `n`.

The values `l` and `u` can only be initialized (when creating a new variable, see Section 35.1) and modified such that they obey the following invariants:

1. The domain is never empty, that is, $\texttt{l} \leq \texttt{u}$.

2. The domain values never exceed the limits defined by (`INT_MAX` is the largest possible value for an **`int`**):

```
LIMITS ≡
  namespace Limits {
    const int max = (INT_MAX / 2) - 1;
    const int min = -max;
  }
```

That is, `Int::Limits::min` $\leq$ `l` $\leq$ `u` $\leq$ `Int::Limits::max`.

The choice of values for `Limits::min` and `Limits::max` are motivated by simplicity only. To keep the example propagators used in Chapter 38 simple, the limits are chosen such that the addition and subtraction of two integer values within the limits do not lead to numerical overflow. A real-life variable implementation would try to make as many values as possible available for a variable domain, see for example Section 4.1.2.

Tip 34.1 (Correctness matters)**.** While the decision to restrict the possible values of a variable implementation is motivated by simplicity, the decision for a real-life variable implementation is absolutely essential.

Being unclear about which values can correctly be maintained by a variable implementation, not ensuring that no numerical overflow occurs, or not checking for the necessary invariants when a new variable is created, renders the very idea of constraint programming obsolete: that whenever a solution is found by Gecode, it *actually happens to be a solution*. Hence correctness does not only matter for implementing propagators and branchers but also for getting the basic design of variables right. ◀

Assigned variables. An integer interval variable is assigned iff `l` $=$ `u`.

Deltas for advisors. We design the delta information for an advisor computed by a modification operation on the variable implementation to be an interval as well. The interval defines the values that are removed by a modification operation. Due to the nature of the modification operations `lq()` and `gq()`, the removed values always form an interval.

The design of deltas to be used by advisors for a variable implementation depends directly on the design of the modification operations provided by a variable implementation. For example, if our integer interval variable implementation also featured an operation `eq()` to assign a variable implementation to a value, then one also would have to choose a different design for the delta information. Assume a variable implementation with domain [`l` .. `u`] and that the modification operation `eq(home,n)` is executed where `l` $<$ `n` $<$ `u`. Then one could design the delta information to either accurately represent the set of removed values [`l` .. `n` $-$ `1`] $\cup$ [`n` $+$ `1` .. `u`] or to provide support for signaling that the domain has changed arbitrarily (this is the design chosen for integer variables in Gecode, see Section 26.3).

Modification events. Any variable implementation must support the mandatory events for no modification (to be implemented as `ME_INT_NONE`), for failure (to be implemented as `ME_INT_FAILED`), and for assignment to a value (to be implemented as `ME_INT_VAL`).

The additional events must be chosen such that they take the following two aspects into account:

- The modification operations should return meaningful values that describe how the domain of a variable implementation has changed. They must return `ME_INT_VAL` if the variable implementation becomes assigned. Otherwise, we choose to return `ME_INT_MIN` if the lower bound changes and to return `ME_INT_MAX` if the upper bound changes.
- When a new propagator is posted and the propagator subscribes to some views (and hence to some variable implementations), the propagator must be scheduled with respect to some modification event. This modification event should capture that "somehow the variable has changed for the propagator". In case an integer interval variable implementation is not yet assigned (otherwise the propagator will be scheduled with the modification event `ME_INT_VAL` anyway), we use an additional modification event `ME_INT_BND` capturing that one or both of the bounds have changed.

Again, there is quite some degree of freedom in the choice of modification events. Another design would be to only provide the modification event `ME_INT_BND` instead (apart from the mandatory modification events). An important aspect in which design to choose is the relation between modification events and propagation conditions to be discussed below.

Propagation conditions. To make our example variable implementations sufficiently interesting, we design the propagation conditions such that they can take full advantage of the modification events.

That is, apart from the mandatory propagation condition for not creating any subscription (to be implemented as `PC_INT_NONE`) and the mandatory propagation condition for an assigned variable implementation (to be implemented as `PC_INT_VAL`), we have three propagation conditions as follows:

- `PC_INT_MIN`: schedule a propagator if the lower bound of a variable implementation changes.
- `PC_INT_MAX`: schedule a propagator if the upper bound changes.
- `PC_INT_BND`: schedule a propagator if lower or upper bound changes.

This design can also be reformulated in terms of modification events that are generated by a modification operation:

- `PC_INT_MIN`: schedule a propagator for `ME_INT_VAL`, `ME_INT_MIN`, and `ME_INT_BND`.
- `PC_INT_MAX`: schedule a propagator for `ME_INT_VAL`, `ME_INT_MAX`, and `ME_INT_BND`.
- `PC_INT_BND`: schedule a propagator for `ME_INT_VAL`, `ME_INT_MIN`, `ME_INT_MAX`, and `ME_INT_BND`.

A simpler design would be to have the single non-mandatory propagation condition `PC_INT_BND`. The decision which design is best is not straightforward, as the tradeoff between the cost for additional propagation conditions (see below) and the gain from avoiding propagator executions depends on many different aspects. For a discussion and an evaluation in the context of Gecode's integer variables, see [48, Section 5].

Costs and limits for modification events and propagation conditions. The cost per each individual modification event and propagation condition is as follows:

- Assume that a variable implementation uses n different modification events (including the mandatory ones). The size of n does not affect efficiency. To represent these modification events, the Gecode kernel reserves $\lceil \log_2(n-1) \rceil$ bits in each propagator for maintaining modification event deltas, see Section 25.4.

  The totally available number of bits for all variable implementation types used by Gecode is 32 (independent of whether Gecode is run on a 32 bit or 64 bit platform). That is, if we assume less than ten modification events per variable implementation type, the Gecode kernel can support at least ten different variable implementation types.[1]

- The number of different propagation conditions m per variable implementation type is only limited by the largest value of an unsigned integer in C++.

  For each propagation condition, every variable implementation needs a 32-bit word, that is a variable implementation requires at least $O(m)$ space (which is typically dwarfed by the space consumed for actually storing the subscriptions of a propagator or an advisor to a variable implementation).

  Subscribing to a variable implementation requires $O(m)$ time. Canceling a subscription with propagation condition p requires $O(m+k)$ time, where k is the number of subscriptions with propagation condition p.

34.2 Base definitions

The variable implementation base class together with definitions of modification events and propagation conditions are not programmed but are generated from a simple specification file. The specification contains three sections: a general section for naming, a section for modification events, and a section for propagation conditions.

In the following we describe how to turn the parts of the design from the previous section that is concerned with modification events and propagation conditions into the specification. The member functions of the generated base class are used and explained in the next section.

[1] If you ever exceed this limit, please let us know. Adding more bits is easy, even though we do not expect that to happen anytime soon.

VARIABLE IMPLEMENTATION SPECIFICATION ≡ [DOWNLOAD]

```
[General]
Name:        Int
Namespace:   MPG::Int
► MODIFICATION EVENTS
► PROPAGATION CONDITIONS
[End]
```

Figure 34.1: Variable implementation specification

General section. The specification file (named `int.vis`, where `vis` stands for variable implementation specification; however the file extension does not matter) is shown in Figure 34.1. The specification file must start with **[General]** defining the start of the general section and must end with a line **[End]**. The `Name` option defines the names of the entities to be generated. In our example, a variable implementation base class `IntVarImpBase` (that is, the specified name is prepended to `VarImpBase`) generated, the identifiers for modification events start with `ME_INT_` (that is, the specified name is put after the `ME_` in capital letters), and the identifiers for propagation conditions start with `PC_INT_`. All these definitions are contained within the namespace as defined by the `Namespace` option.

The general section (and also the other sections discussed below) supports additional specification options, see Section 34.4 for a summary and a specification file template for download.

Modification event section. Every modification event requires a definition that is preceded by a line containing **[ModEvent]** as shown in Figure 34.2. The option `Name` defines the name of the modification event (in fact, just the part after `ME_INT_` for our example). The values on the right-hand side of `=` specify that some modification events are special:

- The modification events named `FAILED` (that is, `ME_INT_FAILED`) and `NONE` (that is, `ME_INT_NONE`) are defined to be the events for failure (`=FAILED`) and no change (`=NONE`).
- The modification event named `VAL` is defined to be the event when a variable implementation becomes assigned (`=ASSIGNED`) to a value.
- The modification event named `BND` is defined to be used for scheduling a propagator when the propagator subscribes to a non-assigned variable implementation (`=SUBSCRIBE`).

Any variable implementation must define special events with `=NONE`, `=FAILED`, and `=ASSIGNED`.

The section for modification events also defines how modification events are combined with a `Combine` option. The combination of modification events is needed for the correctness

```
MODIFICATION EVENTS ≡
  [ModEvent]
  Name:         FAILED=FAILED
  [ModEvent]
  Name:         NONE=NONE
  [ModEvent]
  Name:         VAL=ASSIGNED
  Combine:      VAL=VAL, MIN=VAL, MAX=VAL, BND=VAL
  [ModEvent]
  Name:         BND=SUBSCRIBE
  Combine:      VAL=VAL, MIN=BND, MAX=BND, BND=BND
  [ModEvent]
  Name:         MIN
  Combine:      VAL=VAL, MIN=MIN, MAX=BND, BND=BND
  [ModEvent]
  Name:         MAX
  Combine:      VAL=VAL, MIN=BND, MAX=MAX, BND=BND
```

Figure 34.2: Modification event section

of scheduling propagators and also for modification event deltas, see Section 25.4. An entry $l = r$ for the modification event m defines that m combined with l is r.

The definition of the combination of modification events can be expressed as a table:

	VAL	MIN	MAX	BND
VAL	VAL	VAL	VAL	VAL
MIN	VAL	MIN	BND	BND
MAX	VAL	BND	MAX	BND
BND	VAL	BND	BND	BND

This table is exactly what is specified by the `Combine` options. The special modification events `NONE` and `FAILED` do not have a `Combine` option.

We will not present the full mathematical detail of the properties that must hold for the combination of modification events, the theory is presented in [56, Section 5.5].

Propagation condition section. Every propagation condition requires a definition that is preceded by a line containing **`[PropCond]`** as shown in Figure 34.3. The option `Name` defines the name of the propagation condition (in fact, just the part after `PC_INT_` for our example). The values on the right-hand side of `=` specify that some propagation conditions are special:

- The propagation condition named `NONE` (that is, `PC_INT_NONE`) is defined to be the propagation condition for not creating any subscription (`=NONE`).

```
PROPAGATION CONDITIONS ≡
  [PropCond]
  Name:        NONE=NONE
  [PropCond]
  Name:        VAL=ASSIGNED
  ScheduledBy: VAL
  [PropCond]
  Name:        BND
  ScheduledBy: VAL, BND, MIN, MAX
  [PropCond]
  Name:        MIN
  ScheduledBy: VAL, BND, MIN
  [PropCond]
  Name:        MAX
  ScheduledBy: VAL, BND, MAX
```

Figure 34.3: Propagation condition section

- The propagation condition named `VAL` (that is, `PC_INT_VAL`) is defined to be the condition when a propagator wants to subscribe to the event that a variable implementation becomes assigned (`=ASSIGNED`).

Any variable implementation must define the special propagation conditions `=NONE` and `=ASSIGNED`.

For each propagation condition (but for `=NONE`), it must be defined by a `ScheduledBy` option which modification events schedule a propagator for execution. That is, when defining a propagation condition p, an entry m in the list of modification events defines the following: a propagator subscribed to a variable implementation x with propagation condition p is scheduled for execution when a modification operation on x returns the modification event m. The modification events in our example correspond to the design presented in Section 34.1.

34.3 Variable implementation

The variable implementation for integer interval variables is shown in Figure 34.4. As discussed in the previous sections, the variable implementation inherits from the generated base class `IntVarImpBase` and implements a lower bound `l` and an upper bound `u`.

Access operations. Every variable implementation must implement a member function `assigned()` that tests whether the variable is assigned to a value:

VARIABLE IMPLEMENTATION ≡

```
namespace MPG { namespace Int {

  ▶ LIMITS
  ▶ DELTA FOR ADVISORS

  class IntVarImp : public IntVarImpBase {
  protected:
    int l, u;
  public:
    IntVarImp(Space& home, int min, int max)
      : IntVarImpBase(home), l(min), u(max) {}
    ▶ ACCESS OPERATIONS
    ▶ ASSIGNMENT TEST
    ▶ MODIFICATION OPERATIONS
    ▶ SUBSCRIPTIONS
    ▶ COPYING
    ▶ DELTA INFORMATION
  };

}}
```

Figure 34.4: Variable implementation

ASSIGNMENT TEST ≡
```
bool assigned(void) const {
  return l == u;
}
```

The test for assignment is used in the implementation of other member functions of the variable implementation. Furthermore, variables and views automatically provide implementations of a member function `assigned()` that calls the `assigned()` function of their variable implementation.

The access operations for the lower and upper bound are straightforward. Here, and in the following, we only show one of the operations, the operation for the other bound is analogous:

ACCESS OPERATIONS ≡
```
int min(void) const {
  return l;
}
...
```

Modification operations. The modification operations must notify the Gecode kernel if a variable implementation is modified. As a description how a variable implementation changes, they must pass a modification event and delta information for advisors to a member function `notify()`. The `notify()` function executes subscribed advisors and schedules subscribed propagators (depending on the passed modification event and the propagators' propagation conditions). The `notify()` function is inherited from the generated variable implementation base class and depends on the specified modification events and propagation conditions.

The delta information is implemented as discussed in Section 34.1 as an interval with lower and upper bound:

DELTA FOR ADVISORS ≡
```
class IntDelta : public Delta {
private:
  int l, u;
public:
  IntDelta(int min, int max) : l(min), u(max) {}
  int min(void) const {
    return l;
  }
  ...
};
```

The actual modification operations first test whether the variable implementation does not require modification or whether the operation fails and only then perform the actual

modification. Before updating the upper bound `u` to `n`, the `lq()` operation creates the delta information `d` that describes that values between `n+1` and `u` are being removed.

The `notify()` function is given the `home` space, a modification event, and the variable delta `d` as argument. The modification event passed to `notify()` must capture how the domain has changed. In particular, it must reflect whether the variable implementation has been assigned. The `notify()` function executes the advisors subscribed to this variable implementation and schedules all subscribed propagators with appropriate propagation conditions. Note that the `notify()` function returns a modification event. In case an advisor reports failure after its execution, `notify()` returns `ME_INT_FAILED`. Otherwise it returns the modification event that has been passed as argument:

MODIFICATION OPERATIONS ≡
```
ModEvent lq(Space& home, int n) {
  if (n >= u) return ME_INT_NONE;
  if (n < l) return ME_INT_FAILED;
  IntDelta d(n+1,u); u = n;
  return notify(home, assigned() ? ME_INT_VAL : ME_INT_MAX, d);
}
...
```

Tip 34.2 (Variable implementations must always be consistent)**.** Even if a modification operation fails, the data structures for the variable implementation must be still consistent. That is, all operations must still work. See also Section 4.1.3. ◀

Delta information access. The variable implementation must also implement functions that provide access to the delta information:

DELTA INFORMATION ≡
```
static int min(const Delta& d) {
  return static_cast<const IntDelta&>(d).min();
}
...
```

This construction appears nonsensical at first sight, however there are two good reasons why a variable implementation interprets the information stored in the delta information (of course, in that case one would have to declare the operation as **`const`** but not **`static`**). First, the variable implementation can change the information based on its own state. Second, the very same idea is needed for views (see Section 36.4.2 for an example) and hence this design keeps the interfaces of views and variable implementations as similar as possible.

Subscriptions. A variable implementation must implement `subscribe()` and `cancel()` operations for both propagators and advisors. The implementation of these operations always follow the same structure as shown below.

The reason why these functions have to be implemented in the variable implementation class even though they are (in slightly different form) already defined in the variable

implementation base class is that they require information about whether a variable implementation is assigned. The definitions are as follows:

SUBSCRIPTIONS ≡
```
void subscribe(Space& home, Propagator& p, PropCond pc,
               bool schedule=true) {
  IntVarImpBase::subscribe(home,p,pc,assigned(),schedule);
}
void subscribe(Space& home, Advisor& a) {
  IntVarImpBase::subscribe(home,a,assigned());
}
void cancel(Space& home, Propagator& p, PropCond pc) {
  IntVarImpBase::cancel(home,p,pc,assigned());
}
void cancel(Space& home, Advisor& a) {
  IntVarImpBase::cancel(home,a,assigned());
}
```

Copying during cloning. Copying a variable implementation during cloning is implemented by a constructor and a `copy()` function. The constructor is straightforward and the `copy()` function only creates a new variable implementation if the variable implementation has not been copied before. If it has been copied before (that is, `copied()` returns **`true`**), the `copy()` function must return the forwarding pointer to the previously created copy as follows:

COPYING ≡
```
IntVarImp(Space& home, bool share, IntVarImp& y)
  : IntVarImpBase(home,share,y), l(y.l), u(y.u) {}
IntVarImp* copy(Space& home, bool share) {
  if (copied())
    return static_cast<IntVarImp*>(forward());
  else
    return new (home) IntVarImp(home,share,*this);
}
```

Additional inherited member functions. In addition to the constructor and the member functions defined and used by our variable implementation, several other member functions are typically just inherited and are defined by the class `VarImp`. The most important inherited member functions are summarized in Figure 34.5. For an explanation of degree and accumulated failure count, see Section 8.5.

	access operations
`degree()`	returns degree (number of subscriptions)
`afc()`	returns accumulated failure count
	scheduling support
`schedule()`	schedule propagator
	modification event deltas
`me()`	extract modification event
`med()`	construct modification event delta
	delta information access
`modevent()`	return modification event from delta

Figure 34.5: Summary of member functions predefined by variable implementations

34.4 Additional specification options

This section provides an overview of additional specification options not discussed in Section 34.2.

Comments. Any line starting with # is discarded and hence can serve as a comment in the specification file.

Generating headers, footers, and comments. Any text after the options for a **[ModEvent]** and **[PropCond]** definition until the next definition is added to the generated C++-code before the generated identifier definition. This can be used for defining comments to be added to the generated C++-code. For example, by

```
[PropCond]
Name:      NONE=NONE
// Propagation condition to be ignored
[PropCond]
...
```

the comment

```
// Propagation condition to be ignored
```

is put before the definition of the generated propagation condition.

Related support exists for putting a header before (or a footer after) all generated definitions for modification events and propagation conditions: The text following **[ModEventHeader]**, **[ModEventFooter]**, **[PropCondHeader]**, and **[PropCondFooter]** is inserted at the respective places in the generated code.

Conditional compilation. Giving an option `Ifdef` in the general section followed by some C++-preprocessor identifier `IDENT` wraps the entire generated code in preprocessor directives as follows:

```
#ifdef IDENT
  ...
#endif
```

By this, the Gecode kernel can be compiled with or without a particular variable type without being forced to reconfigure the Gecode kernel, see also Section 38.2.

Explicitly disposing variable implementations. Our example variables are entirely space-allocated and do not require external memory or other resources. However, for some variable types, the variable implementation might use external resources or memory that is not space-allocated and must explicitly be freed.

For an example, suppose the integer interval variables had been implemented by using arbitrary precision integers for the lower and upper bound and that these bounds must explicitly be freed.

By specifying in the general section

```
Dispose: true
```

and implementing in the variable implementation class a `dispose(Space& home)` member function, all variable implementations are disposed by calling their `dispose()` functions. The variable implementations are disposed when their home space is deleted.

Additionally, an object must be created that controls the disposal of variable implementations. Assume that our example integer variables used external memory and that its specification file contains

```
Dispose: true
```

and that `MPG::IntVarImp` implements a `dispose()` function. Then, your program must create a variable implementation disposer as follows:

```
Gecode::VarImpDisposer<MPG::IntVarImp> disposer;
```

The `disposer` object must be initialized before the first variable using `MPG::IntVarImp` is created.

Reserving bits. A limited number of bits b can be reserved within each variable implementation by specifying in the general section

```
Bits: b
```

Then, the variable implementation can get a reference to a value of type **`unsigned int`** by calling the member function `bits()` where the least b bits can be used freely. However, the maximal number of subscriptions (both propagators and advisors) for that variable implementation type is reduced from $2^{31}-1$ to $2^{31-b}-1$. Furthermore, any attempt to use more than the specified number of bits will crash Gecode in a truly spectacular fashion!

Specification file template. The specification template contains all possible specification options to assist in defining your own variable types.

35 Variables and variable arrays

This chapter describes how variables can be programmed from variable implementations and how variable arrays and variable argument arrays can be programmed. The chapter uses integer interval variables as introduced in Chapter 33 together with their implementations as defined in Chapter 34 as its running example.

Overview. How integer interval variables are implemented is detailed in Section 35.1. Variable arrays and variable argument arrays are discussed in Section 35.2.

35.1 Variables

As a variable is just a read-only interface to a variable implementation, its implementation is straightforward. The definition of integer variables is shown in Figure 35.1. The copy constructor uses the member function `varimp()` that returns the pointer to the variable's variable implementation. Note that every variable must have a constructor that takes a pointer to the corresponding variable implementation as argument.

Note that within the class `IntVar`, a pointer to the corresponding variable implementation is available as protected member `x` (see Tip 27.1 for information on **`using`**).

One also must define an output operator `<<` for a variable as shown in Figure 35.1.

It is important to remember that variables are defined in the namespace `MPG`. This is in contrast to variable implementations, which are defined in the namespace `MPG::Int`.

Variable creation. Creating a new variable is done with the following constructor that creates a new variable implementation as follows:

VARIABLE CREATION ≡

```
IntVar(Space& home, int min, int max)
  : VarImpVar<Int::IntVarImp>
      (new (home) Int::IntVarImp(home,min,max)) {
  if ((min < Int::Limits::min) || (max > Int::Limits::max))
    throw Int::OutOfLimits("IntVar::IntVar");
  if (min > max)
    throw Int::VariableEmptyDomain("IntVar::IntVar");
}
```

VARIABLE ≡

```
namespace MPG {

  class IntVar : public VarImpVar<Int::IntVarImp> {
  protected:
    using VarImpVar<Int::IntVarImp>::x;
  public:
    IntVar(void) {}
    IntVar(const IntVar& y)
      : VarImpVar<Int::IntVarImp>(y.varimp()) {}
    IntVar(Int::IntVarImp* y)
      : VarImpVar<Int::IntVarImp>(y) {}
    ▶ VARIABLE CREATION
    ▶ ACCESS OPERATIONS
  };

  template<class Char, class Traits>
  std::basic_ostream<Char,Traits>&
  operator <<(std::basic_ostream<Char,Traits>& os, const IntVar& x) {
    ...
  }

}
```

Figure 35.1: Variable programmed from a variable implementation

	access operations
`varimp()`	returns pointer to variable implementation
`assigned()`	whether variable is assigned
`degree()`	returns degree (number of subscriptions)
`afc()`	returns accumulated failure count
	update during cloning
`update()`	updates variable during cloning

Figure 35.2: Summary of member functions predefined by variables

Note that the constructor ensures the invariants for the lower and upper bound of a variable as discussed in Section 34.1 by possibly throwing exceptions.

Access operations. In addition to constructors, variables typically implement the same access operations as their corresponding variable implementation:

```
ACCESS OPERATIONS ≡
  int min(void) const {
    return x->min();
  }
  ...
```

Additional inherited member functions. In addition to the constructor and member functions defined by our variables, several other member functions are typically just inherited and are defined by the class `VarImpVar`. The most important inherited member functions are summarized in Figure 35.2. For an explanation of degree and accumulated failure count, see Section 8.5.

35.2 Variable arrays and variable argument arrays

Defining variable arrays and variable argument arrays (see also Section 4.2) requires the implementation of the arrays proper together with some traits. The traits classes for variable arrays and variable argument arrays ensure that Gecode-provided functionality for arrays can be used with the newly defined arrays.

Array traits. The definition of the array traits classes is shown in Figure 35.3. The definition is done in two steps. The first step provides forward declarations of the array types `IntVarArgs` and `IntVarArray` in the namespace `MPG` (because that is where these arrays will be defined).

ARRAY TRAITS ≡

```
namespace MPG {
  class IntVarArgs; class IntVarArray;
}

namespace Gecode {

  template<>
  class ArrayTraits<Gecode::VarArray<MPG::IntVar> > {
  public:
    typedef MPG::IntVarArray  StorageType;
    typedef MPG::IntVar       ValueType;
    typedef MPG::IntVarArgs   ArgsType;
  };
  template<>
  class ArrayTraits<MPG::IntVarArray> {
    ...
  };
  template<>
  class ArrayTraits<Gecode::VarArgArray<MPG::IntVar> > {
  public:
    typedef MPG::IntVarArgs   StorageType;
    typedef MPG::IntVar       ValueType;
    typedef MPG::IntVarArgs   ArgsType;
  };
  template<>
  class ArrayTraits<MPG::IntVarArgs> {
    ...
  };

}
```

Figure 35.3: Array traits for variable arrays

VARIABLE ARRAYS ≡

```
namespace MPG {

  class IntVarArgs : public VarArgArray<IntVar> {
  public:
    IntVarArgs(void) {}
    explicit IntVarArgs(int n) : VarArgArray<IntVar>(n) {}
    IntVarArgs(const IntVarArgs& a) : VarArgArray<IntVar>(a) {}
    IntVarArgs(const VarArray<IntVar>& a) : VarArgArray<IntVar>(a) {}
    IntVarArgs(Space& home, int n, int min, int max)
      : VarArgArray<IntVar>(n) {
      for (int i=0; i<n; i++)
        (*this)[i] = IntVar(home,min,max);
    }
  };

  class IntVarArray : public VarArray<IntVar> {
  public:
    IntVarArray(void) {}
    IntVarArray(const IntVarArray& a)
      : VarArray<IntVar>(a) {}
    IntVarArray(Space& home, int n, int min, int max)
      ...
    }
  };

}
```

Figure 35.4: Variable arrays

The second step requires to define traits for these two array types. The trait classes must be defined in the namespace `Gecode`. For each array type, two traits classes are needed: one for the base class (for example, `Gecode::VarArray<MPG::IntVar>`) and one for the class to be implemented (for example, `MPG::IntVarArray`). The definitions for the array type and its base class must be identical and follow the examples shown in Figure 35.3.

Variable arrays. The implementation of variable arrays and variable argument arrays typically only require the implementation of various constructors when inheriting from the base classes `VarArray` and `VarArgArray`. The minimal set of constructors such that the arrays are compatible to arrays as used by Gecode is shown in Figure 35.4.

36 Views

This chapter describes how views as needed for programming propagators and branchers can be programmed. The chapter uses integer interval variables as introduced in Chapter 33 together with their implementations as defined in Chapter 34 as its running example.

Overview. Section 36.1 provides an overview of the different types of views available in Gecode. The remaining sections provide examples for each different view type: Section 36.2 shows how an integer view `IntView` is constructed as a variable implementation view; Section 36.3 shows how a constant integer view `ConstIntView` is programmed as a constant view; Section 36.4 shows how a minus view `MinusView` and an offset view `OffsetView` are programmed as derived views.

36.1 View types

Gecode provides three different types of views:

- *Variable implementation views*: a variable implementation view is nothing but a direct interface to a variable implementation. A variable implementation view must inherit from `VarImpView`. The class `VarImpView` is parametric with respect to a variable and *not a variable implementation* (as one might expect). This is due to the fact that the type of the variable implementation can be obtained automatically from the type of a variable. Making a variable implementation view parametric with respect to a variable type has the advantage that information on both the variable type and variable implementation type become available.

- *Constant views*: a constant view must implement the same interface and must perform the same operations as some assigned variable implementation view. This particular variable implementation view is called the *corresponding* variable implementation view. A constant view must inherit from `ConstView` which is parametric with respect to the corresponding variable implementation view.

- *Derived views*: a derived view is a view that is implemented in terms of some other view (all view types are possible: variable implementation, constant, and derived). The view from which the derived view is derived, is called the *base* view. A derived view must inherit from `DerivedView` which is parametric with respect to the base view.

	access operations
`varimp()`	returns pointer to variable implementation
`assigned()`	whether variable is assigned
`degree()`	returns degree (number of subscriptions)
`afc()`	returns accumulated failure count
	subscriptions
`subscribe()`	subscribe propagator/advisor
`cancel()`	cancel propagator/advisor
	scheduling support
`schedule()`	schedule propagator
	modification event deltas
`me()`	extract modification event
`med()`	construct modification event delta
	delta information access
`modevent()`	return modification event from delta
	update during cloning
`update()`	updates view during cloning

Figure 36.1: Summary of member functions predefined by views

Predefined member functions. The classes `VarImpView`, `ConstView`, and `DerivedView` define already many member functions that simplify the implementation of new views. The most important predefined member functions are summarized in Figure 36.1.

Note that the `varimp()` function for a constant view or for a view derived from a constant view returns `NULL`, as no variable implementation exists.

View test functions. There are three different functions predefined for views:

- The function `shared(x,y)` returns **`true`**, if both views `x` and `y` share a common variable implementation (see Section 27.1.2). Typically, the definition of `shared()` does not need to be overloaded for newly defined views.
- The function `same(x,y)` returns **`true`**, if both views `x` and `y` are identical (see Section 27.1.2). In some cases, the definition of `same()` must be overloaded for newly defined views (see Section 36.3 and Section 36.4.2 for examples).
- The function `before(x,y)` returns **`true`**, if `x` comes before `y` in some arbitrary total and strict order for ordering views. The function is mainly used for sorting arrays of

```
INTEGER VIEW ≡
  namespace MPG { namespace Int {

    class IntView : public VarImpView<IntVar> {
    protected:
      using VarImpView<IntVar>::x;
    public:
      IntView(void) {}
      IntView(const IntVar& y)
        : VarImpView<IntVar>(y.varimp()) {}
      IntView(IntVarImp* y)
        : VarImpView<IntVar>(y) {}
      ▶ ACCESS OPERATIONS
      ▶ MODIFICATION OPERATIONS
      ▶ DELTA INFORMATION
    };

    template<class Char, class Traits>
    std::basic_ostream<Char,Traits>&
    operator<<(std::basic_ostream<Char,Traits>& os, const IntView& x) {
      ...
    }

  }}
```

Figure 36.2: Integer view

views into some order (in particular for detecting duplicate views). In some cases, the definition of `before()` must be overloaded for newly defined views (see Section 36.3 and Section 36.4.2 for examples).

Output operator. For every view also an output operator << must be defined. We sketch this only for integer views in Section 36.2, for all other views the definition is analogous.

36.2 Variable implementation views: integer view

Figure 36.2 shows the definition of the class `IntView` for integer views from the class `VarImpView` for variable implementation views. Please remember that a variable implementation view is parametric with respect to a variable type (`IntVar` in our example, such that `IntView` uses the same variable implementation type `IntVarImp` as `IntVar` does).

Similar to variables obtained from variable implementations, a variable implementation

view has a protected member x that is a pointer to its variable implementation (see Tip 27.1 for information on **using**). A variable implementation view must implement at least the shown constructors such that it can be initialized both from the corresponding variable type and from the corresponding variable implementation type.

The remaining implementation tasks for variable implementation views are straightforward: all operations that are specific to a variable type (in our case, specific to integer interval variables) must be implemented. The implementation is straightforward as only the corresponding operations of the variable implementation are invoked:

- The access operations must be implemented:

```
ACCESS OPERATIONS ≡
  int min(void) const {
    return x->min();
  }
  ...
```

- The modification operations must be implemented:

```
MODIFICATION OPERATIONS ≡
  ModEvent lq(Space& home, int n) {
    return x->lq(home,n);
  }
  ...
```

- Finally, the operations for accessing delta information must be implemented:

```
DELTA INFORMATION ≡
  int min(const Delta& d) const {
    return IntVarImp::min(d);
  }
  ...
```

36.3 Constant views: constant integer view

Figure 36.3 shows the implementation of a constant integer view with `IntView` as the corresponding variable implementation view. A constant integer view `ConstIntView` stores an integer value x and must implement all variable-specific operations that are implemented by the corresponding `IntView` class (as shown in Figure 36.3).

Slightly less obvious is the implementation of operations that access delta information. While these operations must be implemented such that constant integer views can be used instead of integer views, they will never be executed (by definition, a constant view can never change). Hence we use the macro `GECODE_NEVER` (see Tip 31.1) to clarify that the delta information operations are never executed:

CONSTANT INTEGER VIEW ≡

```
  namespace MPG { namespace Int {

    class ConstIntView : public ConstView<IntView> {
    protected:
      int x;
    public:
      ConstIntView(void) : x(0) {}
      ConstIntView(int n) : x(n) {}

      int min(void) const {
        return x;
      }
      ...
      ModEvent lq(Space& home, int n) {
        return (x <= n) ? ME_INT_NONE : ME_INT_FAILED;
      }
      ...
      ▶ DELTA INFORMATION
      ▶ UPDATE DURING CLONING
    };
    ▶ VIEW TESTS

    ...

  }}
```

Figure 36.3: Constant integer view

DELTA INFORMATION ≡
```
int min(const Delta& d) const {
  GECODE_NEVER; return 0;
}
...
```

Update during cloning. The definition of the `update()` member function of `ConstView` does not take care of the integer value `x`. Hence we need to provide a new `update()` function that updates the value of `x` as follows:

UPDATE DURING CLONING ≡
```
void update(Space& home, bool share, ConstIntView& y) {
  ConstView<IntView>::update(home,share,y);
  x = y.x;
}
```

View tests. Also the default definitions of the view test functions `same()` and `before()` for constant views do not take the integer value `x` of the view into account. Overloaded versions for constant integer views are as follows:

VIEW TESTS ≡
```
inline bool same(const ConstIntView& x, const ConstIntView& y) {
  return x.min() == y.min();
}
inline bool before(const ConstIntView& x, const ConstIntView& y) {
  return x.min() < y.min();
}
```

36.4 Derived views

This section exemplifies two different derived views: minus views and offset views. Why these views are useful and what their semantics is can be seen in Section 27.1.1 for minus views and in Section 27.1.2 for offset views.

36.4.1 Minus views

Figure 36.4 shows that a minus view is derived from an integer view `IntView`. The protected member `x` refers to the base view, that is the integer view from which the minus view is

MINUS VIEW ≡

```
namespace MPG { namespace Int {

  class MinusView : public DerivedView<IntView> {
  protected:
    using DerivedView<IntView>::x;
    ▶ MODIFICATION EVENTS AND PROPAGATION CONDITIONS
  public:
    MinusView(void) {}
    explicit MinusView(const IntView& y)
      : DerivedView<IntView>(y) {}
    ▶ ACCESS OPERATIONS
    ▶ MODIFICATION OPERATIONS
    ▶ SUPPORT OPERATIONS
    ▶ SUBSCRIPTIONS
    ▶ DELTA INFORMATION
  };

  ...

}}
```

Figure 36.4: Minus view

```
MODIFICATION EVENTS AND PROPAGATION CONDITIONS ≡
  static ModEvent minusme(ModEvent me) {
    switch (me) {
    case ME_INT_MIN: return ME_INT_MAX;
    case ME_INT_MAX: return ME_INT_MIN;
    default: return me;
    }
  }
  static PropCond minuspc(PropCond pc) {
    ...
  }
```

Figure 36.5: Negation of modification events and propagation conditions

derived (see Tip 27.1 for information on **using**).

Access operations. The access operations are as to be expected for a minus view. That is, the lower bound of the derived view is the negation of the upper bound of the base view:

```
ACCESS OPERATIONS ≡
  int min(void) const {
    return -x.max();
  }
  ...
```

Modification operations. The modification operations are slightly more involved than the access operations as they return a modification event. If the modification event of the base view is `ME_INT_MAX` (the upper bound of the base view has changed), then the modification event for the derived view must be `ME_INT_MIN` (the lower bound of the derived view has changed).

Figure 36.5 shows functions `minusme()` and `minuspc()` that return the negation of modification events and propagation conditions (to be discussed later).

Using the function `minusme`, the modification operations can be defined as follows:

```
MODIFICATION OPERATIONS ≡
  ModEvent lq(Space& home, int n) {
    return minusme(x.gq(home,-n));
  }
  ...
```

Accessing delta information. Accessing delta information must also take into account that the modification event stored in a delta must be converted with `minusme()`. Also the other operations for accessing delta information must be adopted accordingly:

```
DELTA INFORMATION ≡
  static ModEvent modevent(const Delta& d) {
    return minusme(IntView::modevent(d));
  }
  int min(const Delta& d) const {
    return -x.max(d);
  }
  ...
```

Additional operations. Any operation that is concerned with either modification events or propagation conditions must be implemented to take the switch between lower bound and upper bound into account. These operations include the operations for handling subscriptions of propagators (the function `minuspc()` is defined analogously to `minusme()` in Figure 36.5):

```
SUBSCRIPTIONS ≡
  void subscribe(Space& home, Propagator& p, PropCond pc,
                 bool schedule=true) {
    x.subscribe(home,p,minuspc(pc),schedule);
  }
  void subscribe(Space& home, Advisor& a) {
    x.subscribe(home,a);
  }
  ...
```

Note that the operations that subscribe and cancel advisors must be re-implemented even though they are unchanged. This is due to inheritance in C++: as the overloaded functions for propagators are redefined, also the functions for advisors are considered to be redefined.

The remaining operations to be implemented are support operations:

```
SUPPORT OPERATIONS ≡
  static void schedule(Space& home, Propagator& p, ModEvent me) {
    return IntView::schedule(home,p,minusme(me));
  }
  static ModEvent me(const ModEventDelta& med) {
    return minusme(IntView::me(med));
  }
  static ModEventDelta med(ModEvent me) {
    return IntView::med(minusme(me));
  }
```

36.4.2 Offset views

Figure 36.6 shows that an offset view is derived from an integer view `IntView` and stores an additional integer value `c` for the offset. The protected member `x` refers to the base view,

OFFSET VIEW ≡

```
namespace MPG { namespace Int {

  class OffsetView : public DerivedView<IntView> {
  protected:
    using DerivedView<IntView>::x;
    int c;
  public:
    OffsetView(void) {}
    OffsetView(const IntView& y, int d)
      : DerivedView<IntView>(y), c(d) {}

    int offset(void) const {
      return c;
    }
    int min(void) const {
      return x.min()+c;
    }
    ...
    ModEvent lq(Space& home, int n) {
      return x.lq(home,n-c);
    }
    ...
    int min(const Delta& d) const {
      return x.min(d)+c;
    }
    ...
    ▶ UPDATE DURING CLONING
  };
  ▶ VIEW TESTS
  ...
}}
```

Figure 36.6: Offset view

that is the integer view from which the offset view is derived (see Tip 27.1 for information on **using**). The access, modification, and delta information access operations of an offset view are as to be expected.

The `update()` function must also update the integer offset `c` as follows:

UPDATE DURING CLONING ≡

```
void update(Space& home, bool share, OffsetView& y) {
  x.update(home,share,y.x);
  c=y.c;
}
```

Likewise, the view test functions `same()` and `before()` must take into account the integer offset `c`:

VIEW TESTS ≡

```
inline bool same(const OffsetView& x, const OffsetView& y) {
  return same(x.base(),y.base()) && (x.offset() == y.offset());
}
inline bool before(const OffsetView& x, const OffsetView& y) {
  return before(x.base(),y.base())
    || (same(x.base(),y.base()) && (x.offset() < y.offset()));
}
```

37 Variable-value branchings

This chapter explains how to program common variable-value branchings using the abstractions provided by Gecode.

Overview. Section 37.1 explains which simple types must be defined for variable-value branchings. How functions for variable selection and value selection are implemented is demonstrated in Section 37.2. Section 37.3 shows how a function that creates an object for selecting views during branching is implemented. How functions for selecting values and committing to these values are implemented is shown in Section 37.4. This section also explains how add support for no-goods to a variable-value brancher. How the actual branchings are implemented is then detailed in Section 37.5.

This structure is reflected in the part of the `int.hh` header file that is concerned with branching (shown in Figure 37.1).

37.1 Type, traits, and activity definitions

The way how a variable-value branching works can to some extent be controlled by the user by passing pointers to functions:

- A branching filter function defines which variables are actually considered for branching, see Section 8.11.
- A variable value print function defines how to print information on alternatives of variable value branchers during search. Variable value branchers print some default information even if the user does not supply a variable value print function, see Section 8.12.
- A merit function can define which variable is selected for branching, see Section 8.7.
- A branch value function selects a value that is used for branching, see Section 8.8.
- A branch commit function constrains a variable with respect to a value passed as argument, see Section 8.8.

In the following type definitions, the type `IntVar` of the argument `x`, the return type **`int`** of the branch value function of type `IntBranchVal`, and the type **`int`** of the argument `n` is

```
BRANCHING ≡
  ▶ BRANCH FUNCTION TYPES
  ▶ BRANCH TRAITS

  ▶ VARIABLE AFC

  ▶ VARIABLE ACTIVITY

  namespace MPG {
    ▶ VARIABLE SELECTION CLASS
    ▶ VARIABLE SELECTION FUNCTIONS
    ▶ VALUE SELECTION FUNCTIONS
  }

  namespace MPG { namespace Int {
    ▶ VIEW SELECTION CREATION FUNCTION
    ▶ VALUE SELECTION AND COMMIT CREATION FUNCTION
  }}

  namespace MPG {
    ▶ BRANCH FUNCTION
    ▶ BRANCH FUNCTION WITH TIE-BREAKING
  }
```

Figure 37.1: Part of header file concerned with branching

dependent on our integer interval variables (they are of type `IntVar` and they take values of type **`int`**).

The remaining argument and return types are required by Gecode and are as follows:

BRANCH FUNCTION TYPES ≡

```
namespace MPG {
  typedef bool   (*IntBranchFilter)(const Space& home,
                                    IntVar x, int i);
  typedef void   (*IntVarValPrint) (const Space &home,
                                    const BrancherHandle& bh,
                                    unsigned int a,
                                    IntVar x, int i, const int& n,
                                    std::ostream& o);
  typedef double (*IntBranchMerit) (const Space& home,
                                    IntVar x, int i);
  typedef int    (*IntBranchVal)   (const Space& home,
                                    IntVar x, int i);
  typedef void   (*IntBranchCommit)(Space& home, unsigned int a,
                                    IntVar x, int i, int n);
}
```

These function type definitions must be connected to the variable type `IntVar` by means of a traits-class of type `BranchTraits`. As the functionality for variable-value branching is defined in the `Gecode` namespace, the trait class must also be defined there:

BRANCH TRAITS ≡

```
namespace Gecode {
  template<>
  class BranchTraits<MPG::IntVar> {
  public:
    typedef MPG::IntBranchFilter Filter;
    typedef MPG::IntBranchMerit Merit;
    typedef MPG::IntBranchVal Val;
    typedef int ValType;
    typedef MPG::IntBranchCommit Commit;
  };
}
```

The last remaining definitions specializes AFC and activity information for integer interval variables by defining a class `IntAFC` as follows:

VARIABLE AFC ≡

```
namespace MPG {
  class IntAFC : public AFC {
  public:
    IntAFC(void);
    IntAFC(const IntAFC& a);
    IntAFC& operator =(const IntAFC& a);
    IntAFC(Home home, const IntVarArgs& x, double d=1.0);
    void init(Home home, const IntVarArgs& x, double d=1.0);
  };
  ...
}
```

and a class `IntActivity` as follows:

VARIABLE ACTIVITY ≡

```
namespace MPG {
  class IntActivity : public Activity {
  public:
    IntActivity(void);
    IntActivity(const IntActivity& a);
    IntActivity& operator =(const IntActivity& a);
    IntActivity(Home home, const IntVarArgs& x, double d=1.0,
                IntBranchMerit bm=NULL);
    void init(Home home, const IntVarArgs& x, double d=1.0,
              IntBranchMerit bm=NULL);
  };
  ...
}
```

The actual implementations are omitted as they contain nothing more than the type specialization and creation of view arrays in the initializing constructor and the `init()` function.

37.2 Variable and value selection

An important part of the interface of the branching is support for specifying how variables and values are selected for branching. This is implemented by a set of variable and value selection functions that are used for specification. These functions return objects that are then used for creating the appropriate branchers. In this section we are not interested in describing a complete set of variable and value selection functions but in a set that demonstrates the features of variable-value branchings well.

Variable selection. The variable selection functions we are considering here are defined as follows (their names and what they do coincides with the variable selection functions for normal integer variables in Gecode, see Section 8.2):

```
VARIABLE SELECTION FUNCTIONS ≡
  IntVarBranch INT_VAR_NONE(void);
  IntVarBranch INT_VAR_RND(Rnd r);
  IntVarBranch INT_VAR_MERIT_MAX(IntBranchMerit bm,
                                 BranchTbl tbl=NULL);
  IntVarBranch INT_VAR_DEGREE_MAX(BranchTbl tbl=NULL);
  IntVarBranch INT_VAR_ACTIVITY_MAX(double d=1.0,
                                    BranchTbl tbl=NULL);
  IntVarBranch INT_VAR_ACTIVITY_MAX(IntActivity a,
                                    BranchTbl tbl=NULL);
  IntVarBranch INT_VAR_SIZE_MIN(BranchTbl tbl=NULL);
  ▶ VARIABLE SELECTION FUNCTION IMPLEMENTATION
```

All but `INT_VAR_NONE()` take arguments: unsurprisingly, a random number generator must be passed to `INT_VAR_RND()` and a double as decay-factor or an integer activity object to `INT_VAR_ACTIVITY_MAX()`. Both `INT_VAR_NONE()` and `INT_VAR_RND()` are special in that they are not useful for tie-breaking. All other variable selection functions take an optional argument of type `BranchTbl` as a branch tie-breaking limit function (we will abbreviate this here as tbl-function), see Section 8.9 for a description of tie-breaking and tbl-functions.

The implementation of the variable selection functions is simple: each function returns an object of class `IntVarBranch` that stores all necessary information required for creating the appropriate brancher. As an example of an implementation consider the following, the other functions are similar:

```
VARIABLE SELECTION FUNCTION IMPLEMENTATION ≡
  inline IntVarBranch
  INT_VAR_MERIT_MAX(IntBranchMerit bm, BranchTbl tbl) {
    return IntVarBranch(IntVarBranch::SEL_MERIT_MAX,
                        function_cast<VoidFunction>(bm),tbl);
  }
  ...
```

The function `function_cast()` is defined by Gecode and can be used to cast function pointers[1]. The type `VoidFunction` is a generic base type for function pointers defined by Gecode.

The implementation of the class `IntVarBranch` is shown in Figure 37.2. It defines an enumeration of all variable selection strategies and a set of constructors for the different types of arguments the variable selection functions take. The `select()` function returns a

[1]Normal pointers (that is, pointers to data) and function pointers cannot be casted into each other in C++, that is why Gecode provides the function `function_cast()`.

VARIABLE SELECTION CLASS ≡

```
class IntVarBranch : public VarBranch {
public:
  enum Select {
    SEL_NONE,       SEL_RND,          SEL_MERIT_MAX,
    SEL_DEGREE_MAX, SEL_ACTIVITY_MAX, SEL_SIZE_MIN
  };
protected:
  Select s;
public:
  IntVarBranch(void) ;
  IntVarBranch(Rnd r);
  IntVarBranch(Select s0, BranchTbl t);
  IntVarBranch(Select s0, double d, BranchTbl t);
  IntVarBranch(Select s0, Activity a, BranchTbl t);
  IntVarBranch(Select s0, VoidFunction mf, BranchTbl t);
  Select select(void) const;
  ▶ EXPAND ACTIVITY
};
...
```

Figure 37.2: Variable selection class

value of the enumeration type that is stored by the object. All other information is handled by the base class `VarBranch`.

The class must also implement an `expand()` member function. It checks whether `INT_VAR_ACTIVITY_MAX()` had been called just with a decay-factor instead of an integer activity object. In this case it creates an integer activity object and stores it as follows:

EXPAND ACTIVITY ≡
```
void expand(Home home, const IntVarArgs& x) {
  if ((select() == SEL_ACTIVITY_MAX) &&
      !activity().initialized())
    activity(IntActivity(home,x,decay()));
}
```

Value selection. Value selection functions are implemented similarly to variable selection functions. They return an object of class `IntValBranch` (inheriting from the base class `ValBranch`) which stores the necessary information for creating the appropriate brancher. We are considering the following value selection functions as examples:

VALUE SELECTION FUNCTIONS ≡
```
class IntValBranch : public ValBranch {
  ...
};
IntValBranch INT_VAL_MIN(void);
IntValBranch INT_VAL_RND(Rnd r);
IntValBranch INT_VAL(IntBranchVal v, IntBranchCommit c=NULL);
...
```

Note that the last argument of the value selection function `INT_VAL()` is optional, the default behavior will be defined in Section 37.4.

37.3 View selection creation

The view selection creation function shown in Figure 37.3 takes an object `ivb` of class `IntVarBranch` as an argument, creates an object of class `ViewSel` and returns a pointer to it. The object `ivb` is a specification of which object should be returned. The returned object is used to select views during brancher execution.

Selection of the first unassigned view (corresponding to `SEL_NONE`, that is, the object `ivb` has been created by calling the function `INT_VAR_NONE()`) is implemented by the Gecode-defined class `ViewSelNone`. Also random view selection is provided by Gecode through the class `ViewSelRnd`. Both classes are parametric with respect to a view type.

```
VIEW SELECTION CREATION FUNCTION ≡
  ▶ SIZE MERIT CLASS
  inline ViewSel<IntView>*
  viewsel(Space& home, const IntVarBranch& ivb) {
    if (ivb.select() == IntVarBranch::SEL_NONE)
      return new (home) ViewSelNone<IntView>(home,ivb);
    if (ivb.select() == IntVarBranch::SEL_RND)
      return new (home) ViewSelRnd<IntView>(home,ivb);
    if (ivb.tbl() != NULL) {
      ▶ VIEW SELECTION WITH TBL-FUNCTION
    } else {
      ▶ VIEW SELECTION WITHOUT TBL-FUNCTION
    }
    throw UnknownBranching("Int::branch");
  }
```

Figure 37.3: View selection creation function

View selection with tbl-function. The other strategies for view selection exist in two variants: one variant that uses a tbl-function and one variant that does not. In case a tbl-function has been supplied as additional argument to one of the variable selection functions, the following creates the appropriate object for view selection:

```
VIEW SELECTION WITH TBL-FUNCTION ≡
  switch (ivb.select()) {
  case IntVarBranch::SEL_MERIT_MAX:
    return new (home) ViewSelMaxTbl<MeritFunction<IntView> >(home,ivb);
  case IntVarBranch::SEL_DEGREE_MAX:
    return new (home) ViewSelMaxTbl<MeritDegree<IntView> >(home,ivb);
  case IntVarBranch::SEL_ACTIVITY_MAX:
    return new (home) ViewSelMaxTbl<MeritActivity<IntView> >(home,ivb);
  case IntVarBranch::SEL_SIZE_MIN:
    return new (home) ViewSelMinTbl<MeritSize>(home,ivb);
  default: ;
  }
```

Depending on how the view is to be selected, different objects are created. An object of class `ViewSelMaxTbl` selects a variable with maximal merit (for the definition of merit, see Section 8.2), whereas an object of class `ViewSelMinTbl` selects a variable with minimal merit. Objects of both classes take a tbl-function during selection into account. Both classes expect a class as template argument that computes the actual merit value for a given view.

The classes `MeritFunction`, `MeritDegree`, and `MeritActivity` are defined by Gecode and are parametric with respect to the actual view type.

SIZE MERIT CLASS ≡
```
class MeritSize : public MeritBase<IntView,unsigned int> {
public:
  MeritSize(Space& home, const VarBranch& vb)
    : MeritBase<IntView,unsigned int>(home,vb) {}
  MeritSize(Space& home, bool share, MeritSize& m)
    : MeritBase<IntView,unsigned int>(home,share,m) {}
  unsigned int operator ()(const Space& home, IntView x, int i) {
    return x.max() - x.min();
  }
};
```

Figure 37.4: Size merit class

Selecting a view with minimal size is specific to our integer interval variables and views. The implementation of the class `MeritSize` inherits from `MeritBase` and is shown in Figure 37.4.

The class `MeritBase` is parametric with respect to the view type (`IntView` in our case) and the type of the merit value (**`unsigned int`** in our case). The constructors are as to be expected and the call operator must return the merit value of type **`unsigned int`** (the same as the second template argument to `MeritBase`) of the view `x` (`i` refers to the position of the view `x` in the array of views used in the brancher).

In case the merit class uses members that must be deallocated when the home-space is deleted, the merit class must redefine the member functions `notice()` and `dispose()`, for example by:

```
  bool notice(void) const {
    return true;
  }
  void dispose(Space& home) {
    ...
  }
```

View selection without tbl-function. Implementing view selection without a tbl-function is analogous, the only difference is that the classes `ViewSelMax` (instead of `ViewSelMaxTbl`) and `ViewSelMin` (instead of `ViewSelMinTbl`) must be used:

VIEW SELECTION WITHOUT TBL-FUNCTION ≡
```
  switch (ivb.select()) {
  case IntVarBranch::SEL_MERIT_MAX:
    return new (home) ViewSelMax<MeritFunction<IntView> >(home,ivb);
  ...
  }
```

```
VALUE SELECTION AND COMMIT CREATION FUNCTION ≡
  ▶ VALUE SELECTION CLASSES
  ▶ VALUE COMMIT CLASS
  inline ValSelCommitBase<IntView,int>*
  valselcommit(Space& home, const IntValBranch& ivb) {
    switch (ivb.select()) {
    case IntValBranch::SEL_MIN:
      return new (home)
        ValSelCommit<ValSelMin,ValCommitLq>(home,ivb);
    case IntValBranch::SEL_RND:
      return new (home)
        ValSelCommit<ValSelRnd,ValCommitLq>(home,ivb);
    case IntValBranch::SEL_VAL_COMMIT:
      ▶ USER-DEFINED VALUE SELECTION AND COMMIT FUNCTIONS
    default:
      throw UnknownBranching("Int::branch");
    }
  }
```

Figure 37.5: Value selection and commit creation function

37.4 Value selection and commit creation

The value selection and commit creation function is very similar to the variable selection creation function from the previous section. It creates and returns an object that performs value selection and value commit during branching depending on a specification object of class `IntValBranch`.

The function is shown in Figure 37.5 and returns an object of class `ValSelCommitBase`. Again, this class is parametric with respect to the view type (`IntView`) and the value type (**`int`**). Depending on which value selection strategy is defined by the argument `ivb`, a corresponding object of class `ValSelCommit` is created.

The class `ValSelCommit` is parametric with respect to a value selection class and a value commit class (to be discussed below). The classes `ValSelMin`, `ValSelRnd`, and `ValCommitLq` are specific to integer interval variables and views and are discussed below.

Value selection classes. A value selection class must inherit from the class `ValSel` which again is parametric with respect to the view and value type. The constructors (one for creation and for cloning) are exactly the same as for merit classes discussed in the previous section.

Also, similar to merit classes, a value selection class can redefine the member functions `notice()` and `dispose()` if explicit disposal is required when the home-space is deleted.

In addition, the classes must define a member function `val()` that returns a value for a given view `x` as follows (`i` again is the position in the view array):

VALUE SELECTION CLASSES ≡
```
class ValSelMin : public ValSel<IntView,int> {
  ...
  int val(const Space& home, IntView x, int i) {
    return x.min();
  }
};
...
```

Value commit classes. For our integer interval variables and views we need a single value commit class only (how many classes are needed depends of course on which value selection strategies are provided). A value commit class must inherit from the parametric class `ValCommit` and must implement one constructor for creation and one for cloning. In addition, it must define a `commit()` function, an `ngl()` function (to be discussed later), and a default `print()` function. The `commit()` function returns a modification event and takes the number of the alternative `a`, a view `x`, its position `i`, and a value `n` as arguments. The `print()` function takes an output stream `o` as additional argument:

VALUE COMMIT CLASS ≡
```
  ▶ NO-GOOD LITERAL CLASS
...
class ValCommitLq : public ValCommit<IntView,int> {
public:
  ...
  ModEvent commit(Space& home, unsigned int a,
                  IntView x, int i, int n) {
    return (a == 0) ? x.lq(home,n) : x.gq(home,n+1);
  }
  void print(const Space&, unsigned int a,
             IntView, int i, int n,
             std::ostream& o) const {
    o << "x[" << i << "] "
      << ((a == 0) ? "<=" : ">") << " " << n;
  }
  ▶ NO-GOOD LITERAL CREATION
};
```

No-good support. The value commit class must also implement a function `ngl()` that returns a no-good literal for an alternative. The idea is exactly the same as described in Section 32.2, the only difference is that the `ngl()` function here gets a view and a value as arguments rather than a choice.

The `ngl()` function of the `ValCommitLq` class returns a no-good literal implemented by the class `LqNGL` for the first alternative and `NULL` for the second alternative as follows:

NO-GOOD LITERAL CREATION ≡
```
NGL* ngl(Space& home, unsigned int a,
         IntView x, int n) const {
  if (a == 0)
    return new (home) LqNGL(home,x,n);
  else
    return NULL;
}
```

The no-good literal class `LqNGL` used by the `ngl()` function is defined as follows:

NO-GOOD LITERAL CLASS ≡
```
class LqNGL : public ViewValNGL<IntView,int,PC_INT_BND> {
  using ViewValNGL<IntView,int,PC_INT_BND>::x;
  using ViewValNGL<IntView,int,PC_INT_BND>::n;
public:
  LqNGL(Space& home, IntView x, int n);
  LqNGL(Space& home, bool share, LqNGL& ngl);
  virtual NGL* copy(Space& home, bool share);
  virtual NGL::Status status(const Space& home) const;
  virtual ExecStatus prune(Space& home);
};
```

It inherits from the template class `ViewValNGL`, which expects a view type, a value type, and a propagation condition as argument. The definition of the constructors, the `copy()` function, the `status()` function, and the `prune()` function are exactly as discussed in Section 32.2. The remaining functions for disposal and subscription are pre-defined by `ViewValNGL`.

User-defined value selection and commit functions. For the value selection function `INT_VAL(v,c)` for a user-defined value selection function `v` and a user-defined commit function `c` it is possible to leave out `c`, as it has been declared as an optional argument. When the argument is not provided, `c` is equal to `NULL`. This is taken into account as follows:

BRANCH FUNCTION ≡

```
inline BrancherHandle
branch(Home home, const IntVarArgs& x,
       IntVarBranch vars, IntValBranch vals,
       IntBranchFilter bf=NULL,
       IntVarValPrint vvp=NULL) {
  using namespace Int;
  if (home.failed()) return BrancherHandle();
  vars.expand(home,x);
  ViewArray<IntView> xv(home,x);
  ViewSel<IntView>* vs[1] = {
    viewsel(home,vars)
  };
  return ViewValBrancher<IntView,1,int,2>::post
    (home,xv,vs,valselcommit(home,vals),bf,vvp);
}
```

Figure 37.6: Branch function

USER-DEFINED VALUE SELECTION AND COMMIT FUNCTIONS ≡

```
if (ivb.commit() == NULL) {
  return new (home)
    ValSelCommit<ValSelFunction<IntView>,
                 ValCommitLq>(home,ivb);
} else {
  return new (home)
    ValSelCommit<ValSelFunction<IntView>,
                 ValCommitFunction<IntView> >(home,ivb);
}
```

The classes `ValSelFunction` and `ValCommitFunction` are defined by Gecode and are parametric with respect to a view. They use the functions as specified by the object `ivb`.

37.5 Branchings

Implementing the actual `branch()` functions with and without tie-breaking is straightforward. They only have to create a brancher that uses the view selection creation function `viewsel()` from Section 37.3 and the value selection and commit creation function `valselcommit()` from Figure 37.5.

Branching without tie-breaking. The `branch()` function is shown in Figure 37.6. It creates an array of integer views `IntView`, expands a possibly missing integer activity object,

creates an array with a single view selector object returned by the function `viewsel()` as discussed in Section 37.3, posts the view-value brancher of class `ViewValBrancher`, and returns a brancher handle of class `BrancherHandle` (see also Section 8.1) to the posted brancher. The class is parametric, where the arguments describe the following:

1. The view type which is `IntView` in our case.
2. The number of view selection objects to be used during view selection. As we are not using tie-breaking, the number is `1` and corresponds to the number of elements in the array `vs`.
3. The value type which is **`int`** in our case.
4. The number of alternatives that should be created during branching, which is `2` in our example[2].

Branching with tie-breaking. The `branch()` function with tie-breaking is shown in Figure 37.7. It takes an object `vars` of class `TieBreak` as argument, where `vars.a` is the first variable selection strategy of class `IntVarBranch`, `vars.b` the second, `vars.c` the third, and `vars.d` the forth and last to be used during tie-breaking.

Before creating the brancher, the variable selection strategies are normalized. As mentioned earlier, there should be no tie-breaking after the variable selection strategies `INT_VAR_NONE()` and `INT_VAR_RND()` (corresponding to `SEL_NONE` and `SEL_RND`, respectively). The normalization first tries to normalize `var.b`, then `var.c` and finally `var.d` as follows (the `var.c` and `var.d` case is analogous and hence omitted):

NORMALIZING TIE-BREAKING ≡

```
if ((vars.a.select() == IntVarBranch::SEL_NONE) ||
    (vars.a.select() == IntVarBranch::SEL_RND))
  vars.b = INT_VAR_NONE();
vars.b.expand(home,x);
if ((vars.b.select() == IntVarBranch::SEL_NONE) ||
    (vars.b.select() == IntVarBranch::SEL_RND))
  vars.c = INT_VAR_NONE();
vars.c.expand(home,x);
if ((vars.c.select() == IntVarBranch::SEL_NONE) ||
    (vars.c.select() == IntVarBranch::SEL_RND))
  vars.d = INT_VAR_NONE();
vars.d.expand(home,x);
```

After normalization, the `branch()` function shown in Figure 37.7 posts a brancher of class `ViewValBrancher` with the appropriate number of view selection objects. In Figure 37.7, only the cases for two and three objects is shown the other cases are analogous.

[2]By choosing the value `1` here, one can obtain branchers that perform value assignment similar to the `assign()` function described in Section 8.13.

BRANCH FUNCTION WITH TIE-BREAKING ≡

```
inline BrancherHandle
branch(Home home, const IntVarArgs& x,
       TieBreak<IntVarBranch> vars, IntValBranch vals,
       IntBranchFilter bf=NULL,
       IntVarValPrint vvp=NULL) {
  using namespace Int;
  if (home.failed()) return BrancherHandle();
  vars.a.expand(home,x);
  ▶ NORMALIZING TIE-BREAKING
  ViewArray<IntView> xv(home,x);
  if (vars.b.select() == IntVarBranch::SEL_NONE) {
    ...
  } else if (vars.c.select() == IntVarBranch::SEL_NONE) {
    ViewSel<IntView>* vs[2] = {
      viewsel(home,vars.a), viewsel(home,vars.b)
    };
    return ViewValBrancher<IntView,2,int,2>
      ::post(home,xv,vs,valselcommit(home,vals),bf,vvp);
  } else if (vars.d.select() == IntVarBranch::SEL_NONE) {
    ViewSel<IntView>* vs[3] = {
      viewsel(home,vars.a), viewsel(home,vars.b),
      viewsel(home,vars.c)
    };
    return ViewValBrancher<IntView,3,int,2>
      ::post(home,xv,vs,valselcommit(home,vals),bf,vvp);
  } else {
    ...
  }
}
```

Figure 37.7: Branch function with tie-breaking

38 Putting everything together

This chapter finally explains how integer interval variables can be used with Gecode.

Overview. Section 38.1 sketches an example script together with implementations of constraints and branchings using integer interval variables. The following section, Section 38.2, shows how Gecode can be configured to use integer interval variables and how to compile and run the example script.

Important. Please make sure to carefully read Section 2.6.2, before reading any further in this chapter!

38.1 Golomb rulers à la integer interval variables

Figure 38.1 shows the top-level structure of a single C++-file containing a script together with all required implementations of post functions, propagators, and branchers. The example script implements a naive version of the Golomb ruler model presented in Chapter 12. The reason to package everything into a single C++-file is to simplify compiling the example.

The implementations of the constraints in the C++-file are carefully constructed to exercise most of the functionality described in the previous chapters in this part. In particular, some constraints have a slightly non-standard implementation to exercise all views presented in Chapter 36.

38.2 Configuring and compiling Gecode

The following steps configure and compile Gecode with integer interval variables:

1. Start a shell.
2. Create a new directory, say `MPG`, and make it the current directory:

   ```
   mkdir MPG; cd MPG
   ```

3. Download a Gecode 4.3.2 source package or check-out Gecode 4.3.2 from `svn`. We assume that the Gecode source code is contained in a directory named `gecode-4.3.2`.

PUTTING EVERYTHING TOGETHER ≡ [DOWNLOAD]

```
#include "int.hh"

#include <gecode/search.hh>

using namespace MPG;

...

class GolombRuler : public Gecode::Space {
  ...
};

int main(int argc, char* argv[]) {
  ...
}
```

Figure 38.1: Golomb rulers à la integer interval variables

4. If you have not yet done so, download and copy all files required for integer interval variables and the example into the current directory:

   - The header file `int.hh` containing the implementation of integer interval variables.
   - The variable implementation specification file `int.vis`.
   - The file `putting-everything-together.cpp` from the previous section.

5. Configure Gecode to incorporate integer interval variables:

   ```
   cd gecode-4.3.2
   ./configure --with-vis=../int.vis
   ```

   After this step, Gecode has been configured to incorporate the generated definitions as described by the specification file `int.vis`. If you need to pass other options to `configure` to successfully build Gecode, please do so.

6. Compile Gecode and leave the directory

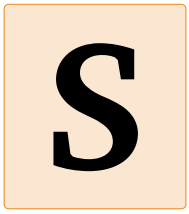
   ```
   make;cd ..
   ```

7. Set the `PATH` environment variable to point to the just compiled Gecode installation:

```
export PATH="gecode-4.3.2:$PATH"
```

   Depending on the platform you use, you might also have to set the environment variable `LD_LIBRARY_PATH` accordingly.

Finally: compile, link, and run the example script `putting-everything-together.cpp` as described in Section 2.3, where you need to make sure that the directory for include files and library files is `gecode-4.3.2`.



Programming search engines

Christian Schulte

This part shows how to program search engines in Gecode.

Chapter 39 (Getting started) presents how to implement simple search engines, where the focus is on understanding the basic operations available on spaces to implement search engines. Chapter 40 (Recomputation) explains recomputation as the most essential technique for efficient search in Gecode. Chapter 41 (An example engine) puts all techniques for search and recomputation from Chapter 39 and Chapter 40 together and presents a realistic search engine.

39 Getting started

This chapters presents how to implement simple search engines. The focus is on understanding the basic operations available on spaces to implement search engines. None of the engines presented here is realistic as they do not use recomputation. The full picture is developed in Chapter 40 and Chapter 41.

Overview. Section 39.1 sets the stage by explaining space operations for programming search engines. A depth-first search engine that makes the simplifying assumption that all choices explored during search are binary is shown in Section 39.2. The next section, Section 39.3, shows depth-first search for choices with an arbitrary number of alternatives. How best solution search can be programmed from spaces is exemplified by a simple branch-and-bound search engine in Section 39.4.

39.1 Space-based search

Search engines compute with spaces: a space implements a constraint model and exploration of its search space is implemented by operation on spaces. The operations on spaces include: computing the status of a space by the `status()` function, creating a clone of a space by the `clone()` function, and committing to an alternative of a choice by the `commit()` function. To commit to an alternative, a space provides the function `choice()` that returns a choice defining how the space can be committed to one of its alternatives. Another operation required to program exploration is the function `alternatives()` defined by a choice that returns the number of alternatives of a choice.

Spaces implement also a `constrain()` function for best solution search. Its discussion is postponed to Section 39.4.

This section reviews the above operations from the perspective of a search engine, the perspective how branchers are controlled by these operations is detailed in Section 31.1. Gecode's architecture for search is designed such that a search engine does not need to know *which* problem is being solved by a search engine: any problem implemented with spaces can be solved by a search engine, and different search engines can be used for solving the same problem. The basic idea of this factorization is due to [44].

Note that here and in the following, spaces and choices are always assumed to be *pointers* to the respective objects. Pointers are necessary as search engines dynamically create and delete spaces and choices.

Status computation. A search engine needs to decide how to proceed during search by computing the *status* of a space by invoking its `status()` function. The `status()` function performs constraint propagation (see Section 22.1) followed by determining the next brancher for branching, if possible (see Section 31.1). Depending on the result of constraint propagation and brancher selection, the `status()` function returns one of the following values of the type `SpaceStatus` (see Programming search engines):

- `SS_FAILED`: the space is *failed*. The search engine needs to backtrack and revisit other spaces encountered during exploration.

  An important responsibility of a search engine is to perform resource management for spaces. In the case of failure, the typical action is to delete the failed space.

- `SS_SOLVED`: the space is *solved*. Hence the search engine has found a solution and typically returns the solution.

  For most engines, the responsibility for deleting a solution lies with the user of a search engine.

  Following the discussion in Section 31.1, calling the `choice()` function of a solved space performs garbage collection for branchers that are not any longer needed. Section 39.2 shows an example search engine that performs garbage collection on solved spaces.

- `SS_BRANCH`: the space requires branching for search to proceed.

  The first step in branching is to compute a choice by calling the `choice()` function of a space. The returned choice can be used for committing to alternatives of a space. In particular, a choice returned by the `choice()` function provides a function `alternatives()` that returns how many alternatives the choice has.

  The pointer to the choice that is returned by the `choice()` function of a space `s` is **`const`**. That is, the following code:

  ```
  const Choice* ch = s->choice();
  ```

  gets a **`const`** pointer to a choice (the choice cannot be modified). Note that it is the obligation of the search engine to eventually delete the choice by

  ```
  delete ch;
  ```

Cloning spaces. A central requirement for a search engine is that it can return to a previous state: as spaces constitute the nodes of the search tree, a previous state is nothing but a space again. Returning to a previous space might be necessary because an alternative suggested by a branching did not lead to a solution, or, even if a solution has been found, more solutions might be requested.

As propagation and branching modify spaces, provisions must be taken that search can actually return to the clone of a previous space. This is provided by the `clone()` function of a space: it returns a clone of a space. This clone can be stored by a search engine such that the engine can return to a previous state. Spaces that are clones of each other are *equivalent*: space operations will have exactly the same effect on equivalent spaces.

The `clone()` function of a space can only be called on a space that is stable and not failed (that is, the `status()` function on a space must return `SS_SOLVED` or `SS_BRANCH`). Otherwise, Gecode throws an exception of type `SpaceNotStable` if the space is not stable and of type `SpaceFailed` if the space is failed.

Committing to alternatives. Given a space `s` and a choice `ch` (assumed to be a **`const`** pointer), the space `s` can be committed to the `i`-th alternative by calling the `commit()` function of a space as follows:

```
s->commit(*ch,i);
```

The choice `ch` must be *compatible* with the space `s`. Before defining when a choice is compatible with a space, let us look at two examples.

Suppose a search engine has invoked `status()` on a space `s` which returned `SS_BRANCH`. The next step is to obtain a choice `ch` for `s` and a clone `c` of `s` by:

```
const Choice* ch = s->choice();
Space* c = s->clone();
```

Further assume that the choice is binary (that is, `ch->alternatives()` returns `2`). A search engine can explore both alternatives (typically, the search engine performs the `commit()` for the second[1] alternative much later) by:

```
s->commit(*ch,0);
c->commit(*ch,1);
```

That is, a choice `ch` is compatible with the space `s` from which it has been computed and with the clone `c` of `s`.

Tip 39.1 (Printing information about alternatives.)**.** Sometimes it might be helpful to print what the `commit()` function does. For this reason, a space provides a `print()` function that compared to `commit()` takes an output stream of type `std::ostream&` as additional argument.

For example, the following

```
s->print(*ch,0,std::out);
std::out << std::endl;
c->print(*ch,1,std::out);
std::out << std::endl;
```

[1]Even though the alternatives are numbered starting from `0` we refer to the alternative with number `0` as the first alternative and the alternative with number `1` as the second alternative.

prints information about what the `commit()` function in the above example actually does. ◀

A search engine for best solution search performs slightly different operations. Let us follow an example scenario. First, the search engine starts exploring the first alternative by:

```
s->commit(*ch,0);
```

Then search continues with `s`. Let us assume that the search engine finds a better solution when continuing search from `s`. Hence, the search engine adds additional constraints to the clone `c` to make sure that exploration from `c` yields a better solution (the constraints are added by calling the `constrain()` function of a space, see Section 39.4). And only then the search engine commits the clone `c` to the second alternative by:

```
c->commit(*ch,1);
```

That is, a choice `ch` is also compatible with the clone `c` of `s`, even though additional constraints have been added to `c` after it had been created by cloning.

In fact, the relation that a choice is compatible with a space is quite liberal. The full notion of compatibility is needed for recomputation and is discussed in Section 40.2.1.

Parallel search. Gecode's kernel is constructed such it does not need to know anything about parallelism: it performs no synchronization[2] in order to keep propagation and search simple and efficient.

By default, a space or clones of a space can only be used by a single thread. However, one can request clones that can be used by different threads as they are created independently. This is achieved by providing **`false`** as an additional argument to the `clone()` member function as follows:

```
Space* c = s->clone(false);
```

The additional argument requests that a non-shared clone is created (in fact, it requests that non-shared copies of data structures used by the space `s` are created, see Section 30.3 for details). The spaces `s` and `c` can now be used by different threads in parallel.

Statistics support. The three main space operations (`status()`, `clone()`, and `commit()`) provide support for execution statistics. For example, statistics from the execution of `status()` on a space `s` can be collected in the object `stat` by:

```
StatusStatistics stat;
s->status(stat);
```

The classes for the statistics correspond to the space operations:

[2]That is a slight over-simplification. Gecode's kernel performs just enough synchronization to maintain global information such as AFC (see Section 8.5.2).

```
status()    StatusStatistics
clone()     CloneStatistics
commit()    CommitStatistics
```

Statistics information is collected by accumulation. That is, for spaces `s1` and `s2`, the following:

```
StatusStatistics stat;
s1->status(stat);
s2->status(stat);
```

collects the combined statistics of performing `status()` on `s1` and `s2`.

The statistics classes also implement addition operators. The following is equivalent to the previous example:

```
StatusStatistics stat;
{
  StatusStatistics a, b;
  s1->status(a);
  s2->status(b);
  stat = a + b;
}
```

which is also equivalent to:

```
StatusStatistics stat;
{
  StatusStatistics a, b;
  s1->status(a); stat += a;
  s2->status(b); stat += b;
}
```

39.2 Binary depth-first search

This section shows a simple search engine that performs left-most depth-first search. It makes the additional simplification that all choices are binary, the general case is discussed in Section 39.3.

Figure 39.1 shows the definition of the function `dfs()` that implements the search engine. It takes a space as input and returns a space as a solution or `NULL` if no solution exists. The resource policy it implements is that it takes responsibility for deleting the space `s` with which `dfs()` is called initially. The solution it returns must eventually be deleted by the caller of `dfs()` (if the initial space happens to be a solution, the engine does not delete it). The search engine starts by executing the `status()` function on `s` and hence triggers propagation and possibly brancher selection.

In this chapter and in Chapter 40 we use recursive functions to implement exploration during search. This is rather inefficient with respect to both runtime and space in C++. A more realistic implementation uses an explicit stack, for an example see Chapter 41.

DFS BINARY ≡ [DOWNLOAD]

```
...
Space* dfs(Space* s) {
  switch (s->status()) {
  case SS_FAILED:
    ▶ FAILED
  case SS_SOLVED:
    ▶ SOLVED
  case SS_BRANCH:
    {
      ▶ PREPARE FOR BRANCHING
      ▶ FIRST ALTERNATIVE
      ▶ SECOND ALTERNATIVE
    }
  }
}
```

Figure 39.1: Depth-first search for binary choices

Failure and solutions. In case the space `s` is failed, the search engine deletes the space and returns `NULL` as specified:

FAILED ≡

```
delete s; return NULL;
```

If the space `s` is solved, the search engine triggers garbage collection of remaining branchers as mentioned in Section 39.1 and returns the solution:

SOLVED ≡

```
(void) s->choice(); return s;
```

Branching. Following the discussion in Section 39.1, before the search engine can start committing to alternatives and perform recursive search, it needs to compute a choice for committing and a clone for backtracking:

PREPARE FOR BRANCHING ≡

```
const Choice* ch = s->choice();
Space* c = s->clone();
```

The search engine tries the first alternative by committing the space `s` to it and continues search recursively:

```
FIRST ALTERNATIVE ≡
  s->commit(*ch,0);
  if (Space* t = dfs(s)) {
    delete ch; delete c;
    return t;
  }
```

If the recursive call to `dfs()` returns a solution (that is, `t` is different from `NULL` and hence the condition of the **`if`** statement is **`true`**) the engine deletes both choice and clone and returns the solution `t`.

Saving memory. It is *absolutely essential* that the search engine uses the original space `s` for further exploration and stores the clone `c` for backtracking. Exchanging the roles of `s` and `c` by:

```
  c->commit(*ch,0);
  if (Space* t = dfs(c)) {
    delete ch; delete s;
    return t;
  }
```

would also find the same solution. However, this search engine would most likely need more memory. Spaces that already have been used for propagation (such as `s`) typically require more memory than a pristine clone (see also Section 30.2). Hence, any search engine should maintain the invariant that it stores pristine clones for backtracking, but never spaces that have been used for propagation.

If the first alternative did not lead to a solution, search commits the clone `c` to the second alternative, deletes the now unneeded choice, and recursively continues search:

```
SECOND ALTERNATIVE ≡
  c->commit(*ch,1);
  delete ch;
  return dfs(c);
```

39.3 Depth-first search

This section demonstrates how left-most depth-first search with choices having an arbitrary number of alternatives can be implemented. By this, the section presents the general version of the search engine from Section 39.2.

Figure 39.2 outlines the depth-first search engine, where computing the space status and handling failed and solved spaces is the same as in Section 39.2. If the search engine needs to branch, it computes the choice `ch` for branching and the number of alternatives `n`.

Choices can actually have a single alternative, for example for assigning variables (see Section 8.13). This special case should be optimized as in fact no clone needs to be stored for backtracking. Hence:

DFS ≡ [DOWNLOAD]

```
...
Space* dfs(Space* s) {
  switch (s->status()) {
  ...
  case SS_BRANCH:
    {
      const Choice* ch = s->choice();
      unsigned int n = ch->alternatives();
      ▶ SINGLE ALTERNATIVE
      ▶ SEVERAL ALTERNATIVES
    }
    break;
  }
}
```

Figure 39.2: Depth-first search

SINGLE ALTERNATIVE ≡

```
if (n == 1) {
  s->commit(*ch,0);
  delete ch;
  return dfs(s);
}
```

If the choice has more than a single alternative, a clone `c` is created and a loop iterates over all alternatives:

SEVERAL ALTERNATIVES ≡

```
Space* c = s->clone();
for (unsigned int a=0; a<n; a++) {
  ▶ SPACE TO EXPLORE
  ▶ RECURSIVE SEARCH
}
delete ch;
return NULL;
```

If the loop terminates, no solution has been found and hence `NULL` is returned.

When trying the `a`-th alternative, the search engine determines which space `e` to choose to continue exploration:

SPACE TO EXPLORE ≡
```
Space* e;
if (a == 0)
  e = s;
else if (a == n-1)
  e = c;
else
  e = c->clone();
```

The choice of `e` avoids the creation of an unnecessary clone for the last alternative.

After committing the space to explore the `a`-th alternative, search continues recursively. If a solution `t` has been found, it is returned after the search engine deletes the clone (unless it has already been used for the last alternative) and the choice:

RECURSIVE SEARCH ≡
```
e->commit(*ch,a);
if (Space* t = dfs(e)) {
  if (a != n-1) delete c;
  delete ch;
  return t;
}
```

39.4 Branch-and-bound search

This section shows how to program a best solution search engine. It chooses branch-and-bound search as an example where choices are again assumed to binary for simplicity. The non-binary case can be programmed similar to Section 39.3.

Constraining spaces. A space to be used for best solution search must implement a `constrain()` function as discussed in Section 2.5. The key aspect of a best solution search engine is that it must be able to add constraints to a space such that the space can only lead to solutions that are better than a previously found solution.

Assume that a best solution search engine has found a so-far best solution `b` (a space). Then, by

```
s->constrain(*b);
```

the engine can add constraints to the space `s` that guarantee that only solutions that are better than `b` are found by search starting from `s`.

The `Space` class actually already implements a `constrain()` function which does nothing. That is, a space to be used with a best solution search engine must redefine the default `constrain()` function by inheritance.

BAB ≡ [DOWNLOAD]

```
...
void bab(Space* s, unsigned int& n, Space*& b) {
  switch (s->status()) {
  ...
  case SS_SOLVED:
    ▶ SOLVED
    break;
  case SS_BRANCH:
    {
      const Choice* ch = s->choice();
      Space* c = s->clone();
      ▶ REMEMBER NUMBER OF SOLUTIONS
      ▶ EXPLORE FIRST ALTERNATIVE
      ▶ CONSTRAIN CLONE
      ▶ EXPLORE SECOND ALTERNATIVE
      delete ch;
    }
    break;
  }
}

Space* bab(Space* s) {
  unsigned int n = 0; Space* b = NULL;
  bab(s,n,b);
  return b;
}
```

Figure 39.3: Branch-and-bound search

Search engine. The basic structure of the branch-and-bound search engine is shown in Figure 39.3. A user of the search engine calls the function `bab()` taking a single space as argument. The function either returns the best solution or `NULL` if no solution exists.

The function `bab()` that takes three arguments implements the actual exploration. The space `s` is the space that is currently being explored, the unsigned integer `n` counts the number of solutions found so far, and the space `b` is the so-far best solution. Note that both `n` and `b` are passed by reference and hence the variables are shared between all recursive invocations of the search engine. The number of solutions `n` is used for deciding when a space must be constrained to yield better solutions.

The single argument `bab()` function initializes `n` and `b` to capture that no solution has been found yet. After executing the `bab()` search engine, `b` refers to the best solution (or is `NULL`) and is returned after garbage collecting remaining branchers.

Finding a solution. The search engine is constructed such that every solution found is better than the previous. Hence, when a solution is found, the previous so-far best solution is deleted[3] and is updated to the newly found solution. As a new solution is found also the number of solutions `n` is incremented:

```
SOLVED ≡
  n++;
  delete b;
  (void) s->choice(); b = s->clone(); delete s;
```

The search engine first garbage collects branchers (by calling `choice()`) and remembers a pristine clone of the solution found.

Branching. Exploring the first alternative differs considerably from exploring the second alternative of a choice. When exploring the first alternative, it is guaranteed that the current space `s` can only lead to better solutions. If a solution is found by exploring the first alternative (or if several solutions are found), then a constraint must be added to the clone `c` such that only better solutions can be found when continuing exploration with `c` for the second alternative. To detect whether a solution has been found when exploring the first alternative, the search engine remembers the number of solutions `m` before starting to explore the first alternative as follows:

```
REMEMBER NUMBER OF SOLUTIONS ≡
  unsigned int m=n;
```

Exploring the first alternative is as to be expected:

```
EXPLORE FIRST ALTERNATIVE ≡
  s->commit(*ch,0);
  bab(s,n,b);
```

[3] Actually, `b` is `NULL` for the first solution found. However, it is legal in C++ to invoke the **`delete`** operator on a `NULL`-pointer.

Before exploring the second alternative, the engine checks whether new solutions have been found during the exploration of the first alternative. If new solutions have been found, the clone `c` is constrained to yield better solutions:

```
CONSTRAIN CLONE ≡
  if (n > m)
    c->constrain(*b);
```

The second alternative is explored as follows:

```
EXPLORE SECOND ALTERNATIVE ≡
  c->commit(*ch,1);
  bab(c,n,b);
```

Note that execution of the `constrain()` function might constrain some variables and possibly add new propagators (even new variables). Even though `c` might not be any longer an identical clone of `s`, the choice `ch` is still compatible with the space `c` (see Section 39.1).

40 Recomputation

This chapter demonstrates recomputation as the most essential technique for efficient search in Gecode. It is highly recommended to read Section 9.1 before reading this chapter.

Overview. The simplest possible search engine based on recomputation, where all spaces needed for search are recomputed from the root of the search tree, is discussed in Section 40.1. Important invariants for recomputation and how they must be taken into account by search engines using recomputation are discussed in Section 40.2. How best solution search is combined with recomputation is discussed in Section 40.3. The following three sections present important optimizations for search engines using recomputation: Section 40.4 shows how last alternative optimization can avoid `commit()` operations during recomputation; Section 40.5 shows how hybrid recomputation that stores additional spaces can be used to speed up search; Section 40.6 shows adaptive recomputation that helps speeding up search in case of failures.

Important. All sections in this chapter make the simplifying assumption that choices are binary, dealing with non binary choices is similar to Section 39.3. The search engine in Chapter 41 combines all recomputation techniques of this chapter for the general case.

40.1 Full recomputation

This section demonstrates search based on full recomputation. While full recomputation is unrealistic, the search engine presented here helps in understanding the ideas behind recomputation. In particular, the search engine serves as an example for important invariants for recomputation that are discussed in Section 40.2.

Search engine. Figure 40.1 shows the outline for the search engine that implements depth-first search with full recomputation. The function `dfs()` that takes a single space `s` as its argument is called by the user, the function `dfs()` that takes three arguments implements exploration with full recomputation.

The basic idea of the search engine is to always keep a single space `r` as the *root* space of the search tree to be explored. Exploration maintains a current space and a path consisting of edges that defines which node in the search tree is currently being explored. Exploration

DFS USING FULL RECOMPUTATION ≡ [DOWNLOAD]

```
...
► EDGE CLASS

Space* dfs(Space* s, Space* r, Edge* p) {
  switch (s->status()) {
  ...
  case SS_BRANCH:
    {
      ► EXPLORE FIRST ALTERNATIVE
      ► EXPLORE SECOND ALTERNATIVE
    }
  }
}

Space* dfs(Space* s) {
  if (s->status() == SS_FAILED) {
    delete s; return NULL;
  }
  Space* r = s->clone();
  Space* t = dfs(s,r,NULL);
  delete r;
  return t;
}
```

Figure 40.1: Depth-first search using full recomputation

EDGE CLASS ≡
```
class Edge {
protected:
  Edge* p;
  const Choice* ch;
  unsigned int a;
public:
  Edge(Space* s, Edge* e)
    : p(e), ch(s->choice()), a(0) {}
  ▶ NEXT ALTERNATIVE
  ▶ COMMITTING A SPACE
  ▶ RECOMPUTING A SPACE
  ~Edge(void) {
    delete ch;
  }
};
```

Figure 40.2: `Edge` class for depth-first search using full recomputation

is governed by the invariant that the current space can always be recomputed from the path of edges maintained by the search engine.

Initially, when the user calls the `dfs()` function taking a single argument, the root space `r` is computed as a clone of the space `s` passed as argument. As only stable and non-failed spaces can be cloned (see Section 39.1), the `status()` function is used to find out whether propagation on `s` results in failure. If not, the root space `r` is created as a clone of `s` and the search engine `dfs()` is called with `s` as the current space, `r` as the root space, and `NULL` as the current path from the current space to the root space.

Edges for recomputation. Figure 40.2 shows the class `Edge` implementing an edge of a path to be used for recomputation. The class stores a pointer `p` to the predecessor edge, a choice `ch`, and the number of the alternative `a` that corresponds to the edge. Edges are organized from the current node (space) of the search tree upwards to the root: the last edge of a path connects to the root of the search tree and has `NULL` as its predecessor `p`. Note that we restrict our attention in this chapter to binary choices only.

Initialization by the `Edge`'s constructor takes the current space of the search engine `s` and the predecessor edge `e` and initializes the edge with the choice for `s` and the first alternative. Deleting an edge also deletes the stored choice `ch`.

An edge provides a `next()` function that redirects the edge to the next alternative:

NEXT ALTERNATIVE ≡
```
void next(void) {
  a++;
}
```

The `commit()` function of an edge commits a space `s` to the alternative that corresponds to the edge:

COMMITTING A SPACE ≡
```
Space* commit(Space* s) const {
  s->commit(*ch,a); return s;
}
```

The function returns the space just for convenience as can be seen below.

Finally, recomputing a space corresponding to an entire path of edges is implemented by the `recompute()` function. The function takes the root space `r` as argument and is implemented as follows:

RECOMPUTING A SPACE ≡
```
Space* recompute(Space* r) const {
  return commit((p == NULL) ? r->clone() : p->recompute(r));
}
```

First, the function traverses the path of edges upwards until the root of the path is reached. Then it creates a clone of the root space `r` and performs the `commit()` operations for each edge on the path.

Exploring alternatives. The central invariant that the current path of edges `p` must always correspond to the current space is essential for how the search engine implementing full recomputation explores the search tree.

Before exploring the first alternative recursively, a new edge is created for the first alternative, the current space `s` is committed to the first alternative, and exploration continues recursively:

EXPLORE FIRST ALTERNATIVE ≡
```
Edge e(s,p);
if (Space* t = dfs(e.commit(s),r,&e))
  return t;
```

Note that all resource management for handling edges is done automatically: as soon as the edge `e` goes out of scope, it is automatically deleted (and hence also the choice for the edge is deleted).

Again, exploration of the second alternative maintains the central invariant: the edge is redirected to the next alternative and then a space corresponding to the path of edges is recomputed:

EXPLORE SECOND ALTERNATIVE ≡
```
e.next();
return dfs(e.recompute(r),r,&e);
```

Cost of recomputation. It is important to notice the following facts about the cost of recomputation, where we compare the search engine using full recomputation to the search engine without recomputation in Section 39.2:

- While the number of `commit()` operations on spaces drastically increases for full recomputation, the number of `status()` operations executed remains exactly the same.

  However, execution of `status()` during recomputation is more expensive as more constraint propagation needs to be done: full recomputation always starts from the root space where only little constraint propagation has been performed.

- The number of `clone()` operations executed by full recomputation is never larger than the number of `clone()` operations without recomputation.

  Typically, the number of `clone()` operations is much smaller: recomputation is *optimistic* in the sense that a clone is created and recomputation is performed only if a node is required for exploration. A search engine without recomputation is *pessimistic* in that it always creates a clone *before* continuing exploration to be able to backtrack to a space that might be required for further exploration.

  Section 40.5 presents hybrid recomputation as a technique that makes recomputation less optimistic in that it creates more clones before continuing exploration. Section 40.6 presents adaptive recomputation that makes recomputation even less optimistic in cases where it is likely that exploration can benefit from additional clones.

A detailed evaluation of recomputation in Gecode and a comparison to other techniques for implementing search can be found in [39]. An evaluation of different recomputation techniques (although in a different setup) can be found in [43] and [44, Chapter 7].

40.2 Recomputation invariants

This section discusses two important invariants that govern recomputation. The first invariant is that recomputation can only use compatible choices for commit operations: the notion of choice compatibility has been sketched in Section 39.1 and is detailed in Section 40.2.1. The second invariant is concerned with the fact that recomputation is not always deterministic. However, Section 40.2.2 explains why recomputation still works even though it is not deterministic.

40.2.1 Choice compatibility

Section 39.1 introduced the notion that a space `s` is compatible with a choice `ch` (that is, `ch` can be used for a `commit()` operation on `s`). For recomputation, a more general notion of compatibility is needed: during recomputation, a search engine performs `commit()` operations using choices that have been computed earlier on a path in the search tree.

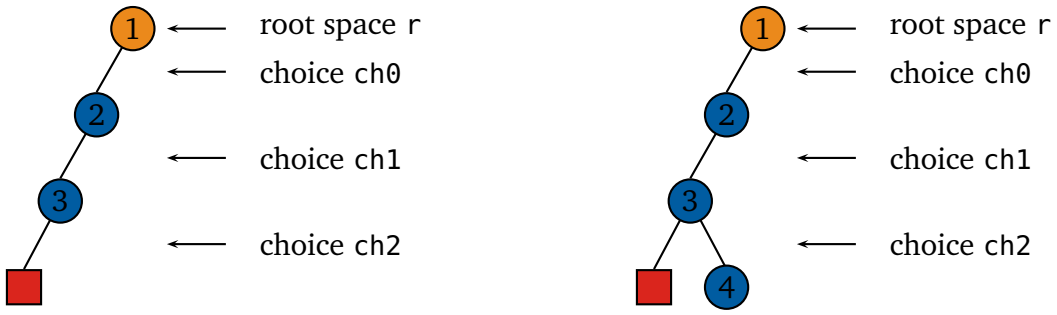


Figure 40.3: Example situations during recomputation

Consider the situation sketched in the left part of Figure 40.3 with a root space `r` and choices `ch0`, `ch1`, and `ch2` as stored on the path of edges for recomputation. That is, all choices have been computed by the `choice()` function from a clone of the root space `r` where in between the calls to `choice()` other operations on the space have been performed. For example, the search engine implementing full recomputation from Section 40.1 has executed the `status()` and `commit()` functions several times in order to compute the choices.

Suppose that `s` is a clone of `r` (computed by `s=r->clone()`). Then all choices are compatible with `s`. Now suppose that `t` is a clone of `s`. Then all choices are still compatible with `t`. Hence, if a choice `ch` has been created for a space `r` all spaces that are related by cloning to `r` are compatible with `ch`. Compatibility continues to hold even if other operations (such as `commit()` operations for other choices, computing the status of a space, or the creation of new variables, propagators, and branchers) are performed on a space that is clone-related. The only exception is the `choice()` function of a space.

Suppose that recomputation proceeds by recomputing the space `s` for node `4` as shown in the right part of Figure 40.3. Assume that `s` requires branching. Hence, a search engine is executing `s->choice()`. After executing the `choice()` function, all previous choices `ch0`, `ch1`, and `ch2` are not any longer compatible with `s`.

Order of `commit()` operations. The choices `ch0`, `ch1`, and `ch2` do not have to be used for `commit()` in the same order in which they have been created. However, it is more efficient to use them in the order `ch0`, `ch1`, and `ch2`. The difference is that if a space has n branchers and the choices correspond to different branchers, then a `commit()` operation uses $O(1)$ time to find the corresponding brancher for a choice if the choices are used in order. If they are not used in order, a `commit()` operation uses $O(n)$ time to find the corresponding brancher. Having said all that, n is typically just one or two.

40.2.2 Recomputation is not deterministic

Assume that a search engine has recorded a path from a node in the search tree and that the space `s` corresponds to that node in the tree. Then, when a space `t` for that node

is recomputed, the two spaces `s` and `t` might actually differ: the amount of propagation performed by `s` and `t` can be different.

The difference in propagation is due to the fact that Gecode supports propagators that are weakly monotonic but not necessarily monotonic. In summary, a weakly monotonic propagator can perform more or less propagation (that is, prune more values from variables domains or prune fewer values from variable domains) but is still correct and complete, see also Section 22.9.

As weakly monotonic constraint propagation is still correct and complete, also recomputation is correct in the following sense: All solutions that are found by search starting from space `s` are also found by search starting from space `t`. However, search might find the solutions in different order when starting from either `s` or `t`.

Consider the special case that `s` is failed. The recomputed space `t` might not necessarily be failed. That is, `s` performed more constraint propagation than `t`. However, additional search from `t` will never find any solution. The same is also true with the roles of `s` and `t` exchanged: even though `s` is not failed, the recomputed space `t` can be failed.

For search engines using recomputation this typically does not pose any problems. In most cases, a search engine does not attempt to recompute a space that it assumes to be non-failed. Consider the depth-first search engine using full recomputation from Section 40.1. There, the spaces that are recomputed correspond to nodes in the search tree not yet explored. In other words, when recomputing spaces that have been explored previously, a search engine cannot make the assumption that the space is not failed just because the space explored previously has not been failed.

An exception is adaptive recomputation to be discussed in Section 40.6 as an optimization for recomputation. Adaptive recomputation recomputes previously explored spaces to speed up further search.

For a detailed discussion of weakly monotonic propagation together with the consequences for search including recomputation, please consult [50].

40.3 Branch-and-bound search

The first idea to combine recomputation with best solution search is to take the engine for best solution search without recomputation (see Section 39.4) and add recomputation to it along the lines of Section 40.1. A search engine implementing this approach would work roughly as follows: recompute a space and, if needed, add the constraints to the space such that the space can only lead to a better solution.

A tighter integration of recomputation and best solution search is in fact much better: instead of adding the constraints for a better solution to the space that is recomputed, add it to the space from which recomputation starts (with full recomputation: the root space). The advantage is that if adding the constraints to the space from which recomputation starts already leads to failure, the entire subtree starting from that space can be discarded (with full recomputation: search is done).

Figure 40.4 sketches branch-and-bound best solution search using full recomputation.

BAB USING FULL RECOMPUTATION ≡ [DOWNLOAD]

```
...
void bab(Space* s, Space*& r, Space*& b, Edge* p) {
  switch (s->status()) {
  ...
  case SS_SOLVED:
    ▶ SOLVED
  case SS_BRANCH:
    {
      ...
      ▶ EXPLORE SECOND ALTERNATIVE
    }
  }
}

Space* bab(Space* s) {
  ...
  Space* r = s->clone();
  Space* b = NULL;
  bab(s,r,b,NULL);
  ...
}
```

Figure 40.4: Branch-and-bound search using full recomputation

The search engine takes the current space `s`, the root space `r`, the so-far best solution `b`, and the predecessor edge `p` as arguments. Note that both the root space `r` and the so-far best solution `b` are passed by reference as they change during exploration.

When the search engine finds a new solution `s`, it replaces the so-far best solution `b` by `s` and updates the root space by adding the constraints that `r` must yield better solutions than `b` as follows:

SOLVED ≡
```
delete b;
(void) s->choice(); b = s->clone(); delete s;
r->constrain(*b);
if (r->status() == SS_FAILED) {
  delete r; r = NULL;
} else {
  Space* c = r->clone(); delete r; r = c;
}
break;
```

The root space then must be checked for failure. If the root space is failed, it is deleted and the pointer to the root space becomes `NULL`. Otherwise, the root space becomes a clone of the newly computed space to save memory (as mentioned earlier, it is better to store a pristine clone of a space than a space on which propagation has been performed).

Exploring the second alternative of a choice checks whether the root space is already failed:

EXPLORE SECOND ALTERNATIVE ≡
```
if (r != NULL)
  bab(e.recompute(r),r,b,&e);
```

40.4 Last alternative optimization

This section presents an important optimization for recomputation that helps to avoid many commit operations during recomputation. Even though the optimization is discussed in the context of full recomputation, it is applicable to all situations in which recomputation is used.

Consider a situation during search as shown in the left part of Figure 40.5. There, the entire left subtree emanating from the root node (colored in orange) has been explored. When exploration continues for the right subtree, each time a node is recomputed, a clone of the root node is made immediately followed by a commit operation for the second alternative. Hence it is much better to compute a new root node for the entire right subtree and perform the corresponding commit operation just once. This optimization is referred to as *last alternative optimization (LAO)* [44, Chapter 7]. Figure 40.5 shows the new root of the search tree after performing LAO. Note that no space needs to be stored for the previous root node as it will never be used again for recomputation.

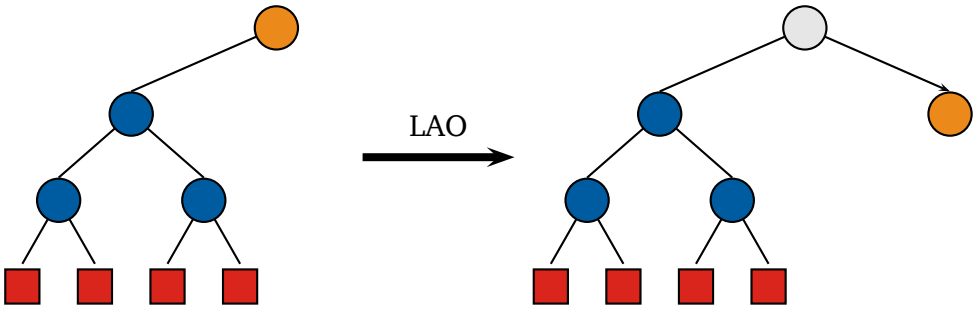


Figure 40.5: Last alternative optimization (LAO)

DFS USING FULL RECOMPUTATION AND LAO ≡ [DOWNLOAD]

```
...
class Edge {
  ...
  ▶ TEST FOR LAST ALTERNATIVE
  ...
};

Space* dfs(Space* s, Space*& r, Edge* p) {
  switch (s->status()) {
  ...
  case SS_BRANCH:
    {
      Edge e(s,p);
      if (Space* t = dfs(e.commit(s),r,&e))
        return t;
      e.next();
      ▶ PERFORM LAO
      return dfs(e.recompute(r),r,&e);
    }
  }
}
...
```

Figure 40.6: Depth-first search using full recomputation and LAO

Figure 40.6 shows a search engine implementing left-most depth-first search using full recomputation and LAO. Only very few aspects have changed compared to the engine for full recomputation without LAO from Section 40.1. An important change is that the root space is now passed by reference: this is necessary as the root space changes during exploration.

The `Edge` class is extended by a function `la()` that tests whether an edge happens to be a last alternative:

```
TEST FOR LAST ALTERNATIVE ≡
  bool la(void) const {
    return (p == NULL) && (a == 1);
  }
```

The actual optimization is performed just before exploring the second alternative:

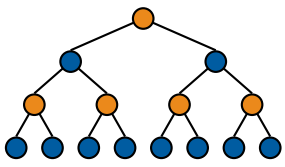
```
PERFORM LAO ≡
  if (e.la()) {
    Space* t = r;
    if (e.commit(t)->status() == SS_FAILED)
      return NULL;
    r = t->clone();
    return dfs(t,r,NULL);
  }
```

The space `t` serves as a temporary reference. After performing the `commit()` operation on `t` for the second alternative, it is tested whether the new root node is already failed. The new root space becomes the clone of `t`: this is important as a clone typically requires less memory than a space on which constraint propagation has been performed (see Section 39.2).

40.5 Hybrid recomputation

Exploration based on copying alone or based on full recomputation is unrealistic. As already described in Section 9.1, Gecode's search engines use hybrid recomputation: they create a clone now and then to limit the amount of recomputation.

This section describes hybrid recomputation where the amount of recomputation is limited by the *commit distance* c_d: during recomputation at most c_d `commit()` operations are executed. Hybrid recomputation with commit distance $c_d = 2$ where orange nodes are nodes that store a clone is sketched to the right.



The additional clones are stored in a field `c` in the `Edge` class shown in Figure 40.7. Initially, the field `c` does not store a clone. The intuition is that the clone `c` (if not `NULL`) is a clone that corresponds to the node to which the edge leads (please remember that edges lead upwards to the root of the search tree).

DFS USING HYBRID RECOMPUTATION ≡ [DOWNLOAD]

```
...
const unsigned int c_d = 5;

class Edge {
protected:
  ...
  Space* c;
public:
  Edge(Space* s, Edge* e)
    : p(e), ch(s->choice()), a(0), c(NULL) {}
  ...
  ▶ CREATE CLONE
  ▶ PERFORM RECOMPUTATION
  ...
};

Space* dfs(Space* s, Edge* p, unsigned int d) {
  switch (s->status()) {
  ...
  case SS_BRANCH:
    {
      ▶ STORE CLONE IF NEEDED
      ▶ EXPLORE FIRST ALTERNATIVE
      ...
    }
  }
}

Space* dfs(Space* s) {
  return dfs(s,NULL,c_d);
}
```

Figure 40.7: Depth-first search using hybrid recomputation

The function `clone()` stores a clone of a space `s` in an edge:

```
CREATE CLONE ≡
  void clone(Space* s) {
    c = s->clone();
  }
```

Recomputation as performed by the `recompute()` function now continues to search the path of edges until an edge that stores a clone is found:

```
PERFORM RECOMPUTATION ≡
  Space* recompute(void) const {
    return commit((c != NULL) ? c->clone() : p->recompute());
  }
```

It must be guaranteed that there is always at least one edge in a path that stores a clone, otherwise recomputation would crash (see below).

The function `dfs()` that implements exploration takes the additional argument `d` (for distance) which defines how many `commit()` operations would be needed to recompute the current space. If `d` reaches the limit `c_d` (for simplicity, `c_d` is a constant as defined in Figure 40.7), a new clone must be stored in the current edge. Hence, the code for branching starts by checking whether a clone must be stored for the current edge `e`:

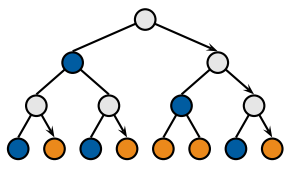
```
STORE CLONE IF NEEDED ≡
  Edge e(s,p);
  if (d >= c_d) {
    e.clone(s); d=0;
  }
```

The initial call to the function `dfs()` that implements exploration takes `c_d` as value for `d`. By this, it is guaranteed that the first edge stores a clone (that is, a clone that corresponds to the root node of the search tree).

Exploring the alternatives is as before, the only change is that the incremented distance `d+1` is passed as additional argument:

```
EXPLORE FIRST ALTERNATIVE ≡
  if (Space* t = dfs(e.commit(s),&e,d+1))
    return t;
```

LAO as described for full recomputation (Section 40.4) can be added analogously to hybrid recomputation. Therefore we do not present hybrid recomputation with LAO. However, how LAO performs with hybrid recomputation is sketched to the right: gray nodes correspond to nodes where a clone had been stored, while orange nodes correspond to nodes where a clone has been moved due to LAO.



40.6 Adaptive recomputation

Consider the situation when a search engine using recomputation encounters a failed space during exploration. Then it is quite likely that as exploration continues, more failed spaces are found during search. This is due to the fact that some decision made during branching (that is, some alternative) has lead to failure. It is quite likely that the wrong decision from which search must recover is somewhere on the path to the root node.

That means that search now must explore the entire subtree starting from the wrong decision. In this situation it would be advantageous if additional clones were available for exploring the entire subtree. With other words, after encountering failure, search should become more pessimistic by investing into the creation of additional clones to speed up further exploration.

Adaptive recomputation [44, Chapter 7] optimizes recomputation in the case of failures: an additional clone is created during recomputation. The idea is that on a path of length n without any clone, an additional clone is placed in the middle of that path. This additional clone then speeds up further recomputation (which is likely to occur as has been argued above).

Adaptive recomputation is controlled by a parameter called *adaptive distance* a_d: only if $n \geq a_d$ an additional clone is created. This avoids creating an excessive amount of clones.

Adaptive recomputation is sketched in Figure 40.8. The only change compared to hybrid recomputation (see Section 40.5) is the implementation of `recompute()`. The argument `n` is the length of the path to the next clone. When a clone is found, `d` is set to $\lfloor \frac{n}{2} \rfloor$ provided that $n \geq$ `a_d`. Otherwise `d` is set to `n` (its old value) which also prevents that an additional clone is created. For the edge that is `d` `commit()` operations away from the current space, a new clone is stored on the path, provided that the space `s` is not failed.

The space `s` can be failed as has been discussed in Section 40.2.2. The search engine here takes a rather simplistic approach to this situation: it just does not store a clone. A real-life engine would take more benefit from the information that a space on the path is actually failed: it would immediately discard the entire path below the failed space, see Chapter 41.

Another optimization that a real-life search engine would employ is to not place a clone in a position where it could be moved by last alternative optimization, see again Chapter 41.

DFS USING ADAPTIVE RECOMPUTATION ≡ [DOWNLOAD]

```
  ...
  const unsigned int c_d = 5;
  const unsigned int a_d = 2;

  class Edge {
  protected:
    ...
    Space* recompute(unsigned int n, unsigned int& d) {
      if (c != NULL) {
        d = (n >= a_d) ? n/2 : n;
        return commit(c->clone());
      } else {
        Space* s = p->recompute(n+1,d);
        if ((d == n) && (s->status() != SS_FAILED))
          clone(s);
        return commit(s);
      }
    }
    Space* recompute(unsigned int& d) {
      return recompute(1,d);
    }
    ...
  };
  ...
```

Figure 40.8: Depth-first search using adaptive recomputation

41 An example engine

This chapter puts all techniques for search and recomputation from Chapter 39 and Chapter 40 together. It presents a realistic search engine that can be used to search for several solutions.

Overview. Section 41.1 sketches the design of the example depth-first search engine to be used in this chapter. How the engine is implemented is shown in Section 41.2. Section 41.3 details how exploration is implemented, whereas Section 41.4 details how recomputation is implemented for the search engine.

41.1 Engine design

The example engine to be developed in this chapter implements depth-first search using hybrid and adaptive recomputation with full last alternative optimization. It provides an interface similar to the interface of Gecode's pre-defined search engines: it is initialized with a space (even though the search engine presented here does not make a clone for simplicity) and provides a `next()` function that returns a space for the next solution or returns `NULL` if there are no more solutions.

The outline of the search engine is shown in Figure 41.1. To keep things simple, the values for the commit distance `c_d` and adaptive distance `a_d` are constants. Furthermore, the search engine uses an array of fixed size to implement the path of edges for recomputation. In case the size of the array is exceeded during exploration, an exception of type `StackOverflow` is thrown. A real-life engine would of course use a dynamic data structure such as a C++ vector.

41.2 Engine implementation

Figure 41.2 shows the class `Engine` implementing depth-first search. The engine maintains a path `p` of edges for recomputation (to be explained below), a current space `s`, and a distance `d`. The current space `s` can be `NULL`. If the current space `s` is not `NULL`, then the path `p` corresponds to `s`. If the space is `NULL`, the search engine uses recomputation to compute the next space needed for exploration.

DFS ENGINE ☰ [DOWNLOAD]

```
#include <gecode/kernel.hh>

using namespace Gecode;

const unsigned int c_d = 5;
const unsigned int a_d = 2;

const unsigned int n_stack = 1024;

class StackOverflow : public Exception {
public:
  StackOverflow(const char* l)
    : Exception(l,"Stack overflow") {}
};
```

▶ PATH

▶ SEARCH ENGINE

Figure 41.1: Depth-first search engine

SEARCH ENGINE ≡

```
class Engine {
protected:
  Path p;
  Space* s;
  unsigned int d;
public:
  Engine(Space* r) : s(r), d(c_d) {}
  Space* next(void) {
    do {
      while (s != NULL)
        ▶ EXPLORATION
      while ((s == NULL) && p.next())
        s = p.recompute(d);
    } while (s != NULL);
    return NULL;
  }
  ~Engine(void) {
    delete s;
  }
};
```

Figure 41.2: Implementation of depth-first search engine

The distance `d` describes the number of commit operations needed for recomputation. It is initialized to `c_d` to force the immediate creation of a clone on the path (analogous to the search engine in Section 40.5).

Exploration mode. The engine operates in two modes: *exploration mode* and *recomputation mode*. It operates in exploration mode while the current space `s` is not `NULL`. Exploration mode continues until the current space `s` becomes failed or solved. In both cases, the current space is set to `NULL`.

If the space `s` becomes solved, it is returned as a solution by the `next()` function. If the `next()` function is called again, then the situation is exactly the same as for failure: the current space is `NULL` and the engine switches to recomputation mode. Exploration is detailed in Section 41.3.

Recomputation mode. In recomputation mode, the engine tries to recompute the current space. The `next()` function of the path `p` moves the path to the next alternative. If there is a next alternative (the search space has not yet been completely explored), the `next()` function of a path returns **`true`**. The `recompute()` function tries to recompute the current space `s` according to the path `p`. Due to adaptive recomputation, the `recompute()` function might update the distance `d` and might actually fail to recompute a space that corresponds to the current path (in which case it returns `NULL`). Recomputation is detailed in Section 41.4.

If no more alternatives are to be tried (that is, the `next()` function of the path `p` has returned **`false`**), the `next()` function of the search engine terminates by returning `NULL`.

41.3 Exploration

The search engine continues in exploration mode while the current space `s` is different from `NULL` and executes the code shown in Figure 41.3. In case the current space `s` is failed, it is discarded, `s` is set to `NULL`, and the engine switches to recomputation mode. The same is true if the engine finds a solution, however it garbage collects branchers on the solution found and returns it. With another invocation of the `next()` function, the engine will operate in recomputation mode.

Edge implementation. Figure 41.4 shows how an edge is implemented. The implementation is analogous to the edge classes used in Chapter 40. Edges support choices with an arbitrary number of alternatives, the test `la()` whether an edge is at its last alternative takes the number of alternatives of the choice into account.

Rather than having a default constructor and a destructor, edges use the `init()` and `reset()` functions. This is more convenient as edges are maintained in an array implementing a stack, see below for details.

EXPLORATION ≡
```
switch (s->status()) {
case SS_FAILED:
  delete s; s = NULL;
  break;
case SS_SOLVED:
  {
    Space* t = s; s = NULL;
    (void) t->choice();
    return t;
  }
case SS_BRANCH:
  if (d >= c_d) {
    p.push(s,s->clone()); d=1;
  } else {
    p.push(s,NULL); d++;
  }
}
```

Figure 41.3: Implementation of exploration

Path implementation. Figure 41.5 shows how a path of edges is implemented. The array `e` stores the edges of the path. The array implements a stack of edges and the unsigned integer `n` defines the number of edges that are currently on the stack. The edge at position `n-1` of the array of edges `e` corresponds to the top of the stack.

Pushing edges on the path. During exploration, the engine pushes new edges on the path `p` as shown in Figure 41.3. If the distance `d` has reached the commit distance `c_d`, the engine pushes an edge to the path that has an additional clone and resets the distance `d` accordingly.

Pushing an edge checks for stack overflow and initializes the field of the edge array that corresponds to the top of stack as follows:

PUSH EDGE ≡
```
void push(Space* s, Space* c) {
  if (n == n_stack)
    throw StackOverflow("Path::push");
  e[n].init(s,c); e[n].commit(s);
  n++;
}
```

Note that the push operation also performs the `commit()` operation on the current space `s` that corresponds to the edge just pushed onto the path.

EDGE ☰

```
class Edge {
protected:
  const Choice* ch;
  unsigned int a;
  Space* c;
public:
  void init(Space* s, Space* c0) {
    ch = s->choice(); a = 0; c = c0;
  }
  Space* clone(void) const {
    return c;
  }
  void clone(Space* s) {
    c = s->clone();
  }
  void next(void) {
    a++;
  }
  bool la(void) const {
    return a+1 == ch->alternatives();
  }
  void commit(Space* s) {
    s->commit(*ch,a);
  }
  ▶ PERFORM LAO
  void reset(void) {
    delete ch; ch=NULL; delete c; c=NULL;
  }
};
```

Figure 41.4: Implementation of edges

```
PATH ≡
  class Path {
  protected:
    ▶ EDGE
    Edge e[n_stack];
    unsigned int n;
  public:
    Path(void) : n(0) {}
    ▶ PUSH EDGE
    ▶ MOVE TO NEXT ALTERNATIVE
    ▶ PERFORM RECOMPUTATION
  };
```

Figure 41.5: Implementation of path of edges

41.4 Recomputation

In recomputation mode, the engine uses operations to move the engine to the next alternative and to perform recomputation of a space corresponding to the current path.

Move to next alternative. Moving to a next alternative discards all edges from the path that are already at their last alternative (that is, the function `la()` returns true). If the engine finds an edge with remaining alternatives, it moves the edge to the next alternative. If no edges are left, the function `next()` returns **`false`** as follows:

```
MOVE TO NEXT ALTERNATIVE ≡
  bool next(void) {
    while (n > 0)
      if (e[n-1].la()) {
        e[--n].reset();
      } else {
        e[n-1].next(); return true;
      }
    return false;
  }
```

Perform recomputation. The `recompute()` function shown in Figure 41.6 performs recomputation. LAO and adaptive recomputation are orthogonal optimizations and are discussed later. First, `i` is initialized such that it points to the closest edge on the path that has a clone. Then, `s` is initialized to a clone of the edge's clone and the distance `d` is updated accordingly. Finally, all `commit()` operations between `i` and `n` are performed on `s`.

```
PERFORM RECOMPUTATION ≡
  Space* recompute(unsigned int& d) {
    ▶ PERFORM LAO
    unsigned int i = n-1;
    for (; e[i].clone() == NULL; i--) {}
    Space* s = e[i].clone()->clone();
    d = n - i;
    ▶ PERFORM ADAPTIVE RECOMPUTATION
    for (; i < n; i++)
      e[i].commit(s);
    return s;
  }
```

Figure 41.6: Implementation of recomputation

Last alternative optimization. Before actually starting recomputation, the `recompute()` function checks whether it can perform LAO. It checks whether the last edge of the path can perform LAO (in which case `t` is different from NULL) as follows:

```
PERFORM LAO ≡
  if (Space* t = e[n-1].lao()) {
    e[--n].reset();
    d = c_d;
    return t;
  }
```

The edge is removed from the path and the distance `d` is set to `c_d` to force the immediate creation of a new clone when the engine continues in exploration mode.

LAO for an edge checks whether the edge is at the latest alternative and whether the edge stores a clone:

```
PERFORM LAO ≡
  Space* lao(void) {
    if (!la() || (c == NULL))
      return NULL;
    Space* t = c; c = NULL;
    commit(t);
    return t;
  }
```

If this is the case, the clone from the edge is removed and is committed to the last alternative.

Adaptive recomputation. If the current distance `d` reaches the adaptive distance `a_d`, recomputation tries to perform adaptive recomputation as follows:

```
PERFORM ADAPTIVE RECOMPUTATION ≡
  if (d >= a_d) {
    unsigned int m = i + d/2;
    for (; i < m; i++)
      e[i].commit(s);
    ▶ SKIP OVER LAST ALTERNATIVES
    ▶ CREATE ADDITIONAL CLONE
  }
```

The value of `m` is the middle between the position of the clone `i` and the position of the last edge on the path. The position `m` is a candidate position where the additional clone might be stored. All commit operations for edges between the clone and edge at position `m` are executed.

It is entirely pointless to store the additional clone at an edge that is already at its last alternative (this is what LAO is all about). Hence, adaptive recomputation skips over all edges that are already at their last alternative as follows:

```
SKIP OVER LAST ALTERNATIVES ≡
  for (; (i < n) && e[i].la(); i++)
    e[i].commit(s);
```

An additional clone for an edge is only created if the edge is not already the topmost edge of the path:

```
CREATE ADDITIONAL CLONE ≡
  if (i < n-1) {
    ▶ PERFORM PROPAGATION
    e[i].clone(s);
    d = n-i;
  }
```

After storing the clone, the distance `d` is adapted accordingly.

Before being able to create a clone, adaptive recomputation performs constraint propagation by executing the `status()` function of the space `s` as follows:

```
PERFORM PROPAGATION ≡
  if (s->status() == SS_FAILED) {
    delete s;
    for (; i < n; n--) {
      e[n-1].reset(); d--;
    }
    return NULL;
  }
```

If constraint propagation leads to a failed space (see Section 40.2.2), all edges below the failed space are discarded and recomputation returns `NULL` to signal that recomputation did not succeed in recomputing a space for the current path.

Bibliography

[1] Anbulagan and Adi Botea. Crossword puzzles as a constraint problem. In Peter J. Stuckey, editor, *Fourteenth International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 550–554, Sydney, Australia, September 2008. Springer-Verlag.

[2] Nicolas Barnier and Pascal Brisset. Solving Kirkman's schoolgirl problem in a few seconds. *Constraints*, 10(1):7–21, 2005.

[3] Adam Beacham, Xinguang Chen, Jonathan Sillito, and Peter van Beek. Constraint programming lessons learned from crossword puzzles. In Eleni Stroulia and Stan Matwin, editors, *Canadian Conference on AI*, volume 2056 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2001.

[4] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Eigth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 63–79, Ithaca, NY, USA, September 2002. Springer-Verlag.

[5] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[6] Christian Bessiere, Emanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the NValue constraint. *Constraints*, 11(4):271–293, 2006.

[7] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.

[8] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[9] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *Sixteenth European Conference on Artificial Intelligence*, pages 146–150, Valencia, Spain, August 2004. IOS Press.

[10] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16:575–577, September 1973.

[11] F[rédéric] Cazals and C[hinmay] Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407:564–568, November 2008.

[12] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 49–58. Springer Verlag, 2006.

[13] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 240–255, Paphos, Cyprus, November 2001. Springer-Verlag.

[14] Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half reification and flattening. In Jimmy Lee, editor, *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming*, volume 6876 of *Lecture Notes in Computer Science*, pages 286–301, Perugia, Italy, September 2011. Springer-Verlag.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman And Company, New York, NY, USA, 1979.

[16] Ian Gent, editor. *Fifteenth International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, Lisbon, Portugal, September 2009. Springer-Verlag.

[17] Ian P. Gent and Toby Walsh. From approximate to optimal solutions: Constructing pruning and propagation rules. In *IJCAI*, pages 1396–1401, 1997.

[18] Carmen Gervet. *Finite Set Constraints*. PhD thesis, L'Université de Franche-Comté, Besançon, France, 1995.

[19] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search lessons learned from crossword puzzles. In *AAAI*, pages 210–215, 1990.

[20] Stefano Gualandi and Michele Lombardi. A simple and effective decomposition for the multidimensional binpacking constraint. In Christian Schulte, editor, *Proceedings of the Nineteenth International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 356–364, Uppsala, Sweden, September 2013. Springer-Verlag.

[21] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 390–396, Pittsburgh, PA, USA, 2005. AAAI Press.

[22] Mikael Z. Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints BPPC'08*, 2008. [download].

[23] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessière, editor, *Thirteenth International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 409–422, Providence, RI, USA, September 2007. Springer-Verlag. [download].

[24] Yat Chiu Law and Jimmy H.M. Lee. Global constraints for integer and set value precedence. In Wallace [66], pages 362–376.

[25] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):147–167, 2007.

[26] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison Wesley, fifth edition, 2013.

[27] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In Georg Gottlob and Toby Walsh, editors, *Eighteenth International Joint Conference on Artificial Intelligence*, pages 245–250, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[28] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[29] Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

[30] Christopher Mears, Maria Garcia de la Banda, Bart Demoen, and Mark Wallace. Lightweight dynamic symmetry breaking. In *Eighth International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'08*, 2008. [download].

[31] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Rina Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319, Singapore, September 2000. Springer-Verlag.

[32] Laurent Michel and Pascal Van Hentenryck. A decomposition-based implementation of search strategies. *Transactions of Computational Logic*, 5(2):351–383, 2004.

[33] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 228–243, Nantes, France, May 2012. Springer.

[34] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliff, NJ, USA, 1966.

[35] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pages 530–535, Las Vegas, NV, USA, 2001. ACM.

[36] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Wallace [66], pages 482–495.

[37] Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In Francesca Rossi, editor, *Ninth International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 600–614, Kinsale, Ireland, September 2003. Springer-Verlag.

[38] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.

[39] Raphael M. Reischuk, Christian Schulte, Peter J. Stuckey, and Guido Tack. Maintaining state in propagation solvers. In Gent [16], pages 692–706. [download].

[40] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[41] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[42] Armin Scholl, Robert Klein, and Christian Jürgens. BISON: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627–645, 1997.

[43] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.

[44] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[45] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Rossi et al. [40], chapter 14, pages 495–526.

[46] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Wallace [66], pages 619–633.

[47] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space? *Transactions on Programming Languages and Systems*, 27(3):388–425, May 2005.

[48] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008. [download].

[49] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 2006. [download].

[50] Christian Schulte and Guido Tack. Weakly monotonic propagators. In Gent [16], pages 723–730. [download].

[51] Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints*, 18(1):75–107, January 2013.

[52] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Forth International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431, Pisa, Italy, 1998. Springer-Verlag.

[53] Paul Shaw. A constraint for Bin Packing. In Wallace [66], pages 648–662.

[54] Helmut Simonis. Kakuro as a constraint problem. In *Workshop on Modeling and Reformulation*, September 2008.

[55] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the golomb ruler problem. School of Computer Studies Research Report 1999.12, University of Leeds, Leeds, UK, June 1999.

[56] Guido Tack. *Constraint Propagation - Models, Techniques, Implementation*. Doctoral dissertation, Saarland University, Germany, 2009. [download].

[57] Nobuhisa Ueda and Tadaaki Nagao. NP-completeness results for nonogram via parsimonious reductions. Technical Report TR96-008, Dept. of Computer Science, Tokyo Institute of Technology, 1996.

[58] Peter van Beek. Backtracking search algorithms. In Rossi et al. [40], chapter 4, pages 85–134.

[59] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. The MIT Press, Cambridge, MA, USA, 1989.

[60] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, MA, USA, 1999.

[61] Willem Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. New filtering algorithms for combinations of among constraints. *Constraints*, 14(2):273–292, 2009.

[62] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In Gent [16], pages 802–816.

[63] Petr Vilím. Max energy filtering algorithm for discrete cumulative resources. In Willem Jan van Hoeve and John N. Hooker, editors, *Sixth International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, pages 294–308, Pittsburgh, PA, USA, May 2009. Springer-Verlag.

[64] Petr Vilím. *Global Constraints in Scheduling*. PhD thesis, Charles University, Prague, Czech Republic, 2007.

[65] H.C. von Warnsdorff. Des Rösselsprungs einfachste und allgemeinste Lösung, 1823. Schmalkalden, Germany.

[66] Mark Wallace, editor. *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*. Springer-Verlag, Toronto, Canada, September 2004.

[67] Mark Allen Weiss. *C++ for Java Programmers*. Pearson Prentice Hall, 2004.

Changelog

- Released for Gecode 4.3.2 (2014-11-06).
- Released for Gecode 4.3.1 (2014-10-22).
- Documented changed restart-based search in Section 9.4 and added information on how to use it for LNS (see Section 9.4.5) (2014-10-20).
- Released for Gecode 4.3.0 (2014-09-01).
- Documented multi-dimensional bin-packing constraints (see Section 4.4.15) (2014-07-27).
- Added missing edges in Figure 4.5 and Figure 4.6 (2014-06-30, thanks to Léonard Benedetti).
- Released for Gecode 4.2.1 (2013-11-05).
- Released for Gecode 4.2.0 (2013-07-19).
- Explained support for no-goods for variable-value branchers (see Section 37.4) (2013-07-10).
- Explained how to add support for no-goods to branchers (see Section 32.2) (2013-07-10).
- Explained how to use no-goods from restarts (see Section 9.5) (2013-07-10).
- Released for Gecode 4.1.0 (2013-06-13).
- Documented display of branching information in Gist (see Section 10.3.4) (2013-05-13).
- Documented variable-value print functions for branching (see Section 8.12) (2013-05-03).
- Released for Gecode 4.0.0 (2013-03-14).
- Fixed documentation of user-defined variable selection (the `_MERIT_` part was missing) (see Section 8.7) (2013-04-12, thanks to Roberto Castañeda Lozano).
- Documented LDSB (see Section 8.10) (2013-03-08).
- Documented restart-based search (see Section 9.4), added example (see Section 21.4) (2013-02-22).
- Complete rewrite of how to branch (you should read it again), see Chapter 8 (2013-02-22).

- Added missing copy constructors and assignment operators in Section 30.3 and Section 30.4 (2013-02-14, thanks to David Rijsman).
- Documented how to implement variable-value branchings (see Chapter 37) (2013-02-04).
- Documented how to implement constraints over float variables (see Chapter 29) (2013-02-04).
- Documented modeling with floats (see Chapter 6, Section 7.1.4, Section 8.4, and Section 2.6.2) (2013-01-29).
- Explain new search options for Gist in Section 10.3.6 (2013-01-25).
- Fixed typo in Section 25.3 (2012-12-17, thanks to Benjamin Negrevergne).
- Explained how to use half reification (see Section 4.3.4) and how to implement it (see Section 24.3) (2012-10-19).
- Properly explained regions (see Section 30.1) (2012-09-07).
- Documented hardware-based random seed generation for random branchers (see Section 8.6) (2012-08-29).
- Documented `pow` and `nroot` constraints (see Section 4.4.5 and Section 7.1.1) (2012-08-27).
- Fixed explanation of advisor deltas (see Section 26.3) (2012-08-21, thanks to Max Ostrowski).
- Explained activity-based search and shared variable selection criteria (see Chapter 8) (2012-03-06).
- Released for Gecode 3.7.3 (2012-03-20).
- Released for Gecode 3.7.2 (2012-02-22).
- Added tip that compilers for Qt and Gecode must match (see Tip 2.10) (2011-11-10, thanks to Pavel Bochman).
- Released for Gecode 3.7.1 (2011-10-10).
- Explained semantics of n-ary implication (see Section 4.4.4) (2011-10-06).
- Released for Gecode 3.7.0 (2011-08-31).
- Added links to the Global Constraint Catalog (GCCAT, [5]) (2011-08-22).
- Documented membership constraints (see Section 4.4.2) (2011-08-22).
- Documented number of values constraints (see Section 4.4.9) (2011-08-17).
- Fixed error in explanation of value precedence constraint for multiple values (see Section 4.4.19) (2011-08-17, thanks to Chris Mears).
- Added missing information on creating a variable implementation disposer (see Section 34.4) (2011-08-15, thanks to Gustavo Gutierrez).

- Fixed some typos (2011-07-25, thanks to Pierre Flener).
- Released for Gecode 3.6.0 (2011-07-15).
- Documented `precede` constraint (see Section 5.2.9) (2011-07-13).
- Explained that constraint post functions are clever in that they select a good propagator (see Section 4.3) (2011-07-08, thanks to Kish Shen).
- Documented `precede` constraint (see Section 4.4.19) (2011-06-30).
- Documented `nooverlap` constraint (see Section 4.4.16) (2011-06-07).
- Documented `path` constraint for Hamiltonian paths (see Section 4.4.17) (2011-06-07).
- Moved graph and scheduling constraints to integer module (see Section 4.4.17 and Section 4.4.18) (2011-05-26).
- Fixed example for `count` constraint (2011-05-03, thanks to Kish Shen).
- Added Tip 2.9 about the library path to the compilation instructions (2011-03-28, thanks to Gabriel Hjort Blindell, Flutra Osmani).
- Added pointers to MiniModel reference documentation (2011-03-24).
- Added archiving for choices and branchers (2011-03-14).
- Adapted to new names for set channeling constraints (2011-02-22).
- Added that Gecode on Windows requires Microsoft Visual C++ 2008 or better (2011-02-13).
- Added missing **`int`**`.hh` file (2011-02-11, thanks to Gustavo Gutierrez).
- Released for Gecode 3.5.0 (2011-02-01).
- Added bin packing case study (Chapter 19) (2011-01-28).
- Documented STL-style array iterators (Section 4.2.3) (2011-01-25, thanks to Gregory Crosswhite).
- Released for Gecode 3.4.2 (2010-10-09).
- Removed discussion of limited discrepancy search (2010-10-09).
- Released for Gecode 3.4.1 (2010-10-06).
- Documented the `binpacking` constraint (Section 4.4.15) (2010-10-06).
- Added explanation how to initially schedule a propagator using advisors (see Tip 26.2) (2010-10-05, thanks to Chris Mears).
- Added installation and compilation instructions (moved and eexpanded from the reference documentation) (Section 2.6) (2010-10-05).
- Explain that variables are re-selected during branching (Tip 8.1) (2010-09-02, thanks to Kish Shen).

- Explain branch filter functions (Section 8.11) (2010-09-01, thanks to Felix Brandt).
- Documented that variable implementation views are parametric with respect to variables (Section 36.2) (2010-08-31).
- Documented that the variable base class is `VarImpVar` (Section 35.1) (2010-08-31).
- Many small fixes everywhere (language, presentation, references) (2010-07-30).
- Released for Gecode 3.4.0 (first complete version) (2010-07-26).
- Added a how to read section (Section 1.3) and overview material to each chapter and part (2010-07-21).
- Added the part on programming search engines (Part S) (2010-07-20).
- Added the part on programming variables (Part V) (2010-07-02).
- Explain that the compiler defines the platform used on Windows (Tip 2.8) (2010-06-17, thanks to Dan Scott).
- Explained that variables do not have `init()` functions as they are not needed in Tip 4.2 (2010-06-04).
- Fixed typo in Section 9.1.2 (2010-05-10, thanks to Andreas Karlsson).
- Documented new MiniModel for set constraints (Section 7.1) and adapted to other MiniModel changes (2010-05-07).
- Added more case studies (2010-05-06).
- Documented new operations on argument arrays (Section 4.2.2) (2010-05-06).
- Only use absolute URLs as not all PDF viewers honor the base URL (2010-04-11).
- Released for Gecode 3.3.1 (first release of "Modeling and Programming with Gecode") (2010-04-09).
- Fixed typo in Figure 2.5 (2010-04-01, thanks to Seyed Hosein Attarzadeh Niaki).
- Released for Gecode 3.3.0 (2010-03-13).
- Described that linear expressions can freely mix integer and Boolean variables and that also sum expressions are supported (Section 7.1) (2010-02-01).
- Fixed typo in example (Section 9.1.2) (2010-01-18, thanks to Vincent Barichard).
- Added tips for linking libraries (Tip 2.7 and Tip 3.1) (2010-01-18).
- Released for Gecode 3.2.2 (2009-11-30).
- Documented `sequence` constraints (Section 4.4.10) (2009-11-24).
- Released for Gecode 3.2.1 (2009-11-04).
- Explained integer shared arrays for `element` (Tip 4.9) (2009-11-01).
- Explained `Home` (Tip 2.1) (2009-10-16).
- Documented AFC-based variable selection for branching (Section 8.2) (2009-10-13).

- Fixed link for reporting bugs (2009-10-08).
- Released for Gecode 3.2.0 (2009-10-05).
- Fixed some broken links (2009-06-15, thanks to Sverker Janson).
- Documented branching on single variables (Section 8.1) (2009-06-08).
- Documented element constraint for matrix interface (Section 7.2) (2009-06-08).
- Released for Gecode 3.1.0 (2009-05-20).
- Documented parallel search (Section 9.2) (2009-05-12).
- Clarified the use of `"<GECODEDIR>"`. (Section 2.3.1) (2009-05-08, thanks to Markus Böhm).
- Documented script commandline driver (Section 3.3, Chapter 11) (2009-04-20).
- Documented `wait` post functions (Section 4.5, Section 5.3) (2009-04-09).
- Released for Gecode 3.0.2 (2009-03-26).
- Fix for gcc compilation instructions in Section 2.3.3 (2009-03-26, thanks to Roberto Castañeda Lozano).
- Released for Gecode 3.0.1 (2009-03-24).
- Generate shorter inter-document references to avoid problems with some PDF viewers (2009-03-23, thanks to Håkan Kjellerstrand).
- Initial release for Gecode 3.0.0 (2009-03-13).

License

This documentation is provided under the terms of the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 license. A summary of the full Legal Code below can be found at the URL

http://creativecommons.org/licenses/by-nc-nd/3.0/.

Legal Code

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions.

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work

is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by

any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection,

upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.

b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

c. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as

permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii. **Voluntary License Schemes.** The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b).

e. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be,

granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.