



Comet Tutorial

Dynamic Decision Technologies Inc.

© *March 24, 2010*

Copyright Notice

Copyright © 2010, by Dynamic Decision Technologies, Inc.
One Richmond Square, Providence RI 02906, United States. All rights reserved.

General Use Restrictions

This documentation and the software and methodologies described in this documentation are the property of Dynamic Decision Technologies, Inc. (Dynadec) and are protected as Dynadec trade secrets. They are furnished under a license or nondisclosure agreement, and may be used or copied only within the terms of such license or nondisclosure agreement.

No part of this work may be reproduced or disseminated in any form or by any means, without the prior written permission of Dynadec in each and every case.

Trademarks

COMET and Dynadec are registered trademarks or trademarks of Dynamic Decision Technologies, Inc. the United States and/or other countries.

All other company and product names are trademarks or registered trademarks of their respective holders.

Preface

You've got to love to play
– Louis Armstrong

This document is a tutorial about COMET, an hybrid optimization system, combining constraint programming, local search, and linear and integer programming. It is also a full object-oriented, garbage collected programming language, featuring some advanced control structures for search and parallel programming, supplemented with rich visualization capabilities.

This tutorial describes the basic functionalities of COMET 2.1 and how it can be used to solve a variety of combinatorial optimization applications. It complements the API description which describes the various classes, functions, interfaces, and libraries through a set of HTML pages. This document is constantly evolving to cover more functionalities of the COMET system and to explain how to use COMET on increasingly complex problems. The COMET system itself is constantly evolving and the material will be updated accordingly.

The tutorial is currently divided in six parts: the first part gives a description of the programming language; the next three parts focus on constraint programming, local search, and mathematical programming; the fifth part describes the graphical and visualization layers of COMET the last part of the tutorial presents how COMET can be interfaced with databases, XML documents, C++ and JAVA.

Questions about COMET, its uses, and bug reports can be posted at the Dynadec forum at <http://forums.dynadec.com/>.

Table of Contents

| | |
|--|------------|
| Copyright Notice | i |
| Preface | iii |
| Table of Contents | v |
| List of Statements | xv |
| I. Comet: An Object-Oriented Language | 1 |
| 1 Using Comet | 3 |
| 1.1 Hardware Requirements | 4 |
| 1.2 Running a Comet Program | 4 |
| 1.3 Launching Comet Studio | 5 |
| 1.4 Debugging Comet | 6 |
| 1.5 Command Line Arguments | 8 |
| 1.6 File Inclusion: include and import | 9 |

| | | |
|----------|---|-----------|
| 2 | Basic Data Types | 11 |
| 2.1 | Integer Numbers | 12 |
| 2.2 | Rational Numbers | 15 |
| 2.3 | Boolean | 17 |
| 2.4 | Strings | 19 |
| 2.5 | File Input-Output | 24 |
| 3 | Advanced Data Types | 27 |
| 3.1 | Ranges | 28 |
| 3.2 | Arrays and Matrices | 29 |
| 3.2.1 | One-Dimensional Arrays | 29 |
| 3.2.2 | Multidimensional Arrays (Matrices) | 33 |
| 3.3 | Sets | 36 |
| 3.3.1 | Set Operations | 38 |
| 3.3.2 | Generating Sets with filter, collect, argMin and argMax | 40 |
| 3.4 | Dictionaries | 42 |
| 3.5 | Stacks | 44 |
| 3.6 | Queues | 45 |
| 3.7 | Heaps | 46 |
| 4 | Flow Control and Selectors | 47 |
| 4.1 | Flow Control | 48 |
| 4.2 | Selectors | 52 |
| 4.2.1 | select | 53 |
| 4.2.2 | selectMin, selectMax | 54 |
| 4.2.3 | selectPr | 56 |
| 4.2.4 | selectFirst | 57 |
| 4.2.5 | selectCircular | 58 |

| | | |
|------------|---|------------|
| 5 | Language Abstractions | 59 |
| 5.1 | Functions | 60 |
| 5.2 | Tuples | 63 |
| 5.3 | Classes and Objects | 63 |
| 5.4 | Interfaces | 67 |
| 5.5 | Inheritance | 68 |
| 5.6 | Operator overloading and equality testing | 69 |
| 5.7 | The Comparable Interface | 71 |
| 5.8 | Pretty Print of Objects | 73 |
| 5.9 | Exceptions | 74 |
| 6 | Closures and Continuations | 75 |
| 6.1 | Closures | 76 |
| 6.2 | Continuations | 77 |
| 7 | Events | 79 |
| 7.1 | Simple Events | 80 |
| 7.2 | Key Events | 82 |
| 8 | Concurrency | 85 |
| 8.1 | Threads | 86 |
| 8.2 | Synchronization | 87 |
| 8.3 | Unbounded parallel loops | 93 |
| 8.4 | Bounded parallel loops | 94 |
| 8.5 | Interruptions | 96 |
| 8.6 | Events | 97 |
| 8.7 | Further Notes | 98 |
| II. | Constraint Programming | 99 |
| 9 | Introduction to Constraint Programming | 101 |
| 9.1 | Variables | 102 |

| | | |
|-----------|---|------------|
| 9.2 | Constraints | 103 |
| 9.3 | Search | 105 |
| 9.4 | Testing if a Solution Was Found | 108 |
| 9.5 | The Queens Problem: Hello World of CP | 109 |
| 9.6 | Optimization in CP | 111 |
| 10 | List of Constraints | 115 |
| 10.1 | Posting a Constraint | 116 |
| 10.2 | The Fixpoint Algorithm | 117 |
| 10.3 | Consistency Level | 119 |
| 10.4 | Constraints on Integer Variables | 120 |
| 10.4.1 | Arithmetic and Logical Constraints | 121 |
| 10.4.2 | Element Constraints | 123 |
| 10.4.3 | Table Constraint | 124 |
| 10.4.4 | Alldifferent Constraints | 125 |
| 10.4.5 | Cardinality Constraints | 126 |
| 10.4.6 | Knapsack Constraints | 128 |
| 10.4.7 | Circuit Constraints | 130 |
| 10.4.8 | Sequence Constraint | 132 |
| 10.4.9 | Stretch and Regular Constraints | 133 |
| 10.4.10 | Soft Global Constraints | 137 |
| 11 | Set Variables | 143 |
| 11.1 | Representation | 144 |
| 11.2 | Interface | 145 |
| 11.3 | Constraints over Set Variables | 148 |
| 11.3.1 | Basic Set Operation Constraints | 148 |
| 11.3.2 | Cardinality | 149 |
| 11.3.3 | Requires and excludes | 150 |
| 11.3.4 | Channeling | 150 |
| 11.3.5 | Set Global Cardinality | 152 |
| 11.4 | The SONET Problem | 154 |

| | |
|---|------------|
| 12 Constraint Programming Examples | 159 |
| 12.1 Labeled Dice | 160 |
| 12.2 Bin-Packing | 163 |
| 12.3 Euler Knight | 167 |
| 12.4 Perfect Square | 169 |
| 12.5 Car Sequencing | 172 |
| 12.6 Sport Scheduling | 174 |
| 12.7 Radio Link Frequency Assignment | 178 |
| 12.8 Steel Mill Slab Design | 180 |
| 12.9 Eternity II | 186 |
| 13 Constraint Programming Techniques | 195 |
| 13.1 Non-Deterministic Search | 197 |
| 13.2 Choosing the Right Decision Variables | 206 |
| 13.2.1 Bin-Packing Problem | 206 |
| 13.2.2 Queens Problem | 210 |
| 13.3 Variable and Value Heuristic | 213 |
| 13.3.1 Variable Heuristic | 213 |
| 13.3.2 Value Heuristic | 214 |
| 13.3.3 Domain Splitting | 215 |
| 13.3.4 Design of a Branching Heuristic for the Queens Problem | 216 |
| 13.4 Designing Heuristics for Optimization Problems | 218 |
| 13.4.1 From a Greedy to a Good Non-Deterministic Search | 219 |
| 13.4.2 From a Good Initial Solution to a Non-Deterministic Search | 223 |
| 13.5 Dynamic Symmetry Breaking During Search | 225 |
| 13.6 Restarts | 228 |
| 13.7 Large Neighborhood Search (LNS) | 232 |
| 13.7.1 Choosing Relaxation Procedure and Failure Limits | 233 |
| 13.7.2 Starting from an Initial Solution | 234 |
| 13.7.3 Combining LNS with Restarts | 236 |
| 13.7.4 Adaptive LNS | 238 |
| 13.7.5 Differences between Restarts and LNS | 240 |
| 13.8 Speeding Up Branch and Bound with CBLS* | 241 |

| | |
|---|------------|
| 14 Over-Constrained Problems | 245 |
| 14.1 Dropping-Then-Relaxing Constraints | 246 |
| 14.2 An Over-Constrained Time-Tabling Problem | 247 |
| 14.3 An Over-Constrained Personnel Scheduling Problem | 252 |
| 15 Propagators | 257 |
| 15.1 Propagator for the Modulo Constraint using AC3 Events | 258 |
| 15.2 Incremental Propagator for the Modulo Constraint with AC5 Events | 263 |
| 15.3 Implementing a Reified Constraint | 269 |
| 16 Scheduling | 275 |
| 16.1 Unary Resources: The Job Shop Problem | 277 |
| 16.2 Unary Sequence Resources | 281 |
| 16.3 Discrete Resources: Cumulative Job Shop | 284 |
| 16.3.1 Cumulative Job Shop Scheduling | 284 |
| 16.3.2 Explanation of setTimes | 286 |
| 16.3.3 Adding an LNS Component | 287 |
| 16.4 Reservoirs | 289 |
| 16.5 State Resources: The Trolley Problem | 292 |
| 16.5.1 Trolley Problem Data | 293 |
| 16.5.2 Modeling the Trolley Problem | 295 |
| 16.5.3 Modeling Limited Trolley Capacity | 298 |
| 16.6 Resource Pools | 301 |
| 16.6.1 Unary Resource Pool | 301 |
| 16.6.2 Unary Sequence Pool | 305 |
| 16.6.3 Discrete Resource Pool | 307 |
| 17 Constraint Programming and Concurrency | 309 |
| 17.1 Parallel Solving | 310 |
| 17.2 Parallel Graph Coloring | 311 |

| | |
|--|------------|
| III. Constraint-Based Local Search | 315 |
| 18 Getting Started with CBLS | 317 |
| 18.1 The Queens Problem | 318 |
| 18.2 Magic Squares | 326 |
| 18.3 Send More Money | 332 |
| 18.4 Magic Series | 338 |
| 18.5 All-Interval Series | 344 |
| 19 Local Search Structures | 351 |
| 19.1 Incremental Variables | 352 |
| 19.2 Invariants | 355 |
| 19.2.1 Numerical Invariants | 356 |
| 19.2.2 Combinatorial Invariants | 356 |
| 19.2.3 Set Invariants | 358 |
| 19.3 Constraints | 362 |
| 19.3.1 The Constraint Interface | 362 |
| 19.3.2 Violations | 363 |
| 19.3.3 Numerical Constraints | 366 |
| 19.3.4 Combinatorial Constraints | 366 |
| 19.3.5 Compositionality of Constraints | 370 |
| 19.4 A Time Tabling Problem | 373 |
| 19.5 Solutions - Neighbors | 382 |
| 19.5.1 The Progressive Party Problem | 382 |
| 19.5.2 Using Solutions | 390 |
| 19.5.3 Using Neighbors | 396 |
| 20 CBLS User Extensions | 403 |
| 20.1 User-Defined Constraints | 404 |
| 20.1.1 User-Defined AllDistinct Constraint | 404 |
| 20.1.2 The Queens Problem using the AllDistinct constraint | 409 |
| 20.2 User-Defined Functions | 411 |

| | |
|---|------------|
| 20.2.1 The Steel Mill Slab Problem | 411 |
| 20.2.2 User-Defined Function for Steel Mill Slab | 417 |
| 21 CBLS Applications | 421 |
| 21.1 The Warehouse Location Problem | 422 |
| 21.1.1 Modeling Warehouse Location | 422 |
| 21.1.2 WarehouseLocation Class Constructor | 426 |
| 21.1.3 Basic Tabu Search | 429 |
| 21.1.4 Iterated Tabu Search | 433 |
| 21.2 The Social Golfers Problem | 435 |
| 21.2.1 Modeling Social Golfers | 435 |
| 21.2.2 The Search | 438 |
| 21.2.3 The User-Defined Constraint SocialTournament | 441 |
| 21.2.4 User-Defined Invariants | 447 |
| IV. Mathematical Programming | 453 |
| 22 Linear Programming | 455 |
| 22.1 Production Planning | 456 |
| 22.2 Warehouse Location | 458 |
| 22.3 Column Generation | 461 |
| 22.3.1 Cutting Stock Problem | 461 |
| 22.3.2 Hybridization | 467 |
| V. Visualization | 469 |
| 23 Comet's Graphical Interface | 471 |
| 23.1 Visualizer and CometVisualizer | 472 |
| 23.2 Adding Widgets through Notebook Pages | 474 |
| 23.3 Dealing with Colors and Fonts | 480 |
| 23.4 Text Tables | 481 |
| 23.5 Drawing Boards | 484 |

| | |
|---|------------|
| 24 Visualization Events | 489 |
| 24.1 Reacting to Widget Events | 490 |
| 24.2 Capturing Mouse Input | 492 |
| 24.3 Updating an Interface Using Events | 495 |
| 25 Visualization Examples | 505 |
| 25.1 Visualization for the Queens Problem (CBLS) | 506 |
| 25.2 Animated Visualization for Time Tabling (CBLS) | 509 |
| 25.3 Visualization for Job Shop Scheduling (CP) | 513 |
| VI. Interfacing Comet | 521 |
| 26 Database Interface | 523 |
| 26.1 Setting up ODBC | 524 |
| 26.2 Connecting to the Database | 525 |
| 26.3 Performing Queries | 526 |
| 26.4 Retrieving Data | 527 |
| 27 XML Interface | 529 |
| 27.1 Sample XML File | 530 |
| 27.2 Reading from an XML File | 532 |
| 27.3 Writing to an XML File | 538 |
| 28 C++ and Java Interface | 543 |
| 28.1 C++ Interface | 544 |
| 28.2 Java Interface | 551 |
| Bibliography | 553 |
| Index | 555 |

List of Statements

| | | |
|-------|---|-----|
| 9.1 | CP Model for the Queens Problem | 110 |
| 9.2 | CP Model for the Balanced Academic Curriculum Problem | 112 |
| 11.1 | Model for the SONET Problem | 155 |
| 12.1 | CP Model for the Labeled Dice Problem | 161 |
| 12.2 | CP Model for the Bin Packing Problem | 164 |
| 12.3 | CP Model for the Euler Knight Problem | 168 |
| 12.4 | CP Model for the Perfect Square Problem | 170 |
| 12.5 | CP Model for the Car Sequencing Problem | 172 |
| 12.6 | CP Model for the Sport Scheduling Problem | 175 |
| 12.7 | CP Model for the Frequency Assignment Problem | 179 |
| 12.8 | CP Model for the Steel Mill Slab Problem | 182 |
| 12.9 | CP Model for Eternity II (Part 1/3) | 187 |
| 12.10 | CP Model for Eternity II (Part 2/3) | 189 |
| 12.11 | CP Model for Eternity II (Part 3/3) | 190 |
| 13.1 | Naive CP Model for a Small Bin-Packing Problem | 207 |
| 13.2 | Naive CP Model for the Queens Problem | 211 |
| 13.3 | Improved CP Model for the Queens Problem | 212 |
| 13.4 | CP Model for the Queens Problem with First-Fail Search | 217 |
| 13.5 | Greedy Algorithm for the Quadratic Assignment Problem | 220 |
| 13.6 | CP Model for QAP Using Search Adapted from a Greedy Algorithm | 222 |
| 13.7 | CP Model for QAP with Labeling Based on Initial Solution | 224 |
| 13.8 | Dynamic Symmetry Breaking During Search on the Scene Scheduling Problem | 226 |
| 13.9 | Magic Square Problem | 229 |
| 13.10 | Magic Square Problem with Restarts | 231 |
| 13.11 | Starting LNS with an Initial Solution: Illustration on the QAP | 235 |
| 13.12 | LNS with Restarts for the QAP | 237 |

| | |
|--|-----|
| 13.13 Adaptive LNS for the QAP | 239 |
| 13.14 Hybridization of CP and CBLS for the QAP | 243 |
| 14.1 Time-Tabling with Hard Constraints | 248 |
| 14.2 Time-Tabling with Soft Constraints | 251 |
| 14.3 Personnel Scheduling with Hard Constraints | 253 |
| 14.4 Personnel Scheduling with Soft Constraints | 255 |
| 15.1 General Structure of the Modulo Constraint | 259 |
| 15.2 Modulo Constraint Implementation | 260 |
| 15.3 Modulo Constraint with AC5 Events (Part 1/2) | 264 |
| 15.4 Modulo Constraint with AC5 Events (Part 2/2) | 265 |
| 15.5 Reified Equality Constraint (Part 1/2) | 270 |
| 15.6 Reified Equality Constraint (Part 2/2) | 272 |
| 16.1 CP Model for Job-Shop Scheduling | 278 |
| 16.2 CP Model for Job-Shop Scheduling Using Unary Sequences | 283 |
| 16.3 CP Model for the Cumulative Job-Shop Problem | 285 |
| 16.4 CP Model for Cumulative Job-Shop with LNS | 288 |
| 16.5 Simple Reservoir Example | 290 |
| 16.6 Data for the Trolley Problem | 294 |
| 16.7 CP Model for the Uncapacitated Trolley Problem | 296 |
| 16.8 CP Model for the Trolley Problem with Capacity 3 | 299 |
| 16.9 CP Model for Flexible Job Shop Using Resource Pools | 303 |
| 16.10 CP Model for Flexible Job Shop Using Sequence Pools | 306 |
| 17.1 Parallel COMET Model for Graph Coloring | 313 |
| 18.1 CBLS Model for the Queens Problem | 319 |
| 18.2 Generic Max-Conflict/Min-Conflict Search Procedure | 325 |
| 18.3 CBLS Model for the Magic Square Problem | 327 |
| 18.4 Generic Swap-Based Tabu Search | 331 |
| 18.5 CBLS Model for the “Send More Money” Puzzle | 333 |
| 18.6 Generic Constraint-Directed Search | 336 |
| 18.7 CBLS Model for the Magic Series Problem | 339 |
| 18.8 CBLS Model Using Functions for the All-Interval Series | 345 |
| 19.1 CBLS Model for the Queens Problem Using Invariants | 361 |
| 19.2 CBLS Model for Time-Tabling: I. Data Collection | 375 |
| 19.3 CBLS Model for Time-Tabling: II. Preprocessing and Model | 377 |
| 19.4 CBLS Model for Time-Tabling: III. Initialization and Search | 380 |
| 19.5 Progressive Party in CBLS: The Model | 383 |
| 19.6 Progressive Party in CBLS: Dynamic Length Tabu Search | 386 |
| 19.7 Progressive Party in CBLS: Aspiration Criteria | 388 |
| 19.8 Progressive Party in CBLS: Intensification | 392 |
| 19.9 Progressive Party in CBLS: Restarts | 394 |

| | |
|---|-----|
| 19.10CBLS Model for the Progressive Party Problem (Part 1/2) | 398 |
| 19.11CBLS Model for the Progressive Party Problem (Part 2/2) | 401 |
| 20.1 User-Defined AllDistinct Constraint | 405 |
| 20.2 CBLS Model for the Queens Problem Using a User-Defined Constraint | 410 |
| 20.3 CBLS Model for the Steel Mill Slab Problem | 413 |
| 20.4 User-Defined SteelObjective Function for the Steel Mill Slab Problem | 418 |
| 21.1 CBLS Model for the Warehouse Location Problem: I. The Class Statement | 424 |
| 21.2 CBLS Model for the Warehouse Location Problem: II. Constructor and Model | 427 |
| 21.3 CBLS Model for the Warehouse Location Problem: III. Tabu Search | 430 |
| 21.4 CBLS Model for the Warehouse Location Problem: IV. Iterated Tabu Search | 434 |
| 21.5 CBLS Model for the Social Golfer Problem: I. The Model | 436 |
| 21.6 CBLS Model for the Social Golfer Problem: II. Tabu Search | 439 |
| 21.7 User-Defined Constraint SocialTournament: I. Statement and Constructor | 442 |
| 21.8 User-Defined Constraint SocialTournament: II. Implementation | 444 |
| 21.9 User-Defined Invariant Meet: I. Statement and Constructor | 449 |
| 21.10User-Defined Invariant Meet: II. Implementation | 451 |
| 22.1 Linear Programming Model for Production Planning | 457 |
| 22.2 MIP Model for the Warehouse Location Problem | 459 |
| 22.3 Column Generation Approach to the Cutting Stock Problem (Part 1/2) | 464 |
| 22.4 Column Generation Approach to the Cutting Stock Problem (Part 2/2) | 466 |
| 22.5 Cutting Stock Problem Using CP for Solving the Pricing Problem | 468 |
| 23.1 Putting a Label in Each Area Through Notebook Pages | 475 |
| 23.2 Using a Text Table | 482 |
| 23.3 Declaring a VisualDrawingBoard | 486 |
| 24.1 Reacting to a Clicked Button | 491 |
| 24.2 Capturing Mouse Events | 494 |
| 24.3 Updating a Playing Board Using Events: I. Playboard Class (1/2) | 496 |
| 24.4 Updating a Playing Board Using Events: I. Playboard Class (2/2) | 497 |
| 24.5 Updating a Playing Board Using Events: II. Visualization Class | 499 |
| 24.6 Updating a Playing Board Using Events: III. Moving Pieces on the Board | 501 |
| 25.1 CBLS Model for the Queens Problem with Visualization | 507 |
| 25.2 CBLS Model for Time-Tabling: IV. Visualization | 510 |
| 25.3 Visualization for Job Shop Using Gantt Charts (1/2) | 515 |
| 25.4 Visualization for Job Shop Using Gantt Charts (2/2) | 516 |
| 27.1 Reading and Printing from a Sample XML File: Reading Route Data | 534 |
| 27.2 Reading and Printing from a Sample XML File: Reading Product Data | 536 |
| 27.3 Writing to a Sample XML File: Rewriting Product Data | 539 |
| 28.1 CP Model for Job-Shop Scheduling with Enhancements for C++ Interfacing | 545 |
| 28.2 C++ Code for Interfacing with COMET Code for Job-Shop Scheduling (1/2) | 547 |
| 28.3 C++ Code for Interfacing with COMET Code for Job-Shop Scheduling (2/2) | 548 |

| COMET: AN OBJECT-ORIENTED LANGUAGE

Chapter 1

Using Comet

This chapter provides basic information on how to run and debug COMET programs. It gives an account of hardware requirements, and shows how to deal with command line arguments, and how to include files and import COMET modules. It also describes how to launch COMETSTUDIO, an IDE for developing COMET applications. COMET.

1.1 Hardware Requirements

We first give a summary of the minimum hardware requirements, in order to run COMET.

- Disk space: 200MB
- Memory: At least 1GB. More memory may be needed for solving larger problems.
- Supported Processors:
 - x86 Pentium 4, or compatible
 - x86_64 AMD64, or compatible
- CPU must support sse2 instructions.

1.2 Running a Comet Program

Running a COMET source file, that is, a file with an extension `.co` (or `.cob`, for encrypted files) can be done from the command line. For example, to run the source file `sourcefile.co`, one simply needs to type:

```
comet sourcefile.co
```

There is a number of command line options, that control how COMET runs the source code:

- `-jx` specifies the JIT (just-in-time compiler) level (0,1,2) [default: 2; running with `-j0` disables JIT]
- `-q` runs COMET in silent mode (no messages printed)
- `-d` runs COMET in deterministic mode. Useful for repeatability of results
- `-t` (tracking) stores the random seeds used in the execution into the file `rndStream.txt`
- `-r` (replay) uses the random seeds stored in file `rndStream.txt`
- `-pXM` instructs COMET to preallocate X MB of virtual machine memory
- `-pXK` instructs COMET to preallocate X KB of virtual machine memory
- `-ipath` specifies inclusion path (directories separated by “:”, for example, `-i../dir1:dir2/`).
COMET will then search for imported files in both the inclusion and the working path

Before proceeding, here are some more details on tracking and replaying. Tracking mode allows COMET to perform a non-deterministic run. It generates the seed of the various random streams, at the beginning of the run, based on date, time, process id. It also stores all the seeds it generates in a file `rndStream.txt`, in the current working directory.

Replay mode allows to reproduce a run that was generated through tracking. COMET will start by reading the content of `rndStream.txt`, to retrieve the seeds used before and will use them. Consequently, this should faithfully reproduce the last run performed in tracking mode. This generalizes the behavior of the deterministic mode, in which COMET uses the same random sequence every time.

Caveat: Runs cannot always be faithfully reproduced, when using concurrency, since factors other than the random seed, such as thread scheduling in the platform used, can influence the order of execution.

1.3 Launching Comet Studio

Version 2.1 of COMET is shipped with a new version of COMETSTUDIO, that can be used for developing, running and debugging COMET applications. This tutorial only describes how to launch COMETSTUDIO on different platforms; a number of video tutorials explaining how to use it are going to be available at Dynadec's website.

Windows: Go to: Start | Comet | Comet Studio

Linux: Run the program: Comet/compiler/CometStudio

MacOS: Double-click the icon for: /Applications/Comet/CometStudio.app
or run: /Applications/Comet/CometStudio.app/Contents/MacOS/CometStudio

1.4 Debugging Comet

COMET natively supports a debugger. In order to use it, it suffices to add one of the following options in the command lines:

- `-gtext` runs the program using the debugger in text mode
- `-gqt` runs the program using the debugger in graphical mode

The debugger supports the same functions in both graphical and text mode. Basically, the graphical mode acts as a front-end to the text mode, so this section focuses on the text mode. Of course, the graphical interface offers enhanced usability, and it should be straight-forward to switch from the text-based to the graphical debugger.

When the text debugger starts, it shows a console prompt to the user. Typing `help` (or `h`) at the prompt will present the user with the available commands of the debugger. We now go through the most useful debugger commands.

- Tracing commands:

list, l *n* : Display next *n* lines (default: *n* = 10)

list n, m : Display next *n* lines starting from line *m*

step, s : Step into next line (stepping into any function calls)

next, n : Step over next line (without stepping into called functions)

threadInfo : List all the threads currently initied in the program

- Breakpoint-related commands:

breakin *function* : Set a breakpoint at the beginning of *function*

break n : Set a breakpoint at line *n*

break n, thread t : Set a breakpoint at line *n*, only when current thread is *t*

clear n : Clear breakpoint at line *n*

breakinfo : List all breakpoints

cont : Continue execution, until next user-defined breakpoint (or till completion, if none defined)

- Frame-related commands:

where : List all frame information (back-trace)

frame f : Information about frame f (name, return type, args, local variables)

up $[n]$: Move up n frames (to the caller frame) (default: $n = 1$)

down $[n]$: Move down n frames (called by this frame) (default: $n = 1$)

- General-purpose commands:

help, h : Print full list of commands

quit : Exit the debugger

<return key> : Repeat last command

whatis $expr$: Determine the type of expression $expr$

display $expr$: Display expression $expr$ as long as it remains in scope

print, p $expr$: Print expression $expr$ just once

do $statement$: Execute COMET $statement$

1.5 Command Line Arguments

The arguments given in the command line can be collected using `System.getArgs()`, while the number of command line arguments is retrieved with `System.argc()`. The function call `System.getArgs()` returns an array of `string` (with a range starting from 0), that stores all the arguments. Strictly speaking, each word delimited by at least one space in the command line is considered a separate argument.

The following example illustrates how to specify an input file through the command line. The input file has to be given using the syntax `-f filename`, which can be located anywhere in the command line.

```
int nbArgs = System.argc();
string[] args = System.getArgs();
string fName = "default.txt";
forall (i in 2..nbArgs)
    if (args[i-1].prefix(2).equals("-f"))
        fName = args[i-1].suffix(2);
```

For instance, if the source file `mysourcefile.co` contains the above fragment, the command line

```
comet mysourcefile.co -j2 -fdata.txt -d
```

will set `fName = "data.txt"`.

1.6 File Inclusion: include and import

COMET has two inclusion mechanisms (`include` and `import`) with slightly different objectives.

The include statement:

```
include "myfile";
```

acts pretty much like the include statement in C or C++. It pulls the specified COMET file (e.g., `myfile.co` in our case) into the model and compiles it along with the model's code. COMET locates the file based on the search path, which includes the standard location of COMET's installation and any other directory specified with the `COMET_INCLUDE` environment variable or with the `-i` option on the command line.

The import statement:

```
import cotfd;
```

does two things. First, it does the equivalent of `include "cotfd"`; i.e., it locates (via the include search path) a file named `cotfd.co` (or `cotfd.cob`) and pulls it in for compilation. Then, it locates a shared library (a `.so` file on Linux or a `.bundle` on MacOS), whose name is based on “`cotfd`”, and loads it into the process's address space. The library contains the native (C++) implementation of the classes defined in the matching `co` file (`cotfd.co`).

To sum up, `import cotfd` first includes the file `cotfd.co` (`cotfd.cob`). Then it loads the shared library `libcotfd.so` (or `cotfd.bundle` on MacOS) into the address space and binds the implementation it provides to the native APIs defined in the `.co` file. If one uses `include` instead of `import`, this will always lead to an error, since the shared libraries will not be loaded.

Chapter 2

Basic Data Types

In this chapter, we review the basic data types of COMET, which includes arithmetic and logical data types and strings. We also describe some basic mathematical functions supported, and conclude with showing how to read and write to text files. COMET has three primitive types: `int`, `float` and `bool`. These primitive types are given by value in function/method parameters. The object versions of these, which are given by reference, are: `Integer`, `Float`, `Boolean`.

2.1 Integer Numbers

Integer values are of type `int` and ranges from -2^{31} to $2^{31} - 1$ on a 32-bit system. The maximum value is accessible through the system method: `System.getMAXINT()`. The available operations between `int` that return an `int` are: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), and modulo (%). The following small fragment illustrates some `int` manipulations:

```
int a = System.getMAXINT();
int b = a-2^20+4;
int c = b%1000+3*9;
cout << c << endl;
```

Pre- and post-increment are available on `int`, as well as the shortcuts `*=`, `/=`, `+=`, `-=`, also available in JAVA or C++. For instance, the code fragment:

```
int a = 3;
cout << a++ << endl;
cout << ++a << endl;
cout << a << endl;
a *= 2;
cout << a << endl;
```

produces the output:

```
3
5
5
10
```

Comparison operators between `int` are also available (`<`, `>`, `<=`, `>=`) and they return a `bool` (`true` or `false`), as illustrated next:

```
int a = 2;
int b = 3;
bool lq = a <= b;
```

Converting an `int` to a `string` is done with the `IntToString` function:

```
string b = IntToString(2);
```

Like in JAVA, there exists the object counterpart of `int` called `Integer`. As illustrated in the next example, these must be seen as references to `int`. The fragment:

```
int b = 1;
int c = b;
c = 3;
cout << "b: " << b << endl;
Integer B(1);
Integer C = B;
C:=3;
cout << "B: " << B << endl;
```

produces the output

```
b: 1
B: 3
```


This also means that `int` are given to function parameters by value, whereas `Integer` are given by reference, as illustrated next:

```
function void opposite(int a) {  
    a = -a;  
}  
function void opposite(Integer A) {  
    A := -A;  
}  
int a = 2;  
Integer A(2);  
opposite(a);  
opposite(A);  
cout << a << endl;  
cout << A << endl;
```

This produces the output:

```
2  
-2
```

The same arithmetic operations are available for `Integer` as for `int`: `+`, `-`, `*`, `/`, `^`, `%`, `<`, `>`, `<=`, `>=`. Note that operations between an `int` and an `Integer` result in an `int`.

2.2 Rational Numbers

Rational numbers are represented with the `float` type. Every operation available on `int` is also available on `float`, except for the modulo operator (%) and the increments ++ and --. The following example illustrates some manipulations on `float`:

```
float a = 2.0;
a = a^0.5;
a += 3;
int b = 1;
float c = b+a;
cout << a << endl;
cout << c << endl;
```

This produces the output:

```
4.414214
5.414214
```

Some useful functions for removing the fractional part of a `float` and returning a `float` are:

- **round**: rounds up or down to the nearest integer (.5 decimals are rounded up)
- **floor**: rounds down to the nearest integer smaller or equal
- **ceil**: rounds up to the nearest integer larger or equal

For example:

```
cout << round(2.6) << endl;
cout << round(2.5) << endl;
cout << round(2.4) << endl;
cout << floor(2.6) << endl;
cout << ceil(2.3) << endl;
```

produces the output

```
3.000000
3.000000
2.000000
2.000000
3.000000
```

It is also possible to convert a `float` to an `int` by casting it:

```
int b = (int) 3.4;
```

The object version of `float` is `Float`. The operations and methods available for `float` are also available for `Float` but returns a `float` type. The only way to create a `Float` is via its constructor. The following fragment illustrates some manipulations on `Float`:

```
Float A(3.2);
A := A+1;
Float B(ceil(A));
cout << A << endl;
cout << B << endl;
```

This gives the output:

```
4.200000
5.000000
```

Mathematical Functions Trigonometric functions are also available on floats, as illustrated next:

```
float pi = arctan(1)*4;
float res1 = arccos(0.5);
float res2 = arcsin(0.5);
float res3 = sin(pi/2);
float res4 = cos(pi/4);
float res5 = tan(pi/4);
```

2.3 Boolean

The `bool` type can take the value `true` or `false`. The logical **or** is obtained with the `||` or `+` operator. The logical **and** is obtained with the `&&` or `*` operator. Finally, the operator `!` is used for the logical **not**. As illustrated next, `&&` has larger priority than `||` and parentheses must be used to change this priority:

```
bool a = true;
bool b = false;
bool c = true;
bool d = c || a && b;
bool e = c + a * b;
cout << d << endl;
cout << e << endl;
bool f = (c || a) && b;
cout << f << endl;
```

produces the output

```
true
true
false
```

When used in a logical operation with a `bool`, `int` are considered as `true` for values different than 0, and as `false` for the value 0:

```
bool a = true;
bool b = a && -1;
bool c = a && 0;
cout << b << endl;
cout << c << endl;
```

produces the output

```
true
false
```

Conversely, when a `bool` is implied in an integer expression, it is considered as the value 1, when it is `true`, and 0, when it is `false`:

```
bool a = true;
bool b = false;
int c = 3+a;
int d = 3+b;
cout << c << endl;
cout << d << endl;
```

produces the output

```
4
3
```

Logical operations on collections of `bool` are also available through the `or` and `and` constructs. Because of the reification of booleans to 0-1 integers, it is also possible to count the number of values that are set to `true` or `false` by using the `sum(i in R) term(i)` construct. This is equivalent to the formula $\sum_{i \in R} term(i)$:

```
bool a [1..4] = [true,false,true,true];
bool b = or(i in 1..4) a[i];
bool c = and(i in 1..4) a[i];
cout << b << endl;
cout << c << endl;
int nbTrue = sum(i in 1..4) a[i];
int nbFalse = sum(i in 1..4) (a[i]==false);
cout << nbTrue << endl;
cout << nbFalse << endl;
```

The above fragment produces the output:

```
true
false
3
1
```

2.4 Strings

A **string** is an immutable object. Two strings can be concatenated with the **+** operator or with the **concat** method.

```
string a = "tom";  
string b = "john";  
string c = a+b;  
string d = a.concat(b);
```

On the other hand, a **string** can be split into several sub-strings with the **split** method, which splits a string according to a delimiter **string**, passed to it as an input argument. In the next example, the first three names in the **string** **a** are split with the delimiter string **:** and the result is an array of **string**. Be careful that the **range** of the resulting array starts at 0. The following COMET fragment

```
string a = "tom:john:alex";  
string [] tab = a.split(":");  
cout << tab << endl;  
cout << tab.getRange() << endl;
```

produces the output

```
[tom,john,alex]  
0..2
```

The methods **prefix**, **suffix** and **substring** allow to extract a sequence of letters from the beginning, the end, or the middle of a string:

- **prefix(n)** extracts the n^{th} first letters of the string
- **suffix(from)** extracts the end letters of the string starting at position **from** (the first letter is in position 0)
- **substring(i,n)** extracts the sub-sequence of length **n** starting from position **i**

The following example illustrates the manipulation of a `string` with these methods:

```
string a = "ABCDEFGH";  
string b = a.prefix(2);  
cout << b << endl;  
string c = a.suffix(2);  
cout << c << endl;  
string d = a.substring(1,3);  
cout << d << endl;
```

It produces the output:

```
AB  
CDEFGH  
BCD
```

The equality between two strings can be tested with the method `equals` or, equivalently, with the operator `==`. Recall that testing the reference between two objects can be done with the operator `===`. For example:

```
string a = "ABC";  
string b = "ABC";  
cout << a == b << endl;  
cout << a.equals(b) << endl;
```

produces the output

```
true  
true
```

Two strings can be lexicographically compared with the method `compare`, that returns a negative `int` result, if the receiver comes before the string passed to `compare`, a positive `int` result, if the receiver comes after, and 0, if they are equal. For example:

```
string s1 = "abce";  
string s2 = "abde";  
cout << s1.compare(s2) << endl;  
cout << s2.compare(s1) << endl;
```

produces the output

```
-1  
1
```

Finding the index of the first occurrence of a substring within a given index range is achieved with the `find` method. If no occurrence can be found, `-1` is returned. The method

```
int find(string s,int start,int end)
```

returns the index of the first occurrence of `s` within the index range between `start` and `(end-1)` (inclusive). The default values for `start` and `end` (if omitted) are 0 and the string's length, respectively.

An example of use of this method is given next:

```
string a = "ABCDEFABCDEF";  
cout << a.find("DEF") << endl;  
cout << a.find("DEF",4) << endl;  
cout << a.find("DEF",4,9) << endl;  
cout << a.find("DEF",4,10) << endl;
```

This produces the output

```
3  
9  
-1  
9
```

A **string** can be trimmed from the left, from the right or from both sides with the functions **lstrip**, **rstrip** and **strip**. The argument of these functions is a string **s**, which represents the set of characters that must be stripped. The effect of a strip is to remove from the string the leading and/or trailing characters that belong to this set. The following example illustrates manipulations using these functions:

```
string a = "32123222ABC321321233";  
cout << a.lstrip("123") << endl;  
cout << a.rstrip("123") << endl;  
cout << a.strip("123") << endl;
```

produces the output

```
ABC321321233  
32123222ABC  
ABC
```

It is possible to replace all occurrences of a substring by another **string** using the method **replace**:

```
string a = "---ABC---ABC---ABC---";  
cout << a.replace("ABC","DEF") << endl;
```

The above example produces the output:

```
---DEF---DEF---DEF---
```

The methods **toInt** and **toFloat** allow to read an **int** and a **float** from a string:

```
cout << F << endl;  
string i = "12";  
int I = i.toInt();  
cout << i << endl;
```

produces the output

```
10245.300000  
12
```

2.5 File Input-Output

To illustrate file reading in COMET we show how to read the contents of a file named "input.txt" and containing the lines:

```
1.2 4 3.4 toto titi
34 4.6
```

Reading To read a file, an `ifstream` object must be created by giving it the name of the file to read. Then, the file is read as a stream, from top-left to bottom-right, by advancing a pointer from the first unread position of the file until the end of the file. The main methods used at this end are:

- `getInt()` parses the next token as an `int` and moves the pointer just after the `int` read
- `getFloat()` works exactly like `getInt` but reads a `float` number instead
- `getline()` reads until the end of the current line and stores the contents into a `string`; it then moves the pointer to the beginning of the next line
- `good()` returns `true`, if the pointer is not yet at the end of the file, and `false` otherwise

The following example shows how to read the file `input.txt`, using the above functions:

```
ifstream file("input.txt");
float f1 = file.getFloat();//1.2
int i1 = file.getInt();//4
float f2 = file.getFloat();//3.4
string s1 = file.getLine();//" toto titi"
cout << f1 << " " << i1 << " " << f2 << s1 << endl;
cout << file.good() << endl;
int i2 = file.getInt();//34
float f3 = file.getFloat();//4.6
cout << file.good() << endl;
```

The output of this is:

```
120.000000 4 3.400000 toto titi
true
false
```

Writing Writing to a file can be done with the << operator on an `ofstream` object. The `ofstream` must be closed when finished writing. The following example writes

printing... a[1,2,3,4,5,6,7,8,9,10]

into the file `outfile.txt`.

```
int a[i in 1..10] = i;
ofstream out("outfile.txt");
out << "printing... " << a << endl;
out.close();
```


Chapter 3

Advanced Data Types

This chapter describes the more advanced data types supported by COMET: ranges, arrays, matrices, and standard data-structures. These include sets, dictionaries, stacks, queues, and heaps. For all these types, the text contains illustrative examples.

3.1 Ranges

Ranges are present everywhere in COMET:

- in the definition of arrays to specify the correct indexing values
- in the arguments of loops (**forall**) and selectors
- in constraint-programming, local search and mathematical programming to specify domains of variables

A range corresponds to an integer interval $[a..b]$ between two integers a and b . A **range** is an immutable object and only four methods are available on it:

- **getLow()** returns the lower bound
- **getUp()** returns the upper bound
- **getSize()** returns the number of integers inside the range
- **atRank(i)** returns the range index at the i^{th} position (starting at 0)

Note that it is also possible to define an empty range by simply setting the upper bound smaller than the lower bound. Defining and manipulating ranges is very simple, as illustrated in the following:

```
range r = -4..10;
cout << r.getLow() << endl;
cout << r.getUp() << endl;
cout << r.getSize() << endl;
cout << r << endl;
range s = 2..1; //empty range
cout << s.getSize() << endl;
```

It is possible to treat a range, as if it were shifted to start at position 0, using the method **atRank**. The command **r.atRank(i)** is equivalent to **r.getLow()+i**. This is illustrated in the following example:

```
range r = -2..6;
cout << r.atRank(0) << endl;
```

which produces the output:

-2

3.2 Arrays and Matrices

We start by explaining how to create and manipulate one-dimensional arrays, and then explain how to manipulate matrices (multidimensional arrays).

3.2.1 One-Dimensional Arrays

An array is always associated with a range, that specifies the valid indexes for retrieving or modifying array entries. The range of an array is specified at its creation and cannot be modified later, since ranges are immutable. Also, no new range can be given as a valid indexing structure of the array. As a result, an array cannot be extended or reduced in size. This functionality is supported by other data structures in COMET, such as **sets**, **stacks** and **queues**.

The easiest way to initialize an array is to give the range and the corresponding elements (in extension) at its creation, as in the first line of the following example:

```
int tab1[1..3] = [4,7,9];
range r1 = tab1.getRange();
tab1[1] = 5;
cout << tab1[tab1.getRange().getUp()] << endl;
cout << tab1[tab1.getUp()-1] << endl;
cout << r1 << endl;
cout << tab1 << endl;
cout << tab1.getSize() << endl;
```

This produces the output:

```
9
7
1..3
tab1[5,7,9]
3
```

The range of this array `tab1` is `[1..3]` and cannot be modified after that. The three elements corresponding to the three indexes specified by the range are namely `4,7,9`.

An array can also be indexed by an enumerated type:

```
enum ALPHABET = {A,B,C};  
int tab[ALPHABET] = [7,2,9];  
int i = tab[B];
```

The available methods on arrays are:

- `getRange()`: retrieves the range of the array
- `getSize()`: retrieves the size of the array (the size of its range)
- `getLow()`: shortcut for `getRange().getLow()`
- `getUp()`: shortcut for `getRange().getUp()`
- `[i]`: allows to retrieve and modify entry at index i

Note that it is possible to define an empty array with an empty range (lower bound larger than the upper bound).

There are other ways to declare and initialize an array. One can decide to initialize every entry to the same value, as in the following example, in which every entry is initialized to 2:

```
int tab2[1..3] = 2;  
cout << tab2 << endl;
```

This produces the output:

```
tab2[2,2,2]
```

One can also initialize an array to values that are a function of the corresponding index. The following example initializes the array entries to the even numbers between 0 and 20:

```
int tab3[i in 0..10] = i*2;  
cout << tab3 << endl;
```

which produces the output

```
tab3[0,2,4,6,8,10,12,14,16,18,20]
```

It is also possible to declare the array and initialize it later, using the **new** keyword to call its constructor (but initialization in extension is not possible in this case). The code:

```
int[] tab4;  
tab4 = new int[i in 0..10] = i*2;  
cout << tab4 << endl;  
int[] tab5;  
tab5 = new int[1..3] = 2;  
cout << tab5 << endl;
```

produces the output

```
anonymous[0,2,4,6,8,10,12,14,16,18,20]  
anonymous[2,2,2]
```

Another useful feature of COMET is the possibility to initialize an array from another array, by filtering it with the instruction `all`:

```
int tab6[i in 0..20] = i;  
int[] tab7 = all(i in 0..20 : tab6[i]%2==0) tab6[i];  
cout << tab7 << endl;  
cout << tab7.getRange() << endl;
```

produces the output

```
anonymous[0,2,4,6,8,10,12,14,16,18,20]  
0..10
```

The condition `tab6[i]%2==0` of the `all` instruction filters out the even-valued elements of array `tab6` and stores them into a new array `tab7`. Note that the resulting array always has a range starting at 0, if a filtering condition is specified in the `all` construct.

The sorted permutation of an integer array can be obtained with the `sortPerm` method, as illustrated next. The algorithm used is an insertion sort algorithm, so it is most efficient, when the receiver is already almost sorted. The result of `sortPerm` is an array containing the ordered indexes of the array. For instance,

```
int a[2..6] = [4,2,7,4,9];  
int[] p = a.sortPerm();  
cout << all(i in p.getRange()) a[p[i]] << endl;
```

produces the output

```
anonymous[2,4,4,7,9]
```

3.2.2 Multidimensional Arrays (Matrices)

Multidimensional arrays are simple extensions of one-dimensional array, where a range is given for each dimension:

```
int mat1[1..3,2..3] = [[3,5],  
                       [7,8],  
                       [9,10]];  
cout << mat1 << endl;
```

produces the output

```
mat1[1,2] = 3  
mat1[1,3] = 5  
mat1[2,2] = 7  
mat1[2,3] = 8  
mat1[3,2] = 9  
mat1[3,3] = 10
```

Of course, arrays of larger dimensions can also be defined. The following example initializes a three-dimensional array with a different range for each dimension and all entries initialized to 5. Then, one of the entries is changed to 9:

```
int mat2[1..3, 2..3, 4..7] = 5;  
mat2[2,2,4] = 9;  
cout << mat2[2,2,4] << " , " << mat2[2,2,5] << endl;
```

This produces the output

```
9 , 5
```

It is also possible to retrieve the arity (number of dimensions) of a multidimensional array:

```
int nbDim = mat2.arity();
```

The methods `getRange()`, `getSize()`, `getUp()` and `getLow()` are not defined on multidimensional arrays but the equivalent methods exists for each individual dimension. Be careful that the first dimension is at index 0.

- `getRange(dim)`: returns the range of dimension `dim`
- `getSize(dim)`: returns the size of dimension `dim`
- `getUp(dim)`: shortcut for `getRange(dim).getUp()`; gives the largest index in dimension `dim`
- `getLow(dim)`: shortcut for `getRange(dim).getLow()`; gives the lowest index in dimension `dim`
- `[i1, i2, ...]`: access an element of a multidimensional array

Adding to the `mat2` matrix above, the following instructions:

```
cout << mat2.getRange(0) << endl;  
cout << mat2.getSize(1) << endl;  
cout << mat2.getUp(2) << endl;
```

produces the output

```
1..3  
2  
7
```

In the same way with one-dimensional arrays, declaration and initialization of a multidimensional array can be separated:

```
int[,] mat3;  
mat3 = new int[i in 1..2,j in 3..4] = -5;
```

It is also possible to flatten a multidimensional array with the `all` instruction. For example

```
int mat4[i in 0..2, j in 0..2] = i*3+j;  
int[] tab8 = all(i in 0..2, j in 0..2) mat4[i,j];  
cout << tab8 << endl;
```

produces the output

```
anonymous[0,1,2,3,4,5,6,7,8]
```

3.3 Sets

The following example shows how to initialize a set in extension.

```
set<int> myset1 = {1,2,5,9};
```

The available methods on a set containing elements of type T are:

- `reset()`: empties the set
- `getSize()`: returns the number of different elements in the set
- `contains(x)`: returns `true`, if element `x` is in the set, or `false` otherwise
- `insert(x)`: inserts element `x` into the set
- `delete(x)`: removes all occurrences of element `x` from the set
- `del(x)`: removes one occurrence of `x` from the set
- `copy()`: returns a copy of the set (of type `set{T}`)
- `compare(set{T})`: makes a lexicographic comparison. Returns 0 if sets are equal, -1 if set is lexicographically smaller than the argument and 1 otherwise.

The next few instructions illustrate some manipulations on sets:

```
set<int> myset1 = {1,2,5,9};
myset1.insert(2);
myset1.delete(5);
set<int> myset2 = myset1.copy();
myset2.delete(1);
cout << myset1 << myset2 << endl;
cout << "is 2 inside myset1 ? " << myset1.contains(2) << endl;
cout << "is myset1 < myset2 ? " << myset1.compare(myset2) == -1 << endl;
cout << "#myset1: " << myset1.getSize() << endl;
myset2.empty();
cout << myset1 << myset2 << endl;
cout << "is myset1 < myset2 ? " << myset1.compare(myset2) == -1 << endl;
```

This produces the output:

```
{1,2,9}{2,9}
is 2 inside myset1 ? true
is myset1 < myset2 ? true
#myset1: 3
{1,2,9}{ }
is myset1 < myset2 ? false
```

Note that COMET sets are really multi-sets (an element can be inserted more than once into the set), which explains the need of two different methods for deletion (**del** and **delete**). Using sets as multi-sets is basically made possible through the method, **del**, that allows to remove single occurrences of set items, as the following example illustrates: The first insertion introduces a second occurrence of 2 in the set (even if it's not visible on printing). Hence, one needs to call **del(2)** twice, in order to completely remove all 2s from the set:

```
set<int> myset3 = {6,2,7};
myset3.insert(2);
cout << myset3 << endl;
myset3.del(2);
cout << myset3 << endl;
myset3.del(2);
cout << myset3 << endl;
```

The above code gives the output:

```
{2,6,7}
{2,6,7}
{6,7}
```


The following methods are only available for sets of integers, since they assume a total ordering on the elements:

- `getLow()`: returns the smallest element
- `getUp()`: returns the largest element
- `atRank(r)`: returns the element at rank `r`. If the rank is not valid, an error is launched.
- `getRank(t)`: returns the rank of integer `t`. If `t` is not in the set, an error is launched.

Note that the rank of the first element of the set (the smallest element) is 0.

The following example illustrates some manipulations using these methods on sets of integers.

```
set<int> myset4 = {4,8,2};  
cout << myset4.atRank(0) << myset4.atRank(1) << myset4.atRank(2) << endl;  
cout << myset4.getRank(myset4.getLow()) << endl;  
cout << myset4.getRank(myset4.getUp()) << endl;
```

It produces the output

```
248  
0  
2
```

3.3.1 Set Operations

COMET supports standard operations on set, such as union, intersection, set difference. The following example illustrates the union of two sets:

```
set<int> A = {1,4,8};  
set<int> B = {2,4,5,8};  
set<int> C = A union B;  
cout << C << endl;
```

It produces the output

```
{1,2,4,5,8}
```

It is also possible to produce the union of an array of sets:

```
set{int} setTab[1..3] = [{1,4,8}, {2,4,5,8}, {2,5,9}];  
set{int} D = union(i in 1..3)(setTab[i]);  
cout << D << endl;
```

produces the output

{1,2,4,5,8,9}

The intersection keyword `inter` is used the same way as `union`.

```
set{int} s1 = {1,4,6,4} inter {2,4,9,1,7};  
cout << s1 << endl;
```

produces the output

{1,4}

The operator `\` is used for computing set differences in COMET. For example, the following code:

```
set{int} s2 = {1,3,5} \ {3,4};  
cout << s2 << endl;
```

produces the output

{1,5}

3.3.2 Generating Sets with filter, collect, argMin and argMax

The `filter` keyword allows to generate a `set` from a `range` or another `set` by filtering out the elements that don't satisfy a given condition. In mathematical notation it allows to create the set $\{i \mid i \in S \text{ and } \text{cond}(i)\}$, where S is a `set` or a `range` and $\text{cond}(i)$ returns a boolean. The following example shows how to keep even elements from a `set` and from a `range`:

```
set{int} myset5 = {1,3,4,5,8,10};
set{int} myset6 = filter(i in myset5)(i%2==0);
cout << myset6 << endl;
set{int} myset7 = filter(i in 1..10)(i%2==0);
cout << myset7 << endl;
```

The above code produces the output:

```
{4,8,10}
{2,4,6,8,10}
```

Given an input set or range S , the `collect` keyword allows to create a new set, that contains all the values produced by applying a given function to each element of S . In mathematical notation, it allows to generate the set $\{f(i) \mid i \in S\}$, where S is a set or a range and $f(i)$ produces values, whose type conforms to the type of the output set. For example, the following code:

```
set{string} myset8 = {"Tom","Matthieu","Emile","Simon"};
set{int} myset9 = collect(name in myset8)(name.length());
cout << myset9 << endl;
```

produces the output

```
{3,5,8}
```

Note that the `all` function is very similar to `collect` but it creates an array instead of a set. Hence, duplicates are possible inside the resulting array, as illustrated in the following example.

```
int[] nameLength = all(name in myset8)(name.length());  
cout << nameLength << endl;
```

The output of this is:

```
anonymous[5,8,5,3]
```

The keywords `argMin` and `argMax` allow to collect the set of indices corresponding to a minimum or maximum numeric evaluation, over a range or a set. The following example illustrates how to use these constructs:

```
int val[1..10]= [2,7,4,7,5,3,6,2,6,2];  
set{int} s = {1,2,5,7,8};  
set{int} minValIndex = argMin(i in s)(val[i]);  
cout << minValIndex << endl;  
set{int} maxValIndex = argMax(i in s)(val[i]);  
cout << maxValIndex << endl;
```

The output of the above is:

```
{1,8}  
{2}
```

3.4 Dictionaries

A dictionary (keyword `dict`) is a set of entries $\langle key, value \rangle$, in which the value can be retrieved with the key. Keys and values can be of any type, but must be specified at the creation of the dictionary. Here is an example of dictionary with keys of type `int` and values of type `string`:

```
dict{int->string} myDict1();  
myDict1{3} = "Laurent";  
myDict1{5} = "Pascal";  
myDict1{6} = "Andrew";  
myDict1{6} = "Ivan";  
cout << myDict1 << endl;  
cout << myDict1{6} << endl;
```

The produced output is:

```
[<3,Laurent>,<5,Pascal>,<6,Ivan>]  
Ivan
```

The call to the constructor can be explicit, as in the following example.

```
dict{int->string} myDict2;  
myDict2 = new dict{int->string}();  
dict{int->string} myDict3 = new dict{int->string}();
```

The available methods on a dictionary `dict{K -> T}` are

- `getKeys()`: returns the set (of type `set{K}`) of all keys
- `remove(K)`: removes the entries corresponding to a particular key. Nothing happens if there is no entry with such a key.
- `hasKey(k)`: returns `true`, if there is an entry with key `k`, or `false` otherwise
- `getSize()`: returns the number of entries in the dictionary
- `T operator{} (K)`: returns the value corresponding to a key, if the key is present, or null otherwise

The following example illustrates some manipulations on a dictionary:

```
dict{int->string} myDict4();  
myDict4{3} = "Namur";  
myDict4{4} = "Louvain";  
myDict4{8} = "Bruxelles";  
myDict4{1} = "Gent";  
myDict4{8} = "Antwerpen";  
myDict4.remove(1);  
cout << myDict4 << endl;  
cout << myDict4.getKeys() << endl;  
cout << myDict4.hasKey(1) << endl;  
cout << myDict4{8} << endl;
```

produces the output

```
[<3,Namur>,<4,Louvain>,<8,Antwerpen>]  
{3,4,8}  
false  
Antwerpen
```

3.5 Stacks

A stack is a let-polymorphic data type. Elements can only be added to the end of a stack, with the method `push()`, but can be removed from both ends of the stack using the methods `pop()` and `popBottom()`. Elements can also be accessed directly: the element in position i can be accessed with the `{i}` operand. The following example shows some manipulations on a stack of integers:

```
stack{int} S();
S.push(3);
S.push(5);
S.push(9);
cout << S.top() << endl; //prints 9
S.push(4);
S.push(2);
cout << S.pop() << endl; //prints 2 and remove it
cout << S << endl;
S.popBottom(); //prints 3 and remove it
cout << S << endl; //S contains 5,9,4
cout << S{1} << endl; //access the element at rank 1 i.e. 9
cout << S.getSize() << endl;
S.reset(); //empty the stack
cout << S.getSize() << endl;
```

This produces the output:

```
9
2
Stack(3,5,9,4)
Stack(5,9,4)
9
3
0
```

3.6 Queues

The queue is another let-polymorphic data type supported by COMET. Contrary to the stack, a queue can grow from both ends, with the methods `enqueueBack()` and `enqueueFront()`. On the other hand, elements cannot be accessed by their position. Instead, one has to use the methods `peekFront()` and `dequeueBack()`, as illustrated in the following example of creating and manipulating a COMET queue.

```
queue{int} q();           // create an empty integer queue
q.enqueueBack(2);         // the queue is now: [2]
q.enqueueBack(4);         // the queue is now [2,4]
int a = q.peekFront();    // a is equal to 2
int b = q.dequeueBack();  // b is equal to 4 and the queue becomes [2]
q.reset();                // empty the queue
```


3.7 Heaps

The heap is a let-polymorphic data type, that stores entries of the form $\langle K, V \rangle$, where K denotes the key type and V the element type. It allows to efficiently retrieve the stored entry that corresponds to the key with the smallest value, according to the comparison order of type K . An example of heap manipulation is given next:

```
heap{int->string} myheap();
myheap.insert(5, "pascal");
myheap.insert(7, "laurent");
myheap.insert(4, "anne");

cout << myheap.getDataMin() << endl;
myheap.pop();
cout << myheap.getDataMin() << endl;
myheap.pop();
cout << myheap.getDataMin() << endl;
cout << myheap.getKeyMin() << endl;
cout << myheap.empty() << endl;
myheap.pop();
cout << myheap.empty() << endl;
```

The output of this code is:

```
anne
pascal
laurent
7
false
true
```

Note that, in order to use as key type a user-defined object, the object's class must implement the `Comparable` interface, as explained in [Section 5.7](#).

Chapter 4

Flow Control and Selectors

This chapter gives an overview of COMET's flow control structures and selectors. COMET supports standard operators, such as **if**, **for** and **while**, along with more advanced loop control capabilities, through the **forall** construct. Selectors enhance the language's expressivity by offering different, often non-deterministic, ways of selecting elements from sets and ranges.

4.1 Flow Control

The usage of the **if** and the conditional operator **? :** follows exactly the syntax of Java or C++:

```
if (2<4)
    cout << "I'm here" << endl;
else
    cout << "I'm there" << endl;
cout << 2<4 ? "I'm here" : "I'm there" << endl;
```

This is also the case for the **switch** control structure:

```
switch (3) {
    case 1:
        cout << "it's 1" << endl;
        break;
    case 2:
        cout << "it's 2" << endl;
        break;
    case 3:
        cout << "it's 3" << endl;
        break;
    default:
        cout << "larger than 3" << endl;
        break;
}
```

The standard loop control operators **for**, **while** and **do while** also follow the Java/C++ syntax:

```
for (int i = 2; i < 5; i++)
    cout << "i:" << i << endl;

int j = 0;
do {
    j++;
} while (j < 4);

while (j > 2)
    j--;
```

The **forall** operator allows to iterate over the values of a **range** or a **set**:

```
forall (i in 1..2) {
    cout << i << endl;
}
set{int} s1 = {1,3,5};
forall (i in s1)
    cout << i << endl;
```

This produces the output

```
1
2
1
3
5
```

Furthermore, **forall** allows to iterate over several sets or ranges to generate the behavior of imbricated **for** loops:

```
set{int} s2 = {4,5,6};
forall (i in 1..2, j in s2)
    cout << i << " , " << j << endl;
```

This produces the output:

```
1 , 4
1 , 5
1 , 6
2 , 4
2 , 5
2 , 6
```

It is also possible to add a condition to the iteration, as shown in the next example. These two loops are equivalent, but the latter is more efficient, as it factors the condition pertaining only to *i* avoiding unnecessary computation:

```
set{int} s3 = {2,4,6,7,10};

forall (i in 1..4, j in s3 : i + j < 7 && i%2 == 0)
    cout << i << " , " << j << endl;
cout << endl;

forall (i in 1..4: i%2 == 0, j in s3 : i + j < 7)
    cout << i << " , " << j << endl;
```

The output produced is:

```
2 , 2
2 , 4
4 , 2

2 , 2
2 , 4
4 , 2
```

A very useful feature of **forall** is the **by** option, that allows to loop over the elements of a set or range in increasing order, according to a numerical function. The following code prints the elements in the range [1..3] in decreasing order:

```
forall (i in 1..3) by (-i)
  cout << i << endl;
```

produces the output

```
3
2
1
```

Ties in the ordering can be broken with a secondary criterion, as shown in the next example:

```
int t1[1..4] = [1,1,4,4];
int t2[1..4] = [4,3,2,1];
forall (i in 1..4) by (t1[i],t2[i])
  cout << t1[i] << " " << t2[i] << " " << i << endl;
```

this gives the output:

```
1 3 2
1 4 1
4 1 4
4 2 3
```

4.2 Selectors

Selectors allow to select an element from a **set** or a **range**, randomly, or according to a probability distribution, or according to an evaluation function. It is possible to add conditions to selectors (in similar fashion to the **forall** statement), so that only elements satisfying the given set of conditions are considered for selection. This is a summary of the selectors available in COMET:

select selects a random element

selectMin selects minimum element according to an evaluation function

selectMax selects maximum element according to an evaluation function

selectPr selects an element according to a probability distribution

selectFirst selects the lexicographically smallest element

selectCircular selector with a state; successive executions consider elements in a circular way

There are different levels of determinism in COMET's selectors. Selectors **selectFirst** and **selectCircular** are deterministic, with the latter offering a more diversified selection. In the case of **selectMin** and **selectMax**, as we'll soon see, there is the option to introduce some randomness, by randomly selecting among the k smallest or largest elements. Also, note that ties are broken randomly. Finally, there are the completely randomized selectors: **select**, which chooses uniformly at random, and **selectPr**, which chooses according to a probability density function.

We now proceed to describe the above selectors in more detail. Given the stochastic nature of most selectors, running the following examples on your computer may produce different results.

4.2.1 select

As illustrated in the following examples, the selection can be made over a set or range, or even over combinations of sets and range, to select pairs, or, more generally, tuples of elements. Note how a filtering condition is stated in some of the selectors.

```
set<int> S = {2,6,9,14,19};
select(x in S) {
    cout << x << endl;
}
select(x in 1..10)
    cout << x << endl;

select(x in S: x%2 == 0)
    cout << x << endl;

select(x in 1..10: x > 5)
    cout << x << endl;

select(x1 in 1..10, x2 in S: x1 > 3 && x2 < 12)
    cout << x1 <<" , " << x2 << endl;
```

The above code fragment produces the output:

```
9
2
2
6
4 , 2
```


Sometimes, no element satisfies the **select** condition, and it may be useful to perform an action in that case. The optional **onFailure** block of the **select** statement is intended for this purpose. The following example will print “there”, since there are no odd elements in the set $\{2, 4, 6\}$:

```
select(i in {2,4,6} : i%2 == 1) {
    cout << "here" << endl;
}
onFailure {
    cout << "there"<< endl;
}
```

4.2.2 selectMin, selectMax

Selectors are particularly useful, when a randomized greedy choice needs to be made. The **selectMin** and **selectMax** are the easiest selectors of this kind. We are going to focus on **selectMin**, since **selectMax** functions in the symmetric way. In mathematical notation, **argMin** allows to implement

$$\operatorname{argmin}_{x \in S} \{f(x) \mid \text{condition}(x)\},$$

where S is a range or a set.

The following example searches for the integer that minimizes a second degree function $x^2 - 5x + 7$. One can see that the minimum is obtained for $x = 2$ and $x = 3$. The function is evaluated to 1 for both these values. Since there are two possible values, **selectMin** chooses uniformly at random between them. Thus, several identical consecutive call to **selectMin** can produce different results in the case of ties:

```
cout << all(x in 1..10)(x^2-5*x+7) << endl;
selectMin(x in 1..10)(x^2-5*x+7)
    cout << x << endl;
```

One possible output of this is:

```
anonymous[3,1,1,3,7,13,21,31,43,57]
3
```

A condition on the candidate elements can be added to **select**:

```
selectMin(x in 1..10: x > 4) (x^2-5*x+7)
  cout << x << endl;
```

Sometimes the condition and the function to optimize for the selection are linked, as shown in the next example:

```
selectMin(x in 1..10: x^2-5*x+7 > 1) (x^2-5*x+7)
  cout << x << endl;
```

This gives the output:

4

In cases like this, it is possible to evaluate the function only once, saving computation time, when the function is costly to evaluate. The code segment that follows is equivalent to the previous one, but the function is now evaluated only once and is stored into the variable **f**:

```
selectMin(x in 1..10, f = x^2-5*x+7: f > 1) (f)
  cout << x << endl;
```

Sometimes **selectMin** may be too greedy, especially when used to select the next move in a constraint-based local search algorithm. In situations like that, it is often beneficial to introduce some randomization into the move selection. One can instruct **selectMin** to choose uniformly at random among the **k** smallest values, by providing an argument **k** to the selector and typing **selectMin[k]**. More precisely a command of the form:

```
selectMin[k](i in R) (f(i)) { block }
```

randomly picks an index **i**, whose corresponding target value, **f(i)**, belongs to the set of the **k** smallest different target values for **f(i)** (when **i** ranges in **R**).

In the next small example, which considers $k=3$, the three smallest target values are $\{1, 3, 7\}$ and the corresponding indices, that produce one of these values are $\{1, 2, 3, 4, 5\}$. This means that each selection will randomly choose one of the values among $\{1, 2, 3, 4, 5\}$, as can be checked in the output of this fragment:

```
int f[1..10] = [3,1,1,3,7,13,21,31,43,57];
forall (i in 1..20)
  selectMin[3](x in 1..10) (f[x])
    cout << x << ",";
```

The output should look like:

```
1,3,2,1,3,3,5,3,4,5,5,1,4,2,3,4,3,3,2,5,
```

4.2.3 selectPr

For some algorithms, it is interesting to select according to a discrete probability density function. The density function must be well defined, i.e., summing to 1. The following example illustrates the selection according to a one dimensional density function, defined in the array `density`:

```
float density[1..4] = [0.1,0.4,0.25,0.25];
int count[1..4] = 0;
forall (i in 1..1000)
  selectPr(i in 1..4) (density[i])
    count[i]++;
cout << count << endl;
```

A typical output of this code would be:

```
count [89,407,249,255]
```

It is also possible to select according to a multidimensional discrete density function:

```
float density2D[1..2,1..2] = [[0.1, 0.4],  
                               [0.25, 0.25]];  
int count2D[1..2, 1..2] = 0;  
forall (i in 1..1000)  
  selectPr(i in 1..2, j in 1..2) (density2D[i,j])  
    count2D[i,j]++;  
cout << count2D << endl;
```

This produces the output:

```
count2D[1,1] = 103  
count2D[1,2] = 401  
count2D[2,1] = 257  
count2D[2,2] = 239
```

4.2.4 selectFirst

The **selectFirst** selector is deterministic. It selects the lexicographically smallest element satisfying a set of conditions. The following example selects the smallest index `x`, such that the corresponding value in the array `g` is odd:

```
int g[1..10] = [6,4,7,3,9,2,8,1,0,1];  
selectFirst(x in 1..10 : g[x]%2 == 1)  
  cout << x << endl;
```

This gives the output:

```
3
```

4.2.5 selectCircular

The **selectCircular** selector is also deterministic, but can be useful to enforce diversification in stochastic algorithms. It maintains a state, that remembers the index of the last selected element. Every time **selectCircular** is called, it selects the element immediately after the last selected one, in the selection range (or set). When the end of the selection range is reached, the selection starts over from the its beginning. The selection can be restricted to elements satisfying a given condition.

In the following example, only elements 3, 6 and 9 satisfy the condition $i\%3 == 0$. These elements can be viewed as a circular linked list, and, each time **selectCircular** is called, it returns the next element in that list:

```
forall (j in 1..10)
  selectCircular(i in 1..10: i%3 == 0)
    cout << i << ",";
cout << endl;
```

produces the output

3,6,9,3,6,9,3,6,9,3,

Chapter 5

Language Abstractions

In this chapter, we give a review of some of COMET's language abstractions, illustrating the object-oriented nature of COMET. The main focus is on functions, classes and interfaces. The topics discussed include tuples, class inheritance, operator overloading and exceptions.

5.1 Functions

It is possible to declare a function that can be called before or after its definition:

```
int h[1..10] = [6,4,7,3,9,2,8,1,0,1];
cout << mySum(h,2,3) << endl;
function int mySum(int[] tab,int off,int len) {
    return sum(i in off..off+len-1) tab[i];
}
cout << mySum(h,2,4) << endl;
```

The output of this code is:

```
14
23
```

COMET functions can be of type `void`, as shown in the following example, which reverses an integer array:

```
function void reverse(int[] tab) {
    range r = tab.getRange();
    forall (i in 0..r.getSize()/2-1) {
        int tmp = tab[r.getLow()+i];
        tab[r.getLow()+i] = tab[r.getUp()-i];
        tab[r.getUp()-i] = tmp;
    }
}
int m[1..9] = [6,4,7,3,9,2,8,1,0];
reverse(m);
cout << m << endl;
```

This produces the output:

```
m[0,1,8,2,9,3,7,4,6]
```

Function arguments are given by reference for every object and by value for the basic types `int`, `float` and `bool`. If you want pass `int`, `float` and `bool` by reference, we suggest that you use the corresponding object versions of these, as shown in the following example:

```
function void modify(int i, bool b, float f, Integer I, Boolean B, Float F) {  
    i = i+1;  
    b = !b;  
    f = f+1;  
    I := I+1;  
    B := ! B;  
    F.setValue(F+1);  
}  
  
int i = 3;  
bool b = true;  
float f = 3.0;  
Integer I(i);  
Boolean B(b);  
Float F(f);  
  
modify(i,b,f,I,B,F);  
  
cout << " i: " << i << " b: " << b << " f: " << f  
    << " I: " << I << " B: " << B << " F: " << F << endl;
```

The output is:

```
i: 3 b: true f: 3.000000 I: 4 B: false F: 4.000000
```


Functions can be recursive, as in the following example, that computes the factorial of a natural number:

```
function int factorial(int n) {  
    assert(n>0);  
    if (n == 1)  
        return 1;  
    else  
        return factorial(n-1) * n;  
}  
cout << factorial(5) << endl;
```

The above code produces the output:

```
120
```

There are no global variables in COMET, so a function can only use in its computation what is given in its argument or what is declared inside the function. Note that, sometimes, we want to define as return values constants having a common meaning to all functions. The solution is to declare an enumerated type (**enum**) outside the function, like in the following example:

```
enum PossibleOutcome = {SUCCESS,FAILURE,UNCERTAIN};  
function PossibleOutcome outcome() {  
    return FAILURE;  
}  
cout << "is outcome a failure? " << outcome() == FAILURE << endl;
```

This gives the output:

```
is outcome a failure? true
```

5.2 Tuples

A tuple is a construct to store and modify immutable values. A tuple itself is immutable, because tuples can be used in the definition of invariants:

```
tuple pair{int a; int b;}  
pair p(2,3);  
cout << p << endl;  
cout << p.b << endl;
```

This fragment produces the output:

```
pair<a=2,b=3>  
3
```

5.3 Classes and Objects

Classes in COMET closely follow the syntax of Java classes. Some features of COMET classes are, in summary, the following:

- There are no modifiers on the class or on the methods: everything is public
- They support method or constructor overloading
- There is no default constructor like in Java
- Instance variables can only be accessed from inside the class or from inheriting classes. Getter and setter methods must be implemented to access or modify them from the outside.
- Instance variables cannot be initialized from outside the body of a constructor or a method
- The object `this` is available inside the class's implementation, but cannot be used to access an instance variable (e.g., with `this.instanceVar`). The reason is that COMET would treat `this` as a `tuple` (see [5.2](#)), which is not the case.

An example of a class with some methods is given next. As in C++, the **assert** function expects a boolean expression that evaluates to true, otherwise the program is interrupted at this point.

```
class Account {
    int amount;
    Account() {
        amount = 0;
    }
    Account(int initial) {
        amount = initial;
    }
    void deposit(int value) {
        assert(value > 0);
        amount += value;
    }
    void withDraw(int value) {
        amount -= value;
    }
    int getAmount() {
        return amount;
    }
}

Account pierre();
Account jean(10);
cout << "pierre " << pierre.getAmount() << endl;
cout << "jean " << jean.getAmount() << endl;
```

The above code produces the output:

```
pierre 0
jean 10
```

Like in C++, COMET allows to decouple the implementation of the methods from their declaration inside the class. The following example revisits the **Account** class, by implementing the **withdraw** method outside the class:

```
class Account {
    int amount;
    Account() {
        amount = 0;
    }
    Account(int initial) {
        amount = initial;
    }
    void deposit(int value) {
        assert(value > 0);
        amount += value;
    }
    void withdraw(int value);
    int getAmount() {
        return amount;
    }
}

void Account::withdraw(int value) {
    amount -= value;
}
```

COMET supports several types of object instantiation. The C++ like syntax:

```
Account tom(10);
```

The Java like syntax:

```
Account emile = new Account(10);
```

It is also possible to initialize an array of objects. The following line declares and initializes an array `family1` of three `Account` with an initial deposit of 10 into each account.

```
Account family1[1..3](10);
```

The following line declares and initializes an array `family2` of three `Account` with an initial deposit of 1, 2 and 3 for the first, second and third accounts, respectively:

```
Account family2[i in 1..3](i);
```

The following lines first declare and then initializes an array `family3` of three `Account`, with an initial deposit of 10 into each account:

```
Account[] family3;  
family3 = new Account[1..3](10);
```

5.4 Interfaces

The interface mechanism in COMET is exactly like in Java, and a class can implement several interfaces as illustrated in the following example:

```
interface Speaker {
    void speak();
}

interface Walker {
    void walk();
}

class Duck implements Speaker, Walker {
    Duck(){} // empty constructor
    void speak() {
        cout << "quack" << endl;
    }
    void walk() {
        cout << "walking like a duck" << endl;
    }
}

Speaker duck1 = new Duck();
duck1.speak();
Walker duck1_ = (Walker) duck1;
duck1_.walk();
Walker duck2 = new Duck();
duck2.walk();
```

The above code block produces the output:

```
quack
walking like a duck
walking like a duck
```

5.5 Inheritance

Like in Java, COMET supports single inheritance (a class can extend at most one class). It is mandatory to call the parent constructor in the constructor of a child class. Note that a class that does not explicitly extend another class implicitly extends the `Object` class, like in Java. The example given next illustrates inheritance:

```
class Book {
    string _title;
    Book(string title) {
        _title = title;
    }
    string getTitle() {
        return _title;
    }
}

class Dictionary extends Book {
    int _nbDefinitions;
    Dictionary(string title,int nbDefinitions) : Book(title) {
        _nbDefinitions = nbDefinitions;
    }
    string getDefinition() {
        return "Comets have been feared throughout much of human history,
               and even in our own time their goings and comings
               receive great attention.";
    }
}
```

5.6 Operator overloading and equality testing

The operators that can be overloaded are `+, -, *, /, <, >, <=, >=, %, ^, abs, :=, !=, ==, ===`. The only limitation is that operators `!=, ==, ===` must return a result of type `bool`. We now give an example of operator overloading. We enhance class `Account` by allowing:

- to test whether two accounts have the same amount, with the operator `==`
- to create a new account whose amount is the sum of two accounts, with the operator `+`

Let's restate the definition of class `Account`:

```
class Account {
  int amount;
  Account() {
    amount = 0;
  }
  Account(int initial) {
    amount = initial;
  }
  void deposit(int value) {
    assert(value > 0);
    amount += value;
  }
  void withdraw(int value) {
    amount -= value;
  }
  int getAmount() {
    return amount;
  }
}
```


We now show the code for overloading binary operators + and == between two `Account` objects:

```
operator bool ==(Account a1,Account a2) {  
    return a1.getAmount() == a2.getAmount();  
}  
operator Account +(Account a1,Account a2) {  
    return Account(a1.getAmount()+a2.getAmount());  
}  
Account a1(10);  
Account a2(5);  
Account a3 = a1 + a2;  
Account a4(15);  
cout << a4 == a3 << endl;    //semantic  
cout << a4 === a3 << endl;   //syntactic  
cout << a4 == a1 << endl;
```

The output of this code block is:

```
true  
false  
false
```

The operators that can be overridden in COMET are: +, -, *, /, <, >, <=, >=, %, ^, abs, :=, !=, ==, ===. Note that for the `Account` class we have overridden the semantical (logical) equality. The operator for checking syntactical equality is `===`.

5.7 The Comparable Interface

Assume, now, that we would like to use the class `Account` as a key in a dictionary to store integers. We enhance the `Account` class, to allow comparison between two `Account` objects. The comparison method is very simple: in order to compare two accounts, we simply compare their corresponding holder names. This means that two accounts are considered the same, if they have the same holder. As a result, in the dictionary, we cannot have two different `Account` keys with the same holder.

In order to do this, we implement the `Comparable` interface of COMET, and rely on the `compare` method for `string` objects, for returning the desired comparison value (i.e., the result of a lexicographic comparison between the holder names). The enhanced version of the class is shown in the following COMET code:

```
class Account implements Comparable {
    int amount;
    string name;
    Account(int initial,string holder) {
        amount = initial;
        name = holder;
    }
    void deposit(int value) {
        assert(value > 0);
        amount += value;
    }
    void withdraw(int value) {
        amount -= value;
    }
    int getAmount() {
        return amount;
    }
    int compare(Comparable account) {
        Account acc = (Account) account;
        return name.compare(acc.getHolder());
    }
    string getHolder(){
        return name;
    }
}
```

The following example illustrates the behavior by adding some `Account` instances as keys inside a dictionary:

```
Account acc1(10,"pierre");
Account acc2(20,"yannis");
Account acc3(30,"pierre");

dict{Account->int} mydict();
mydict{acc1} = 3;
mydict{acc2} = 4;
mydict{acc3} = 4;

forall (acc in mydict.getKeys())
    cout << acc.getHolder() << ":" << acc.getAmount() << endl;
cout << "acc1 vs acc2 ? :" << acc1.compare(acc2) << endl;
cout << "acc1 vs acc3 ? :" << acc1.compare(acc3) << endl;
```

It produces the output:

```
pierre:10
yannis:20
acc1 vs acc2 ? :-9
acc1 vs acc2 ? :0
```

Indeed, there is already an account with the name of `pierre`, so it will not be replaced in the dictionary. In general, the method `compare(Comparable obj)` should return a negative integer to indicate that *this* is smaller than the object `obj`, 0 to indicate that they are equal, and a positive integer to indicate that *this* is larger. In our implementation, we rely on the implementation of method `compare` for strings.

Note that the `Comparable` interface can also be useful, in order to have a specific behavior with the `sortPerm` method, or to use user-defined objects as keys in a `heap` data structure (see Section 3.7).

As illustrated next, the `sortPerm` method allows to lexicographically sort (according to their holder names) an array of `Account` objects:

```
Account acc1(10,"pierre");
Account acc2(20,"yannis");
Account acc3(30,"pierre");
Account acc4(10,"stephane");
Account acc5(20,"alessandro");
Account acc6(30,"andrew");

Account accs[1..6] = [acc1,acc2,acc3,acc4,acc5,acc6];
int[] perm = accs.sortPerm() ;
cout << all(i in 1..6) accs[perm[i]].getHolder() << endl;
```

This produces the output:

```
anonymous[alessandro,andrew,pierre,pierre,stephane,yannis]
```

5.8 Pretty Print of Objects

Objects can be printed to the standard output with the operator `cout <<`. The printing behavior can be specified by overloading the method `void print(ostream os)`, as shown in the following example (in a similar fashion with the `toString` method of Java):

```
class Foo {
    int _v;
    Foo(int v) {_v = v;}
    void print(ostream os) {
        os << "my value is: " << _v << endl;
    }
}
Foo foo(12);
cout << foo << endl;
```

The output of this code is:

```
my value is: 12
```

5.9 Exceptions

In COMET, there is an exception mechanism similar to the one of Java. Any COMET object can be thrown as an exception, and the exception throw can be surrounded with the classical `try/catch`, as shown in the following example:

```
class Error {
    string _msg;
    Error(string msg) {_msg = msg;}
    void print() {
        cout << _msg << endl;
    }
}

try {
    throw Error("this is an error");
}
catch (Error e) {
    e.print();
}
```

The above code produces the output:

```
this is an error
```

Chapter 6

Closures and Continuations

Closures and continuations are present in several abstractions offered by COMET (e.g., local search neighborhoods, events, search controllers), although they are generally hidden from the user. Still, getting an understanding of these abstractions can be very valuable to COMET users. Closures and continuations are first-class objects in COMET. This means that they can be stored in variables, or passed as parameters to functions and methods. This chapter is giving a first introduction to the use of closures and continuations in COMET.

6.1 Closures

A closure is a piece of code together with its environment. Closures are ubiquitous in functional programming languages, but are rarely supported in mainstream object-oriented languages (SMALLTALK is a notable exception). To illustrate the use of closures, consider the following class:

```
class DemoClosure {  
    int v;  
    DemoClosure() {}  
    void setVal(int val) { v = val; }  
    Closure print(int i) {  
        return closure {  
            cout << i << ", " << v << endl;  
        };  
    }  
}
```

The class `DemoClosure` has a basic (empty) constructor and a method `print`, that receives an integer `i` and returns a closure. This closure, when executed, prints `i` to the standard output. Observe that, when the method `print` is executed, it does not execute the closure and hence produces no display. The following code snippet shows how to use closures in COMET:

```
DemoClosure demo();  
demo.setVal(2);  
Closure c1 = demo.print(6);  
demo.setVal(7);  
Closure c2 = demo.print(4);  
call(c2);  
call(c1);
```

This gives the output:

```
4, 7  
6, 7
```

The first line declares an instance `demo` of the class `DemoClosure`, which is then used to collect two closures `c1` and `c2`. These closures are then executed by applying the `call` function. Note that the environment captured by a closure is the stack. Instances of objects are located in the heap, and, hence, instance variables are also located there, and not in the stack. This is why the value of `v` in the class `DemoClosure` is not captured in the environment of the closure. There are two possible ways to declare a closure as illustrated in the following example:

```
int i = 3;
closure c11 {
  cout << i << endl;
}
Closure c12 = closure {
  cout << i << endl;
};
```

6.2 Continuations

Informally speaking, a continuation is a snapshot of the runtime data structures, so that execution can restart from this point at a later stage of computation. More precisely, a continuation is a pair $\langle I, S \rangle$, where I is an instruction pointer and S is a stack that can be used to execute the code starting at I .

Continuations are low-level control structures, that can be used to implement a variety of higher-level abstractions, such as exceptions, co-routines, and non-determinism. Continuations are obtained in COMET through instructions of the form

```
continuation c <body>
```


Such an instruction binds `c` to a continuation $\langle I, S \rangle$, where I is the next instruction in the code and S is the stack when the continuation is captured. It then executes its body and continues in sequence. The resulting continuation can be invoked with

```
call(c)
```

which would restore stack S and restart execution from instruction I . Consider the following code:

```
1. function int fact(int n) {  
2.   if (n == 0)  
3.     return 1;  
4.   else  
5.     return n * fact(n-1);  
6. }  
7. int i = 4;  
8. continuation c { i = 5; }  
9. int r = fact(i);  
10. cout << "fact(" << i << ") = " << r << endl;  
11. if (i == 5)  
12.   call(c);
```

It outputs:

```
fact(5) = 120  
fact(4) = 24
```

Indeed, the continuation `c` in line 8 consists of an instruction pointer to line 9 and a stack, whose entry for `i` stores the value 4. The COMET implementation first calls the factorial function with argument 5 (since `i = 5` is executed when the continuation is taken). Since `i` has value 5, the implementation calls the continuation (line 10), which restarts execution from line 9 with a stack, whose entry for `i` has value 4. The COMET implementation thus calls `fact(4)`, displays its result, and terminates (since `i` is 4).

Chapter 7

Events

Events are a particularly useful abstraction, that allows to decompose different parts of an implementation. Typical uses of events in `COMET` are

- to decouple visualization of an algorithm from its implementation
- to decouple hyper-heuristic strategies from move selection in Constraint Based Local Search. For instance, restarts can be implemented using events.

In general, events can improve code readability by making it more modular. Many events are already defined in `COMET`, but there is also the option for user-defined events. The most useful events are defined for local search variables (`var{int}`), constraint programming variables (`var<CP>{int}`) and visualization objects.

7.1 Simple Events

We describe the event functionality of COMET through an example of a user-defined counter class `Count`. The class implementation declares a kind of event named `Event changes(int ov,int nv)`, having two integer parameters, that correspond to the old and the new value of the counter. Then, in the implementation of method `increment`, we notify all the observers about the change of the counter's value:

```
class Count {  
    int x;  
    Event changes(int ov,int nv);  
    Count() { x = 0; }  
    int getVal() { return x; }  
    int increment() {  
        notify changes(x,x+1);  
        x++;  
        return x;  
    }  
}
```

An observer can subscribe to the notifications of this event on a particular instance `c` of class `Count` and specify a block of instructions, that should be executed every time the event `changes(int ov,int nv)` is notified on instance `c`:

```
Count c();  
whenever c@changes(int ov,int nv) {  
    cout << "old val:" << ov << " new val:" << nv << endl;  
}
```

Finally the next two instructions

```
c.increment();  
c.increment();
```

produce the output

```
old val:0 new val:1  
old val:1 new val:2
```

Caveat A common logical error arises from modifying an `int` inside the body of a **whenever**. The block to be executed, when an event occurs, is captured inside a closure. Since a closure only captures the stack of the environment, no modification will happen to the `int`. In order to retain changes to variables defined in that block, we have to use the object version `Integer` instead of an `int`. The following code fragment illustrates this phenomenon:

```
int nb = 0;  
Integer NB(0);  
whenever c1@changes(int ov,int nv){  
    nb++;  
    NB := NB+1;  
}  
c1.increment();  
c1.increment();  
cout << "nb:" << nb << " NB:" << NB << endl;
```

This produces the output

```
nb:0 NB:2
```

As we can see, only the `Integer` object `NB` “remembers” increases to its value, thus exhibiting the intended behavior.

7.2 Key Events

In certain cases, we don't want to take action at *every* change of an object associated with an event, but only *once*, when the object is assigned a particular value. Let us consider the **Count** class again. Assume we want to execute a block of code, when a counter *x* reaches the value 2. A way to do this is by using a **whenever** block, as shown in the following block of code:

```
Count c1();
Boolean B(true);
whenever c1@changes(int ov, int nv) {
    if (nv>=2 && B){
        cout << "x is now " << c1.getVal() << endl;
        B := false;
    }
}
c1.increment();
c1.increment();
c1.increment();
```

This generates the output:

```
x is now 2
```

However, this is not the most efficient way to proceed. A more efficient way is through the use of *key events*. We show how this can be done in our example. We enhance the **Count** class to declare a **KeyEvent**, that triggers an event-handler associated with it only once. More precisely, when the value of the event first reaches the value specified in the parameter of **when** in the event-handler, the closure of the handler is called.

Here is the updated implementation of class `Count`:

```
class Count{
    int x;
    KeyEvent reaches();
    Count(){ x = 0; }
    int getVal() { return x; }
    int increment() {
        x++;
        notify reaches[x]();
        return x;
    }
}
Count c1();
when c1@reaches[2]() {
    cout << "x is now " << c1.getVal() << endl;
}
c1.increment();
c1.increment(); // triggers execution of the when block
c1.increment();
c1.increment();
```

The produced output is:

```
x is now 2
```

Comet Counters Note that COMET offers a built-in counter functionality through the class `Counter`, which supports a `KeyEvent reaches`. A simple example of using this class is:

```
Solver<LS> ls();
Counter it(ls);
when it@reaches[5]()
    cout << it << endl;
it := 2;
forall (i in 1..100)
    it++;
```

The above code fragment produces the output:

```
5
```


Chapter 8

Concurrency

COMET provides support for concurrency programming at multiple levels of abstraction. At its core, the support is embodied by POSIX style synchronization primitives and abstractions. COMET builds upon this foundation to provide monitors and parallel control abstractions. This chapter reviews the various primitives and demonstrates how they can be used to build parallel programs.

Section [8.1](#) provides an overview of computation threads while section [8.2](#) reviews the synchronization primitives. Section [8.3](#) describes unbounded parallel loops while section [8.4](#) shows how to control bounded parallelism. Section [8.5](#) describes the mechanisms needed for interrupting parallel computations.

8.2 Synchronization

Running a parallel computation mandates the ability to synchronize the computations performed by several threads of control within the application. COMET provides a number of synchronization primitives to achieve this result. Each construction is reviewed next.

Mutual exclusion COMET provides two classes `Mutex` and `RMutex` to *manually* support mutual exclusion. The `Mutex` class has the following API

```
native class Mutex {  
    Mutex();  
    void lock();  
    void unlock();  
}
```

The two `lock` and `unlock` methods can be used to, respectively, enter and leave a critical section, i.e., a section of code that accesses a shared data structure, so that the competing computation threads do not concurrently read or write to the structure. For instance, if one wishes to maintain a shared counter (an integer) and ensure that any number of concurrent read or write operations leave the structure in a consistent state and that no writes are lost, it is necessary to use a mutex. The fragment below illustrates this idea

```
Mutex m();  
int counter = 0;  
...  
m.lock();  
counter++;  
m.unlock();
```

An instance of the `RMutex` class implements a *recursive* mutex, i.e., it not only supports mutual exclusion but also allows any given thread to repeatedly lock the same mutex without inducing a deadlock (which would occur with the standard mutex). A recursive mutex is particularly helpful, when a sequence of method invocations leads a thread to re-enter the critical section, while already executing inside that same critical section.

A naive (and arguably contrived) example illustrates this situation with a recursive method inside a counter class:

```
class Counter {
    int    c;
    RMutex m;
    Counter() {
        c = 0;
        m = new RMutex();
    }
    void increaseBy(int x) {
        if (x==0)
            return;
        m.lock();
        c++;
        increaseBy(x-1);
        m.unlock();
    }
}
```

Observe how the method `increaseBy` recursively calls itself, while still holding onto the mutex `m`. Given that the mutex is recursive, the second (re-entering) call to `lock` by the thread t currently holding `m` will succeed and grant the lock a second time to t . Naturally, t is expected to call `unlock` as many times as it has called `lock` and a different thread of computation, say t' , won't be authorized to enter the critical section until t has fully unlocked `m`.

Conditions COMET supports POSIX-style conditions with the class `BCondition`. Boolean conditions provide a mechanism to suspend the execution of a computation thread, until a boolean condition is satisfied. The API of the `BCondition` class is shown below:

```
native class BCondition {  
    BCondition();  
    void wait(Mutex m);  
    void notice();  
    void broadcast();  
}
```

The two core operations supported by a POSIX condition are (1) putting a thread (that currently holds a mutex protecting a critical section) to sleep, while releasing the mutex, and; (2) waking up a slumbering computation thread, when the boolean condition is susceptible to have changed, in which case the waking thread automatically re-acquires the mutex, before proceeding with its execution.

The `wait` method is used by a thread to willingly suspend its execution and *atomically* relinquish the mutex it currently holds. The behavior is undefined, if the mutex is not locked. When this thread wakes up at a future point in time, it will re-acquire the mutex `m`, before proceeding and returning from its call to `wait`.

The `notice` and `broadcast` methods are used by a computation thread to notify, either one or all, slumbering threads that the boolean conditions that they are observing has potentially changed state and might now be true. Note that the POSIX semantics do not guarantee that the boolean will be true upon wake up. It is the responsibility of the waking thread to re-check the boolean conditions before proceeding.

This gives rise to the traditional pattern shown below, that demonstrates a hand-made producer-consumer buffer class

```
class PCBuffer {
    queue<string> buf;
    Mutex        m;
    BCondition    full;
    BCondition    empty;
    PCBuffer() {
        buf    = new queue<string>(10); // a queue of maximum 10 elements.
        m      = new Mutex();
        full   = new BCondition();
        empty  = new BCondition();
    }
    void produce(string s) {
        m.lock();
        while (buf.getSize() >= 10)
            full.wait(m);
        buf.enqueue(s);
        empty.notice();
        m.unlock();
    }
    string consume() {
        m.lock();
        while (buf.getSize() == 0)
            empty.wait(m);
        string s = buf.dequeue();
        full.notice();
        m.unlock();
        return s;
    }
}
```

The producer-consumer buffer provides two methods to add and retrieve strings from the shared buffer. Naturally, both methods use a mutex to protect the accesses to the shared buffer. First, let us assume that the buffer is currently full (i.e., it already contains 10 strings). A thread calling the **produce** method first acquires the lock and proceeds with a check of the buffer size. Given that the buffer is full, it cannot deposit the additional string and is compelled to fall to sleep with a call

to **wait** on the boolean condition **full**. Blocking on the condition releases the mutex **m** atomically, which enables another thread to invoke a method of the **PCBuffer** class.

If a second thread then invokes the **consume** method, it acquires the mutex and proceeds with a check on the buffer size. Given that the buffer is not empty (the size is different from 0), it carries on, dequeues a string and then *notifies* any potentially waiting producer that the buffer now has an empty slot. It finally releases the mutex and returns the string. At that point, any slumbering producer can wake up, re-acquire the lock and initiate another check of the **full** condition. If there is still space, it carries on and enqueues its argument string, before notifying any waiting consumer, and then releases the mutex.

Barriers Barriers are designed to offer a rendez-vous between a fixed number of computation threads, that must all meet, before the entire group is allowed to proceed with the rest of the computation. Conceptually, a barrier has a very simple API with only two operations

```
native class Barrier {  
    Barrier(int k);  
    void enter();  
}
```

The constructor creates a **Barrier** with a fixed size k . The **enter** method is used by a computation thread to enter the barrier. If the barrier was created for k threads, the first $k - 1$ threads entering the barrier all go to sleep. The last (k^{th}) thread entering the barrier causes all the threads to wake up and exit the barrier (they all return from their own calls to the **enter** method). Note that threads might be re-entering the very same barrier, before all threads have exited. This is not a problem and corresponds to two separate occurrences of the meeting. The threads that are entering in the second “round” are quite distinct from those exiting the current round, and will have to wait until k threads enter the second round before proceeding.

Monitors While mutex and conditions are usable on their own merit, the vast majority of the situations encountered in practice can be dealt with with a slightly more abstract concept of *monitors*, as found in ADA and more recently in JAVA.

Monitors are classes whose methods are automatically synchronized. Entering a method in a monitor automatically triggers the acquisition of a (hidden) lock associated with the receiver. Leaving the method triggers the release of the same (hidden) lock. In COMET a monitor can easily be created with the **synchronized** keyword.

For instance, the following code

```
synchronized class Count {  
    int x;  
    Count(int _x) { x = _x; }  
    void increase() { x++; }  
    void decrease() { x--; }  
    int get() { return x; }  
}
```

creates a monitor named `Count`. Its methods are completely traditional with the exception of the implicit locking and unlocking that will take place for every instances of the class.

Naturally, one might wish to use POSIX-style conditions with monitors. This is also possible. The keyword `Condition` can be used inside a class definition to declare a condition attribute for the class. This condition attribute can work with the (hidden) mutex to provide the same level of control, as described earlier in the producer-consumer buffer. The example is rewritten below in this lighter style:

```
synchronized class PCBuffer {  
    queue{string} buf;  
    Condition full;  
    Condition empty;  
    PCBuffer() { buf = new queue{string}(10); }  
    void produce(string s) {  
        while (buf.getSize() >= 10) full.wait();  
        buf.enqueue(s);  
        empty.signal();  
    }  
    string consume() {  
        while (buf.getSize() == 0) empty.wait();  
        full.signal();  
        return buf.dequeue();  
    }  
}
```

8.3 Unbounded parallel loops

The `parall` instruction is the parallel variant of the `forall` instruction as it will execute all iterations of the `forall` loop in parallel. For instance, the fragment

```
parall (i in 1..10) {  
    int k = 0;  
    forall (i in 1..1000000)  
        k++;  
    cout << "thread" << i << " " << k << endl;  
}  
cout << "done" << endl;
```

executes the body of the `parall` loop 10 times in 10 threads. It can be understood in term of the following rewrite:

```
ZeroWait b();  
forall (i in 1..10) {  
    b.incr();  
    thread t {  
        int k = 0;  
        forall (i in 1..1000000)  
            k++;  
        cout << "thread" << i << " " << k << endl;  
        b.decr();  
    }  
}  
b.wait();  
cout << "done" << endl;
```

which places the body of each iteration in a separate thread and uses a barrier to synchronize them and ensure that the execution of the main control thread does not proceed past the `parall` instruction to the final print out, until all the iterations have completed.

The output of the code above might be:

```
thread7 1000000
thread4 1000000
thread3 1000000
thread2 1000000
thread1 1000000
thread5 1000000
thread6 1000000
thread9 1000000
thread8 1000000
thread10 1000000
done
```

8.4 Bounded parallel loops

While the `parall` instruction provides the ability to easily execute code in parallel, it does not take into account the inherent degree of parallelism of the underlying hardware. For instance, if the parallel machine has 4 processors, spawning more than 4 concurrent threads is typically harmful, as the threads are competing for the scarce CPU resources and causing unneeded contention and context switching.

Instead, it is preferable to limit the amount of concurrency to the number of processors. Therefore, COMET offers the concept of bounded parallel loops and thread pools.

Thread Pool A thread pool is an object modeling a collection of execution agents that are at the disposal of a parallel control instruction to execute code. A thread pool with 4 threads is easily created with the instruction

```
ThreadPool pool(4);
```

COMET can also lookup, via a method call, the number of CPU on the machine to automatically adapt to the underlying hardware

```
ThreadPool pool(System.getNCPUS());
```

Bounded loops If we want to limit the number of concurrent threads, it suffices to parameterize the `parall` loop with the `ThreadPool` instance. The effect of this is that all the iterations of the loop are created and scheduled to run on the threads present in the pool.

Consequently, the number of concurrent iterations of the loop can never exceed the size of the thread pool. For instance, in the following fragment

```
ThreadPool pool(4);
parall<pool> (i in 1..10) {
  int k = 0;
  forall (i in 1..1000000)
    k++;
  cout << "thread" << i << " " << k << endl;
}
pool.close();
cout << "done" << endl;
```

the 10 iterations of the `parall` loop are dispatched on the four threads in the `pool`.

8.5 Interruptions

Quite often, it is desirable to end a parallel computation, as soon as one of the computing threads has found a suitable solution. The situation naturally arises in parallel local search, where multiple independent searches are conducted by separate threads. It also occurs, when using randomized searches on models defined with complete solvers and where the intent is to stop with the first feasible solution. The fragment below shows that a simple addition to the basic `parall` loop suffices to easily support *interruptions*.

```
Boolean found = false;
Model model = ...;
ThreadPool pool(System.getNCPUS());
parall<pool> (i in 1..100) {
    Solution s = model.search();
    found := s.getValue() == 0;
} until found;
pool.close();
```

The core structure of the earlier examples is retained and consists of a bounded parallel loop. However, any iteration is now at liberty to solve a model and produce a solution. In the case of a feasibility problem, the *value* of a solution indicates the violation degree of the solution, i.e., a feasible solution has no violations and its value is therefore 0. Observe in the above how the boolean `found` is set to true, if and only if the produced solution has no violations.

The `parall` instruction ends with an `until found` clause, that specifies that the looping should end, as soon as the boolean becomes true. When this occurs, COMET automatically takes care of cancelling the threads that might still be running and guarantees that no further iterations will be submitted to the pool. The execution will resume sequentially after the `parall` loop, as soon as all the threads are returned to the pool.

8.6 Events

The event mechanics described in the previous chapter are fully functional in the context of concurrent programming. The semantics of event dispatching are merely slightly refined, to account for the specificities of threads.

Event Notifications The event notification mechanism is entirely unaffected by threads. Any thread can notify an event on any object and the event *listeners* will be called as usual. The notifications can occur from any thread at any time.

Event listeners Recall that an event listener is created with a **when** or a **whenever** instruction. For instance, the statements

```
class Count {
    int x;
    Event changes(int ov,int nv);
    Count() { x = 0; }
    int getVal() { return x; }
    int increment() {
        notify changes(x,x+1);
        x++;
        return x;
    }
}

Count c();
whenever c@changes(int ov,int nv)
    cout << "old val:" << ov << " new val:" << nv << endl;
forall (k in 1..1000)
    c.increment();
```

create a **Count** class with a change event that is triggered whenever its **increment** method is called. The **whenever** statement appearing at the bottom produces a line of output each time the **changes** event is raised from a subsequent call to **increment**.

This code can be modified, so that the calls to the **increment** method occur from different threads. For instance, the following refined code fragment

```
synchronized class Count {
    int x;
    Event changes(int ov,int nv);
    Count() { x = 0; }
    int getVal() { return x; }
    int increment() {
        notify changes(x,x+1);
        x++;
        return x;
    }
}

Count c();
whenever c@changes(int ov,int nv) {
    cout << "old val:" << ov << " new val:" << nv << endl;
}

ThreadPool tp(4);
parall<tp> (k in 1..1000)
    c.increment();
tp.close();
```

states that the *main thread* listen to the **changes** events, whereas a bounded parallel loop uses four threads to schedule the 1000 iterations (and therefore calls to **increment**) on the shared instance.

Note that concurrent calls to increment from different threads are automatically synchronized, given that the **Count** class is a monitor. However, while the occurrences of **changes** events are raised by four different threads, all the handlers execute inside the thread that requested the listening, i.e., in this case, the main thread.

8.7 Further Notes

This chapter covered the basic syntax and semantics of parallel control abstractions. The support for parallel computation extends beyond these primitives, with specific support for the parallelization of the finite domain solver (which, of course, builds upon the primitives described here). The topic will be revisited in a subsequent chapter (17) in the constraint programming part (II) of this manual.

II CONSTRAINT PROGRAMMING

Chapter 9

Introduction to Constraint Programming

This chapter is giving an introduction to the main concepts of Constraint Programming (CP). Constraint Programming can be characterized pretty well by the equation:

$$CP = Model + Search$$

A CP model looking for a feasible solution has the following structure:

```
import cotfd;
Solver<CP> cp();
//declare the variables
solve<cp> {
    //post the constraints
}
using {
    //non deterministic search
}
```

Note the clear separation between the modeling part, that declares the variables and posts the constraints, and the search part. To enumerate *all* feasible solutions, the keyword `solve` must be replaced by `solveall`. More details on each of these parts are given in the sections to follow.

9.1 Variables

The first step in modeling a problem is to declare the variables. Discrete integer variables, also called *finite domain* integer variables (f.d. variables), are the most commonly used. A CP f.d. variable must be given a CP solver and a domain. The domain of a variable specifies the set of values that can be assigned to that variable.

The following example declares a variable `x` with the integer interval domain `[1..10]`.

```
var<CP>{int} x(cp,1..10);
```

The domain of a variable can also be given as a set of integers:

```
set{int} dom = {1,3,7};  
var<CP>{int} y(cp,dom);
```

It can also be defined on an enumerated type, to obtain more readable programs or solutions.

```
enum country = {Belgium, USA, France, Greece, China, Spain};  
var<CP>{country} z(cp,country);
```

Besides f.d. integer variables, COMET also allows to declare boolean and float variables:

```
var<CP>{bool} b(cp);  
var<CP>{float} f(cp,1,5);
```

In the above fragment, the domain of boolean variable `b` is `{true,false}` and the domain of rational variable `f` is the rational interval `[1,5]`. Note that, currently, there are no constraints defined on float variables and they should only be used in sum expressions (for instance in an objective), not as decision variables.

9.2 Constraints

Constraints act on the *domain store* (the current domain of all variables) to remove inconsistent values. Behind every constraint, there is a sophisticated filtering algorithm, that prunes the search space by removing values that don't participate in any solution satisfying the constraint.

The domain store is the only possible way of communication between constraints: whenever a constraint C_1 removes a value from the domain store, this triggers the detection of a possible inconsistent value for another constraint C_2 . This inconsistent value is in turn removed, and this propagates, until reaching a *fixed point*, which means that no constraint can remove a value. This is the basic idea of the *fixpoint* algorithm. As soon as a variable's domain becomes empty, the domain store fails, meaning that there is no possible solution, and the search has to backtrack to a previous state and try another decision.

To summarize, a constraint must implement two main functionalities:

- Consistency Checking: verify that there is a solution to the constraint, otherwise tell the solver to backtrack
- Domain Filtering: remove inconsistent values, i.e., values not participating in any solution

A constraint can be posted to the CP solver with the `post` method. The constraint must always be posted inside a `solve{}`, `solveall{}` or `such that{}` block, otherwise COMET does not guarantee the results. The following example posts the constraint that variables `x` and `y` must take two different values.

```
cp.post(x != y);
```

COMET allows posting constraints that involve complex arithmetic and logical expressions, as in the following example. Note that `(x[i]==2)` is reified to a 0/1 variable and that `=>` posts a logical implication constraint that needs to be `true`.

```
cp.post(sum(i) (x[i]==2) * size[i] == totSize);
cp.post(x>y => a<b);
```

It is possible to index an array (of constants or variables) by a variable. This is called an **element** constraint, and is a specificity of Constraint Programming. For example, consider the following constraint, in which **z** is an array of f.d. variables and **x** and **y** are also f.d. variables.:

```
cp.post(z == x[y]);
```

For some constraints, it may be more efficient to post them globally, as one big constraint, rather than decomposing them into a conjunction of smaller constraints. For example, consider an array of f.d. variables. The constraint that each variable in the array takes a different value can be decomposed into a set of binary constraints, as follows:

```
forall (i in x.getRange(), j in x.getRange(): j>i)
  cp.post(x[i]!=x[j]);
```

The same constraint can also be stated globally as a single **alldifferent** constraint:

```
cp.post(alldifferent(x));
```

A large number of constraints is already implemented in COMET. Chapter [10](#) contains a detailed list of most of the available constraints and describes the procedure of posting a constraint.

9.3 Search

Search in constraint programming consists in a non-deterministic exploration of a tree with a back-tracking search algorithm. The default exploration algorithm is Depth-first Search. Other search strategies available are: Best-First Search, Bounded Discrepancy Search, Breadth-First Search, etc. The following example explores all combinations of three 0/1 variables using a depth-first strategy:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
solveall<cp> {
}
using {
    label(x);
    cout << x << endl;
}
```

This block of code produces the output:

```
x[0,0,0]
x[0,0,1]
x[0,1,0]
x[0,1,1]
x[1,0,0]
x[1,0,1]
x[1,1,0]
x[1,1,1]
```

Search in COMET can be deterministic, which means that, at each node of the search tree, you have the possibility to control the sub-trees under that node. The following example illustrates how to reproduce the same behavior as in the previous example by explicitly creating, at each node, a left and a right branch on the search tree through the use of the `try` command:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
solveall<cp> {
}
using {
  forall (i in x.getRange())
    try<cp> cp.label(x[i],0); | cp.label(x[i],1);
  cout << x << endl;
}
```

One can also randomize the decisions taken at each node. The following example randomly decides, at each node, whether to assign 0/1 to the left or the right branch:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
UniformDistribution distr(0..1);
solveall<cp> {
}
using {
  forall (i in x.getRange()) {
    int v = distr.get();
    try<cp> cp.label(x[i],v); | cp.diff(x[i],v);
  }
  cout << x << endl;
}
```

The next variable to instantiate can also be decided on the fly, at each choice point. The following example uses a randomized selector `select` (see Section 4.2) to choose the index of the next variable to instantiate.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
UniformDistribution distr(0..1);
solveall<cp> {
}
using {
  while(!bound(x)) {
    select(i in x.getRange():!x[i].bound()) {
      int v = distr.get();
      try<cp> cp.label(x[i],v); | cp.diff(x[i],v);
    }
  }
  cout << x << endl;
}
```

9.4 Testing if a Solution Was Found

At the end of search, it can be useful to test if the a solution was found before attempting to print it. The method `getSolution()` can be useful to this end, as illustrated in the next example, which outputs “no solution”:

```
import cotfd;
Solver<CP> cp();
range n = 1..2;
var<CP>{int} x(cp,0..1);
solve<cp> {
    cp.post(x!=0);
    cp.post(x!=1);
}
using {
    label(x);
}
if (cp.getSolution() != null) {
    cout << "solution with x=" << x << endl;
}
else {
    cout << "no solution" << endl;
}
```

9.5 The Queens Problem: Hello World of CP

The classical toy problem of Constraint Programming is the n -queens problem, in which n queens must be placed on a $n \times n$ chess board, so that they do not attack each other. The complete code to find a solution to the n -queens problem is given in Statement 9.1. It declares one f.d. variable for each row of the board: $q[i]$ is the variable representing the column of the queen in row i .

The constraints

```
cp.post(alldifferent(all(i in S) q[i] + i));
cp.post(alldifferent(all(i in S) q[i] - i));
```

specify that no two queens can be on the same diagonal.

Notice the array comprehension `all(i in S) q[i] + i`, which builds on the fly an array of shifted f.d. variables, that is given as parameter to the `alldifferent` constraint.

The constraint

```
cp.post(alldifferent(q));
```

specifies that no two queens can be placed on the same column.

The search used to find a solution is a default first-fail heuristic `labelFF(q)` :

- The next variable to select is the one with the smallest domain
- The values are tried in increasing order

The call to function `labelFF(q)` is completely equivalent to the following alternative implementation:

```
forall (i in S) by (q[i].getsize())
  tryall<cp>(v in S) cp.label(q[i],v);
```

```
import cotfd;
Solver<CP> cp();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](cp,S);
solve<m> {
    cp.post(alldifferent(all(i in S) q[i] + i));
    cp.post(alldifferent(all(i in S) q[i] - i));
    cp.post(alldifferent(q));
}
using {
    labelFF(q);
}
cout << cp.getNFail() << endl;
```

Statement 9.1: CP Model for the Queens Problem

9.6 Optimization in CP

Constraint Programming can solve optimization problems by minimizing or maximizing complex expressions or variables. This is done as follows: Every time a solution is found during the search, a constraint is dynamically added, stating that the next solution found must be better. By construction, the last solution found in this way will be an optimal solution.

The structure of a COMET program minimizing an expression is the following (replace **minimize** with **maximize** for maximization):

```
import cotfd;
Solver<CP> cp();
//declare the variables
minimize<cp>
    //expression or variable
subject to {
    //post the constraints
}
using {
    //non deterministic search
}
```

Minimization in CP is illustrated through the Balanced Academic Curriculum Problem (BACP). In this problem, the goal is to place courses into periods, so that we balance the workload of each period, while satisfying prerequisite relationships between courses and requirements on the minimum/maximum number of courses placed in each period. The balance objective is attained by minimizing the maximum workload.

The data for the problem consists of the number of periods, the **credits** of each course and the **prerequisites**. The interpretation of the **prerequisites** array is that course 7 must be placed before course 1, course 8 before course 2, etc. The values **minCard** and **maxCard** are the minimum and maximum number of courses that can be assigned to each period.

```

int    nbCourses = 66;
range  Courses = 1..nbCourses;
int    nbPeriods = 12;
range  Periods = 1..nbPeriods;
int    nbPre = 65;
range  Prerequisites = 1..nbPre;
int    credits[Courses] = [1,3,1,2,...];
int    totCredit = sum(i in Courses) credits[i];
int    prerequisites[1..2*nbPre] = [7,1,8,2,8,6,...];
int    minCard = 5;
int    maxCard = 6;

import cotfd;
Solver<CP> cp();
var<CP>{int} x[Courses](cp,Periods);
var<CP>{int} l[Periods](cp,0..totCredit);
minimize<cp>
    max(i in Periods) l[i]
subject to {
    cp.post(multiknapsack(x,credits,l));
    forall (i in Prerequisites)
        cp.post(x[prerequisites[i*2-1]]<x[prerequisites[i*2]]);
    cp.post(cardinality(all(p in Periods)(minCard),x,all(p in Periods)(maxCard)));
}
using {
    labelFF(x);
}
cout << "loads:" << l << endl;

```

Statement 9.2: CP Model for the Balanced Academic Curriculum Problem (bacp-cp.co)

The global constraint

```
cp.post(multiknapsack(x,credits,1));
```

ensures that, for each period p :

$$l[p] = \text{sum}(c \text{ in Courses}) (x[c]==p) * \text{credit}[c]$$

The prerequisite relations are then simply posted one by one with:

```
forall (i in Prerequisites)
  cp.post(x[prerequisites[i*2-1]] < x[prerequisites[i*2]]);
```

Finally the `cardinality` global constraint ensures that, for every period p :

$$\text{minCard} \leq \text{sum}(c \text{ in Courses}) (x[c]==p) \leq \text{maxCard}$$

```
cp.post(cardinality(all(p in Periods)(minCard),x,all(p in Periods)(maxCard)));
```

Note that we could also give the `onDomains` parameter to the `cardinality` global constraint, if we wanted domain consistent filtering. The search used is a standard first-fail heuristic `labelFF`. The First-Fail default heuristic is not the best-choice, though, for this optimization problem. The problem with `labelFF` is that there is no value heuristic that might drive the search rapidly towards good feasible solutions, with respect to the objective function. Finding a good solution early on is important, since then the problem becomes constrained rapidly and the search tree gets smaller.

By following the value heuristic of `labelFF` for the Balanced Academic Curriculum Problem, at each node, the chosen course will be placed in the first available period. To quickly find a good solution to the problem, it is better to place the chosen course in the currently least loaded period. To increase the chances of satisfying the cardinality constraint, we can break ties by preferring the period with the smallest number of courses. For better placement, we can further improve the First-Fail criterion by breaking ties with the number of course credits, since the placement of a large course has more impact on the objective.

The improved search for this problem, replacing the `labelFF(x)` command in the **using** block, is the following:

```
forall (i in Courses: !x[i].bound()) by (x[i].getSize(), -credits[i]) {  
  int nbCourses [p in Periods] = sum(j in Courses: x[j].bound()) (x[j]==p);  
  tryall<cp>(p in Periods: x[i].memberOf(p)) by (l[p].getMin(), nbCourses[p])  
    cp.label(x[i], p);  
}
```

Chapter 10

List of Constraints

This chapter provides a complete list of the constraints currently implemented into COMET. For each constraint, we detail the consistency levels available, we give an example of assignment that satisfies it and, when possible, refer to examples in the tutorial that use the constraint. Note that, the current chapter does not cover set variable constraints. These are presented in full detail in [Chapter 11](#), that reviews set variable support in COMET's Constraint Programming module. Before getting into the list of constraints, we take a moment to describe the correct way of posting constraints.

10.1 Posting a Constraint

A constraint `myconstraint` is generally posted in the `solve{...}` or `subject to{...}` block by calling a function having a signature like:

```
Constraint<CP> myconstraint(some variables parameters)
```

The constraint is added (posted) to the model with the `post` method on the `Solver<CP>` object:

```
import cotfd;
...
Solver<CP> cp();
...
solve {
    cp.post(myconstraint(some variables parameters) [,Consistency<CP> cl]);
    ...
}
...
```

There are two important issues related to the `post` instruction:

- the fixpoint algorithm, that start at every `post` instruction
- the consistency level (`cl`), an optional argument of the `post` method

We are going to discuss these in the following sections.

10.2 The Fixpoint Algorithm

When a constraint is posted, the fixpoint algorithm is immediately applied. This means that all the constraints posted until then, will participate in the filtering of the domains, one constraint after the other, until no more values can be removed from any of the domains. The following example and its output clearly illustrate the behavior of the fixpoint and propagation mechanism of COMET.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x(cp,1..3);
var<CP>{int} y(cp,1..3);
var<CP>{int} z(cp,1..3);
solve<cp> {
    cout << x << y << z << endl;
    cp.post(x<y);
    cout << x << y << z << endl;
    cp.post(y<z);
    cout << x << y << z << endl;
}
```

The output is:

```
(3) [1..3] (3) [1..3] (3) [1..3]
(2) [1..2] (2) [2..3] (3) [1..3]
123
```


In some cases it is interesting to delay the execution of the fix point algorithm. The `with atomic` block can be used for that, as illustrated next:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x(cp,1..3);
var<CP>{int} y(cp,1..3);
var<CP>{int} z(cp,1..3);
solve<cp> {
  with atomic(cp) {
    cout << x << y << z << endl;
    cp.post(x<y);
    cout << x << y << z << endl;
    cp.post(y<z);
    cout << x << y << z << endl;
  }
  cout << x << y << z << endl;
}
```

The corresponding output is:

```
(3) [1..3] (3) [1..3] (3) [1..3]
(3) [1..3] (3) [1..3] (3) [1..3]
(3) [1..3] (3) [1..3] (3) [1..3]
123
```

10.3 Consistency Level

Typically, when designing a filtering (propagation) algorithm for a constraint, one is faced with the trade-off of filtering power versus speed: for instance, it may be possible to achieve more filtering at the expense of a slower algorithm. This means that several filtering algorithms may be available for the same constraint, each with varying degrees of filtering power. The consistency level basically determines which filtering algorithm will be used for a posted constraint.

The optional argument `cl` of the `post` command represents the consistency level for the constraints implied in the post. In other words, it specifies which filtering (propagation) algorithm will be used for these constraints.

It can take any value from the enumerated type `Consistency<CP>`, that is:

[Auto, onDomains, onBounds, onValues]

We now give a summary of the different consistency levels in COMET:

onDomains is the strongest possible filtering. It guarantees that every value not participating in a solution is removed.

onValues is the weakest consistency level. It only does some filtering, when a variable becomes bound (assigned) to a value.

onBounds stands somewhere in between **onValues** and **onDomains**. It guarantees that the upper and lower bound of every variable `x` involved in the constraint (i.e., `x.getMin()` and `x.getMax()`) belong to at least one solution satisfying the constraint after the propagation.

Auto is the default filtering of a constraint. It may correspond to one of the three consistency levels above, or may correspond to a filtering algorithm that does not belong to one of these three categories.

If no consistency level is specified in a `post` command, the **Auto** consistency level is used. Note that not all consistency levels are available for every type of constraint. If the requested consistency level is not available for a particular constraint, then the default consistency level, **Auto**, is used instead.

In Section 10.4, when describing the different constraints, we also mention the available consistency levels, as well as the default consistency level.

10.4 Constraints on Integer Variables

This section gives a list of most useful constraints on integer variables (`var<CP>{int}`):

- arithmetic and logical constraints [10.4.1](#)
- element constraints [10.4.2](#)
- table constraint [10.4.3](#)
- alldifferent and minAssignment [10.4.4](#)
- atleast, atmost, cardinality [10.4.5](#)
- binary- and multi-knapsack [10.4.6](#)
- circuit and minCircuit [10.4.7](#)
- sequence [10.4.8](#)
- stretch and regular [10.4.9](#)
- soft global constraints [10.4.10](#)

10.4.1 Arithmetic and Logical Constraints

We first describe the conceptually easiest form of CP constraints, involving expressions formed with integer CP variables combined with arithmetic and logical operators.

Binary and Expression Constraints It is possible to post constraints using the binary operators $+$, $-$, $*$, $/$ (integer division), $\%$ (modulo) and $==$ (equality):

```
import cotfd;
Solver<CP> cp();
var<CP>{int} w(cp,2..3);
var<CP>{int} x(cp,1..3);
var<CP>{int} y(cp,1..4);
var<CP>{int} z(cp,{1,5,2});
solve<cp> {
    cp.post((w*x + x/y) % y == (z - 3));
}
```

A solution satisfying this constraint is $w=2$, $x=1$, $y=3$, $z=5$. Only the **Auto** consistency level is available for these constraints.

Sum and Product Constraints of the form $\sum_{i \in [1..r]} x_i = z$ and $\prod_{i \in [1..r]} x_i = z$ can be posted using the **sum** and **prod** aggregate operators:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,1..3);
var<CP>{int} z(cp,{1,5,2});
solve<cp> {
    cp.post(sum(i in 1..3) x[i] == z);
}
```

A possible solution to this constraint is $x=[1,1,3]$ and $z=5$. Again, only the **Auto** consistency level is available for these constraints.

Logical and Reified Constraints Logical constraints using `&&` (and), `||` (or), `==` (equality), `=>` (implication) can be posted to build logical expressions:

```
import cotfd;
Solver<CP> cp();
var<CP>{bool} b1(cp);
var<CP>{bool} b2(cp);
var<CP>{bool} b3(cp);
solve<cp> {
    cp.post(b1 && b2 || b3);
}
```

A solution to this constraint is `b1 = false`, `b2 = false` and `b3 = true`.

The logical `and` and `or` constructs can be used in the same way as `sum` and `prod`, to post logical constraints over a range:

```
import cotfd;
Solver<CP> cp();
var<CP>{bool} b[1..5](cp);
solve<cp> {
    cp.post((and(i in 1..5) b[i]) == false);
    cp.post((or(i in 1..5) b[i]) == true);
}
```

A solution to these constraints is `b = [false,false,false,false,true]`.

10.4.2 Element Constraints

Element (or indexing) constraints allow to index an array of integers or variables by indexes that are variables themselves. In the following example, array `a` is indexed by variable `x` to be linked with variable `y`.

```
import cotfd;
Solver<CP> cp();
int a[1..7] = [9,3,4,6,2,7,4];
var<CP>{int} x(cp,1..7);
var<CP>{int} y(cp,{1,4,6});
solve<cp> {
    cp.post(a[x] == y);
}
```

A solution is `x=3, y=4` given that `a[3]==4`. The available consistency levels are `onBounds`(default) and `onDomains`. In the same way, we can index a matrix with a row and column variable: `cp.post(c[x,y])`.

10.4.3 Table Constraint

The **table** constraint is an example of a constraint given in extension. It constrains three variables (x_1, x_2, x_3) to take values according to one of the enumerated triples contained in the **Table<CP>** object given as its parameter.

```
import cotfd;
Solver<CP> cp();

int possibleTriples [1..4,1..3] = [[2,4,2],
                                   [5,1,6],
                                   [2,1,7],
                                   [2,3,3]];

var<CP>{int} x[1..3](cp,1..5);

solve<cp> {
  Table<CP> t(all(i in 1..4) possibleTriples[i,1],
              all(i in 1..4) possibleTriples[i,2],
              all(i in 1..4) possibleTriples[i,3]);

  cp.post(table(x[1],x[2],x[3],t));
}
```

A solution to this constraint is $x[1]=2$, $x[2]=1$ and $x[3]=7$ since (2,1,7) is one of the triples in the **possibleTriples Table<CP>** object. The only available consistency is **onDomains** (default).

10.4.4 Alldifferent Constraints

We now describe simple and weighted versions of the **alldifferent** constraint on arrays of variables.

alldifferent The **alldifferent** function allows to state that each variable in an array of CP variables takes a different value. For example:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..5](cp,1..6);
solve<cp>
  cp.post(alldifferent(x));
```

A solution is $x = [4, 2, 5, 1, 6]$ since all the values are different. The available consistency levels are **onValues** (default) and **onDomains**.

minAssignment The **alldifferent** constraint, in which there is a cost associated with each (variable,value) pair, is called the **minAssignment** constraint. An example of using this constraint with integer costs is given next:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..4](cp,1..4);
int w[1..4,1..4] = [[2,4,1,3],
                    [6,1,8,2],
                    [9,1,4,7],
                    [5,8,2,3]];
var<CP>{int} cost(cp,0..100);
minimize<cp>
  cost
subject to {
  cp.post(minAssignment(x,w,cost));
}
```

The best solution found is $x=[1, 4, 2, 3]$ with a cost of 7. The available consistencies are **onValues** (default) and **onDomains**.

10.4.5 Cardinality Constraints

The constraints discussed in this section are bounding, in different ways, the number of occurrences of certain domain values in an assignment of an array of CP variables.

atleast The **atleast** constraint states a lower bound on the number of occurrences of each value in an array of variables. In the following example, the value 1 must appear 0 times, the values 2 at least 2 times, the value 3 at least 1 time, and so on:

```
import cotfd;
Solver<CP> cp();
int low[1..5] = [0,2,1,3,0];
var<CP>{int} x[1..6](cp,1..5);
solve<cp> {
    cp.post(atleast(low,x));
}
```

A solution satisfying this constraint is $x=[2,2,3,4,4,4]$. The available consistencies are **onValues** (default) and **onDomains**.

atmost In an analogous fashion, **atmost** constrains the maximum number of occurrences of each value. For example:

```
import cotfd;
Solver<CP> cp();
int up[1..5] = [0,2,1,2,1];
var<CP>{int} x[1..6](cp,1..5);
solve<cp> {
    cp.post(atmost(up,x));
}
cout << x << endl;
```

A solution to this constraint is $x=[2,2,3,4,4,5]$. The available consistencies are **onValues** (default) and **onDomains**.

cardinality A combination of the two previous constraints is the **cardinality** constraint that limits both the minimum and maximum number of occurrences for each value:

```
import cotfd;
Solver<CP> cp();
int low[1..5] = [1,2,1,0,1];
int up[1..5] = [3,2,3,2,1];
var<CP>{int} x[1..6](cp,1..5);
solve<cp> {
    cp.post(cardinality(low,x,up));
}
cout << x << endl;
```

A solution to the above CP model is $x=[1,1,2,2,3,5]$. The available consistencies for this constraint are **onValues** (default) and **onDomains**.

exactly Finally, a special case of the **cardinality** constraint, in which **low** is the same with **up**, i.e., we specify an exact value for the number of occurrences of each value, is the **exactly** constraint, that has the following syntax:

```
exactly(int[] occ,var<CP>{int}[] x)
```

The constraint holds if there are exactly $occ[i]$ variables in the array **x** that are assigned to value **i**. The consistency levels available are **onValues** (default) and **onDomains**.

10.4.6 Knapsack Constraints

Different forms of knapsack constraint are a very valuable modeling tools for some Constraint Programming applications, such as packing and resource allocation.

binaryKnapsack The `binaryKnapsack` constraint is a constraint on the scalar product between a vector of 0-1 variables and a vector of integer. It can be viewed as the decision on which weighted items are placed into a knapsack of variable capacity.

```
import cotfd;
Solver<CP> cp();

int w[1..4] = [1,5,3,5];
var<CP>{int} x[1..4](cp,0..1);
var<CP>{int} c(cp,6..7);

solve<cp>
  cp.post(binaryKnapsack(x,w,c));
```

A solution to the above model is $x=[1,0,0,1]$, $c=6$, since $1 \cdot 1 + 0 \cdot 5 + 0 \cdot 3 + 1 \cdot 5 = 6$. Only the `onDomains` (default) consistency is available. For a weaker but more efficient propagation, you should use the `lightBinaryKnapsack` constraint.

multiknapsack A natural generalization of **binaryKnapsack** is the **multiknapsack** constraint, under which a set of weighted items must be placed into several bins:

```
import cotfd;
Solver<CP> cp();

int w[1..4] = [1,5,3,5];
var<CP>{int} x[1..4](cp,0..3);
var<CP>{int} l[1..3](cp,3..7);

solve<cp>
  cp.post(multiknapsack(x,w,l));
```

A solution satisfying the constraint is $x=[1,1,2,3]$, $l=[6,3,5]$ because

- objects 1 and 2, with weights 1 and 5, respectively, are placed into bin 1 ($x[1]=x[2]=1$) raising its total load to 6 ($l[1]=6$)
- object 3, with weight 3, is placed into bin 2 ($x[3]=2$), so that the load of bin 2 is 3 ($l[2]=3$)
- object 4 is placed into bin 3 ($x[4]=3$), so that the load of bin 3 is 5 ($l[3]=5$)

The only consistency level available for the **multinapsack** constraint is **Auto**. When there are additional precedence constraints between items (stating that some items must be placed into an earlier bin than others), we suggest to use the **multiknapsackWithPrecedences** constraint.

10.4.7 Circuit Constraints

This section presents the weighted and unweighted versions of the circuit (or no-cycle) constraint, a particularly interesting combinatorial constraint.

circuit The `circuit` constraint states that an array of variables has to represent a Hamiltonian circuit in a directed graph. Assume a graph $G = (V, E)$, where

- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (1, 3), (2, 4), (3, 1), (3, 2), (4, 2), (4, 1), (4, 3)\}$

In the following program, the graph is represented by the successor array of variables `x`, and the `circuit` constraint forces `x` to represent an Hamiltonian circuit on G .

```
import cotfd;

Solver<CP> cp();
range Nodes = 1..4;
set<int> succ[Nodes] = [{2,3},{4},{1,2},{2,1,3}];
var<CP>{int} x[s in Nodes](cp,succ[s]);
solve<cp>
  cp.post(circuit(x));
```

A solution to the above model is `x=[2,4,1,3]`, representing the circuit $(1 \rightarrow 2 \rightarrow 4 \rightarrow 3)$, where each node is different and all the edges exist in G .

minCircuit The same constraint with costs assigned to the edges is **minCircuit**. An application of it is for modeling the traveling salesman problem:

```
import cotfd;
Solver<CP> cp();
range Nodes = 1..4;
set{int} succ[Nodes] = [{2,3},{4},{1,2},{2,1,3}];
var<CP>{int} x [s in Nodes] (cp,succ[s]);
var<CP>{int} c(cp,0..100);
int costs[Nodes,Nodes] = [[1,3,2,4],
                           [5,2,7,3],
                           [6,2,4,1],
                           [2,7,4,1]];

minimize<cp>
  c
subject to
  cp.post(minCircuit<CP>(x, costs, c));
using
  labelFF(x);
```

10.4.8 Sequence Constraint

The sequence constraint has the following format:

```
sequence(var<CP>{int}[] x,int[] demand,int p,int q,set{int} V)
```

It essentially enforces two conditions:

- for every value *i* in the range of the integer array **demand**, exactly **demand[i]** variables from the array **x** are assigned to value *i*
- at most **p** out of any **q** consecutive variables in the array **x** are assigned values from the given integer set **V**

A very interesting application of the **sequence** constraint will be shown in Section 12.5, where it is used to model a car sequencing problem. The default consistency level is **onDomains** but **onValues** is also available.

10.4.9 Stretch and Regular Constraints

We now give an overview of constraints that control the pattern of occurrences of values in CP variable arrays. These range from simply limiting the number of consecutive appearances of the same value, to more advanced patterns involving finite automata.

stretch The **stretch** constraint controls the number of consecutive occurrences of values within an array of CP variables. Its simplest version has the following syntax:

```
stretch(int[] shortest,var<CP>{int}[] x,int[] longest)
```

This limits the number of consecutive occurrences of each value *i* between **shortest**[*i*] and **longest**[*i*].

An example of using this version of the constraint is the following:

```
import cotfd;
Solver<CP> cp();
set<int> dom[1..8] = [{1,3},{1,3},{1,2,3},{2,3},{2,3},{1,2,3},{1,3},{1,3}];
int shortest[1..3] = [2,3,4];
int longest[1..3] = [4,3,5];
var<CP>{int} x[i in dom.getRange()](cp,dm[i]);
solve<cp> {
    cp.post(stretch(shortest,x,longest));
}
```

A possible solution is $x=[1,1,1,2,2,2,1,1]$. Note how both runs of consecutive 1s are of length between 2 and 4. Also, note that the upper and lower bounds only apply to values that appear in the sequence. The only consistency level for the **stretch** constraint is **onDomains**.

stretch with transitions A more complex version of the `stretch` constraint can also specify the allowed transitions between values. The syntax in this case is:

```
stretch(int[] shortest, var<CP>{int}[] x, int[] longest, int[] from, int[] to)
```

For example, we can state that we only want the following transitions in the sequence :

- $1 \rightarrow 2$
- $1 \rightarrow 3$
- $3 \rightarrow 1$

Then the solution $x=[1,1,1,2,2,2,1,1]$ is not valid any more, since the transition $2 \rightarrow 1$ is not allowed.

The modified model, including the specification of allowed transitions, is the following:

```
import cotfd;
Solver<CP> cp();
set<int> dom[1..8] = [{1,3},{1,3},{1,2,3},{2,3},{2,3},{1,2,3},{1,3},{1,3}];
int shortest[1..3] = [2,3,4];
int longest[1..3] = [4,3,5];
var<CP>{int} x[i in dom.getRange()](cp, dom[i]);
int from[1..3] = [1,1,3];
int to[1..3] = [2,3,1];
solve<cp> {
  cp.post(stretch(shortest,x,longest,from,to));
}
```

A solution satisfying the constraint is, for instance, $x=[1,1,1,3,3,3,3,3]$.

regular An even more sophisticated way to specify rules on a sequence of variables is to use a finite deterministic automaton (see Figure 10.1) and a **regular** constraint. A regular constraint forces a sequence of variables to represent a valid execution path of an automaton.

Consider the graph representing the automaton. A valid execution path starts from the initial state (corresponding to the diamond-shaped state 1 in Figure 10.1) and traverses a sequence of directed arcs in the graph, until it reaches an accepting (terminal) state (corresponding to the square-shaped states 6 and 7). At every arc it traverses, it emits the value that labels the arc. For example, a possible sequence of values generated by a valid execution path of the automaton shown in Figure 10.1 is (3, 2, 2, 1, 4). This corresponds to visiting the states: $1 \rightarrow 2 \rightarrow 2 \rightarrow 1 \rightarrow 4$.

The following code fragment first encodes the automaton, and then posts a **regular** constraint on a sequence of 6 variables:

```
import cotfd;
Solver<CP> cp();
Automaton<CP> automaton(1..7,1..4,1,{6,7});
automaton.addTransition(1,2,3);
automaton.addTransition(1,3,4);
automaton.addTransition(2,2,2);
automaton.addTransition(2,5,1);
automaton.addTransition(3,4,1);
automaton.addTransition(4,5,3);
automaton.addTransition(5,5,3);
automaton.addTransition(5,6,2);
automaton.addTransition(5,7,4);
var<CP>{int} x[i in 1..6](cp,1..4);
solve<cp> {
    cp.post(regular(x,automaton));
}
```

A solution is $x=[3,1,3,3,3,2]$. The only consistency level for **regular** is **onDomains**.

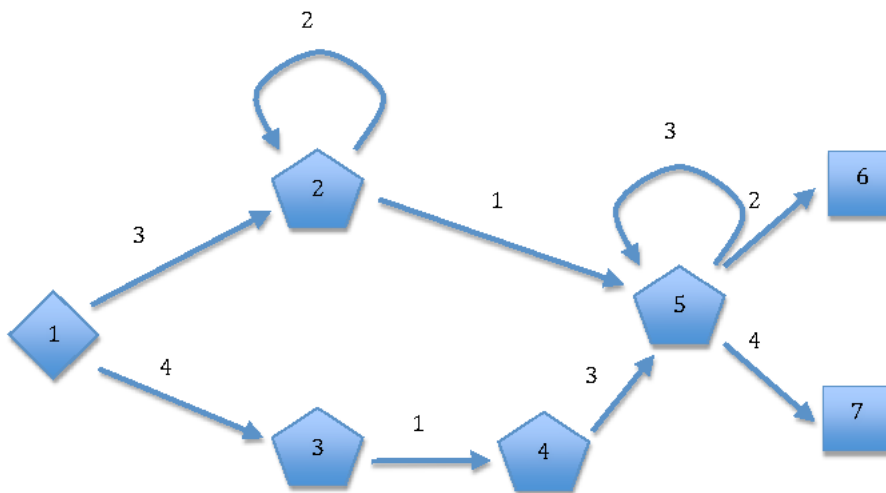


Figure 10.1: Depth first search exploration: Numbers inside leaf nodes is the order in which the solution is encountered

10.4.10 Soft Global Constraints

COMET supports soft versions for a number of global constraints. Generally speaking, a soft constraint is a constraint that allows for violations, with respect to its corresponding hard constraint. Typically, a soft constraint adds an extra CP variable parameter to its hard constraint counterpart, to represent the number of violations (or alternatively the proximity to satisfying the hard constraint). Soft constraints can be very useful for over-constrained problems or for problems in which constraints can rather be viewed as preferences. The following table shows a list of soft global constraints available in COMET, alongside with their hard constraint counterpart.

| hard version | soft version | consistency |
|----------------------------------|---------------------------|-----------------------------------|
| alldifferent | atLeastNValue | onValue , onDomains |
| atleast | softAtLeast | onDomains |
| atmost | softAtMost | onDomains |
| cardinality | softCardinality | onDomains |
| $x[1] = x[2] = \dots = x[n] = m$ | spread / deviation | onBounds |

We now give a more detailed presentation of the soft constraints supported in COMET.

atLeastNValue The **atLeastNValue** global constraint is the soft version of the **alldifferent**. It counts the number of different values in a vector of variables. The signature of this constraint is:

AtLeastNValue<CP> **atLeastNValue**(**var**<CP>{**int**}[] **x**, **var**<CP>{**int**} **numberOfValues**)

The constraint maintains that the variable **numberOfValues** is the number of different values taken by the variables in the array **x**. A simple example is given in the following code fragment. We will see a more interesting example of **atLeastNValue** in Section 14.2.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..5](cp,1..6);
var<CP>{int} n(cp,3..5);
solve<cp>
    cp.post(atLeastNValue(x,n));
```

A solution to the above constraint is **x**=[1,1,1,2,3], **n**=3, since there are three different values in the array **x**. The available consistencies are **onValues** (default) and **onDomains**.

softAtLeast The relaxation of an **atleast** constraint is a **softAtLeast** constraint:

```
SoftAtLeast<CP> softAtLeast(int[] low, a var<CP>{int}[] x, var<CP>{int} viol)
```

In the above signature, the finite domain variable **viol** represents the total number of violations. This is equal to the sum, over all values, of the shortage with respect to what is specified in the array **low**. Let R be the range of **low** and let $\text{card}(v) = |\{i \mid x[i] = v\}|$. Then the formula for the violations is:

$$\text{viol} = \sum_{v \in R} \max(0, \text{low}[v] - \text{card}(v))$$

We are going to see an interesting application of the **softAtLeast** constraint in Section 14.3.

softAtMost The **softAtMost** constraint is the soft version of **atmost** and is simply the symmetrical case of **softAtLeast**:

```
SoftAtMost<CP> softAtMost(int[] up, var<CP>{int}[] x, var<CP>{int} viol)
```

where now the violation represents the sum, over all values, of the excess with respect to what is specified in the array **up** (R is the range of **up**, and $\text{card}(v)$ is defined as before):

$$\text{viol} = \sum_{v \in R} \max(0, \text{card}(v) - \text{up}[v])$$

softCardinality The soft version of the `cardinality` constraint is the `softCardinality` constraint, which combines the `softAtLeast` and the `softAtMost` constraints:

```
softCardinality(int[] low, var<CP>{int}[] x, int[] up, var<CP>{int} viol)
```

The violation represents the sum, over all the values, of the shortage or excess with respect to what is specified in arrays `low` and `up` (Both arrays need to be defined over the same range R):

$$\text{viol} = \sum_{v \in R} \max(0, \text{card}(v) - \text{up}[v], \text{low}[v] - \text{card}(v))$$

An example using `softCardinality` is given next:

```
import cotfd;
Solver<CP> cp();
int low[1..5] = [1,2,1,0,1];
int up[1..5] = [3,2,3,2,1];
var<CP>{int} x[1..6](cp,1..5);
var<CP>{int} viol(cp,0..5);
solve<cp>
  cp.post(softCardinality(low,x,up,viol));
```

A solution to this constraint is `x=[1,1,1,1,2,2]`, `viol=3`. This is because:

- the value 1 has one violation: it appears 4 times and it should appear maximum 3 times
- the value 3 has one violation: it should appear at least once and doesn't appear
- the value 5 has one violation for the same reason as value 3

We conclude this section with two very interesting balancing constraints, that control the distribution of values in an array of variables. When we want a perfect balance of a set of n variables, so that all of them are assigned to the same value m , we can try posting the constraint $x[1] = x[2] \dots = x[n] = m$. If this results into an over-constrained problem, though, we need to relax this constraint, maintaining some kind of balance. This can be accomplished using one of constraints `spread` and `deviation`. These constraints can be particularly useful in load-balancing problems, for instance.

spread The `spread` constraint is useful for obtaining a balanced vector of variables, around a given mean. It constrains the *mean quadratic deviation* with respect to the given mean value \mathbf{s}/n . In more detail, posting the constraint

`spread(var<CP>{int}[] x, int s, var<CP>{int} d)`

enforces that: $\sum_i x[i] = \mathbf{s}$ and $n \cdot \sum_i (x[i] - \mathbf{s}/n)^2 = \mathbf{d}$, where $n = |\mathbf{x}|$. Consider, for example, the following block of COMET code:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..4](cp,1..4);
var<CP>{int} d(cp,0..100);
minimize<cp> d
subject to
    cp.post(spread(x,7,d));
```

An optimal solution is $\mathbf{x}=[1,2,2,2]$ with $\mathbf{d}=3$. We can check that the sum is correctly equal to 7 and that $4 \cdot \sum_i (x[i] - 7/4)^2 = 3$.

deviation The **deviation** constraint works exactly like **spread**, but it instead constrains the *mean absolute deviation* from the mean s/n . More formally, posting the constraint:

$$\text{deviation}(\text{var}\langle\text{CP}\rangle\{\text{int}\}[] \ x, \text{int } s, \text{var}\langle\text{CP}\rangle\{\text{int}\} \ d)$$

enforces that $\sum_i x[i] = s$ and $n \cdot \sum_i |x[i] - s/n| = d$, where $n = |x|$. The following example shows how to use **deviation**:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..4](cp,1..4);
var<CP>{int} d(cp,0..100);
minimize<cp> d
subject to
    cp.post(deviation(x,7,d));
```

An optimal solution is $x=[1,2,2,2]$ with $d=6$. We can check that the sum is correctly equal to 7 and that $4 \cdot \sum_i |x[i] - 7/4| = 6$.

Note that, strictly speaking, none of the soft constraints presented in this section are strictly necessary from a modeling point of view. They can all be expressed/decomposed as combinations of simple arithmetic constraints. Nevertheless, it is highly recommended to use them rather than their equivalent decomposition, in order to achieve much stronger filtering. This also holds for most of the hard global constraints.

Chapter 1

Set Variables

In this chapter, we describe how set variables are supported in COMET's Constraint Programming module. Set variables (`var<CP>{set{int}}`) represent variables, whose values are set of integers, as opposed to integer variables (`var<CP>{int}`), that take integer values. After discussing how set variables are represented and accessed in COMET, we give a review of the constraints available for set variables, and conclude with an application of set variables into solving the SONET problem.

11.1 Representation

Obviously, a set variable could be represented by a set of integer variables; however, there are some reasons, why a set variable is preferred in some cases:

- easier representation
- more natural modelling for certain problems, where solutions are sets
- inherently breaks some variable symmetries
- may achieve better pruning
- potentially lower memory consumption

In practice, though, the best results are obtained in hybrid models, in which integer and set variables coexist and are linked through channeling constraints.

Currently, in `COMET` set variables are represented by a pair of integer sets (R, P) and by an integer variable c ; R represents the set of required elements of the set variable, P is the set of possible elements and c is the cardinality of the set variable. Let S be a set variable, then its domain (set of sets) is defined by:

$$D(S) = \{s \mid R \subseteq s \subseteq P, |s| = c\}$$

Note that, if U denotes the universe of elements and E the set of elements excluded by the set variable, then $P = U \setminus E$.

11.2 Interface

A set variable can be defined in COMET by specifying the possible elements, as a range or as a set of integers, and, optionally, its cardinality. The cardinality can be given as an integer value or as a range of integers. For example, consider the following code fragment:

```
import cotfd;
Solver cp();
var<CP>{set{int}} S(cp,1..5,2..4);
var<CP>{set{int}} S1(cp,{3,5,7,8,9},2..4);
```

It creates two set variables: **S**, whose possible elements range from 1 to 5, and **S1**, whose possible elements are {3,5,7,8,9}; in both cases, their cardinality must be between 2 and 4. Note that we cannot specify the required set *R* in the constructor of a set variable. By default, every time a set variable is defined, its corresponding required set *R* is initialized to the empty set. Required elements have to be added to *R* when stating the CP model.

The most common methods available in the API for set variables, aside from the constructors, are summarized in the following block:

| | |
|---------------------------|--|
| boolean | <code>bound();</code> |
| set{int} | <code>getValue();</code> |
| set{int} | <code>evalIntSet();</code> |
| var<CP>{int} | <code>getCardinalityVariable();</code> |
| set{int} | <code>getRequiredSet();</code> |
| set{int} | <code>getPossibleSet();</code> |
| boolean | <code>isRequired(int v);</code> |
| boolean | <code>isExcluded(int v);</code> |

In the same way with other types of CP variables, the method **bound** returns **true**, when the set variable has been assigned a value (which is a set of integers in this case). That value can be retrieved using either of the methods **getValue** and **evalIntSet**, when the set variable is bound. If the set variable is not bound, then **getValue** will return **null**, whereas **evalIntSet** will raise an exception.

The second group of methods is more interesting. They allow access to the set variable's cardinality and give information about the required set and the set of possible values.

We illustrate these functionalities by extending the example shown in the beginning of the section.

```
import cotfd;
Solver<CP> cp();
var<CP>{set{int}} S(cp,1..5,2..4);
cp.post(S.getCardinalityVariable()!=3);
cp.post(requiresValue(S,3));
cp.post(excludesValue(S,2));
cout << S.getRequiredSet() << endl;
cout << S.getPossibleSet() << endl;
cout << S.isRequired(4) << endl;
cout << S.isExcluded(2) << endl;
```

In the above code block, we can see how the `getCardinalityVariable` method allows to retrieve and constrain the cardinality variable c of a set variable. In this case, it imposes that the cardinality of the set variable S cannot be equal to 3. The next two lines constrain the required and possible elements of the set variables using the constraints `requiresValue` and `excludesValue`. A detailed account of the constraints available for set variables is going to be given in the following section. Normally, the previous post instructions should be included in a `solve` or `solveall` or `minimize` block. If we consider all the constraints posted on variable S , the possible values that it may take are now limited to:

$$D(S) = \{\{1, 3\}, \{3, 4\}, \{3, 5\}, \{1, 3, 4, 5\}\}$$

As we can notice, this includes no sets with cardinality 3; no set includes the element 2, since it was excluded; and all of the possible sets include the element 3, since it was required.

Finally, the code retrieves the set of required and possible elements using the methods `getRequiredSet` and `getPossibleSet`. The methods `isRequired` and `isExcluded` can be used for testing whether individual values are contained in the corresponding sets.

The output of the above code fragment is:

```
{3}
{1,3,4,5}
false
true
```

If we tried to print directly the set variable S , we would get the following output:

```
((1){3},(1){2},(4){1,3,4,5} | (DOM:2)[2,4])
```

The first set (i.e., $\{3\}$) represents the set of required elements, the second (i.e., $\{2\}$) the set of excluded elements, the third (i.e., $\{1,3,4,5\}$) the possible elements; finally, the domain of the cardinality variable is printed.

In addition to the above described methods, COMET gives direct access to boolean variables that represent the inclusion or exclusion of specific values in a set variable. This can be very useful for reification purposes, or for more fine-grained control of what to include or exclude from a set variable, when stating a model.

This is the relevant part of the API:

```
var<CP>{boolean} getRequired(int v);  
var<CP>{boolean} getExcluded(int v);  
boolean          hasRequiredVariable(int v);  
boolean          hasExcludedVariable(int v);
```

The method `getRequired` creates a boolean variable once and returns it. The returned variable represents whether a value is required in the set or not, and may be used to post constraints. We can check whether the boolean variable representing the requirement of value `v` in the set has been created, by calling the method `hasRequiredVariable`. Methods `getExcluded` and `hasExcludedVariable` work in an analogous fashion.

Caveat Note that class `var<CP>{set{int}}` also provides the methods `requires(int v)` and `excludes(int v)`. These methods are intended to be used exclusively in constraint propagators and should not be used for posting constraints.

11.3 Constraints over Set Variables

In this section, we present the constraints provided by COMET for set variables. We identify the following families of set variable constraints:

- Basic operations (equality, union, intersection, difference, subset) [11.3.1](#)
- Constraints on set cardinality [11.3.2](#)
- Requirement and exclusion constraints [11.3.3](#)
- Channeling constraints [11.3.4](#)
- Global constraints (Set Global Cardinality) [11.3.5](#)

11.3.1 Basic Set Operation Constraints

We first show how to express constraints involving basic set operations, such as equality, union, intersection, set difference and subset. These are summarized in the following block:

```
cp.post(S1==S2);
cp.post(subset(S1,S2));
cp.post(setunion(S1,S2,RES));
cp.post(setinter(S1,S2,RES));
cp.post(setdifference(S1,S2,RES));
```

Testing equality is self-explanatory. The subset constraint enforces $S1 \subseteq S2$. In the rest of the constraints, RES corresponds to the output of the corresponding set operation between set variables S1 and S2. In particular, `setunion` states that $RES = S1 \cup S2$, `setinter` states that $RES = S1 \cap S2$ and `setdifference` that $RES = S1 \setminus S2$.

The last three constraints can be alternatively stated as:

```
RES = setunion(S1,S2);
RES = setinter(S1,S2);
RES = setdifference(S1,S2);
```

11.3.2 Cardinality

We can express constraints on the cardinality of a set by:

```
cp.post(cardinality(S1,k));
```

This enforces $|S1| = k$. Alternatively, we can retrieve the integer variable representing the cardinality with the method `getCardinalityVariable` and post constraints directly on that.

Cardinality of intersection It is possible to constrain the cardinality of the intersection of two sets with one of the following constraints:

```
cp.post(atleastIntersection(S1,S2,k));  
cp.post(atmostIntersection(S1,S2,k));  
cp.post(exactIntersection(S1,S2,k));
```

These enforce that the intersection between the two sets has to be, respectively, at least, at most or exactly equal to the integer passed as last parameter. Note that these constraints achieve stronger filtering than intersecting the two sets and posting a constraint on the cardinality of the result.

In case the intersection needs to be empty, i.e., the set variables need to be disjoint, we can use:

```
cp.post(disjoint(S1,S2));  
cp.post(allDisjoint(SA));
```

where `S1` and `S2` are set variables and `SA` is an array of set variables.

11.3.3 Requires and excludes

To enforce that a value must be present or must be excluded from a set variable, we can respectively use:

```
cp.post(requiresValue(S,k1));
cp.post(excludesValue(S,k2));
```

Analogously, we may want to require or to exclude from a set variable the value taken by an integer variable, as follows:

```
cp.post(requiresVariable(S,x1));
cp.post(excludesVariable(S,x2));
```

As mentioned earlier, in Section 11.2, one can also retrieve boolean variables that represent the inclusion or exclusion of a specific value from a set variable, as for example in:

```
var<CP>{boolean} bReq = S.getRequired(k1);
var<CP>{boolean} bExc = S.getExcluded(k2);
```

Clearly, we can bound and constrain these boolean variables to enforce inclusion or exclusion.

11.3.4 Channeling

We start this subsection with describing the channeling constraint between two arrays of set variables:

```
cp.post(channeling(SA1,SA2));
```

which enforces:

$$SA_1[i] = s \iff \forall j \in s : i \in SA_2[j]$$

Equivalently, we can express the condition enforced by channeling in the following way, which makes more clear the symmetric nature of this constraint:

$$\begin{aligned} SA_1[i] &= \{j \mid SA_2[j] \text{ contains } i\} \\ SA_2[j] &= \{i \mid SA_1[i] \text{ contains } j\} \end{aligned}$$

An example that satisfies the constraint is (assuming that the ranges of the arrays are 1..3):

```
SA1 = [{1,2},{3},{1,2}]
SA2 = [{1,3},{1,3},{2}]
```

Sometimes, it is necessary to model problems with both set variables and integer variables, in order to exploit the best of both representations. COMET provides some channeling constraint to automatically link integer variables with set variables:

```
cp.post(channeling(SA,X));
```

Here, **SA** is an array of `var<CP>{set{int}}`, whereas **X1** is an array of `var<CP>{int}`.

This constraint enforces:

$$SA[i] = s \iff \forall j \in s : X[j] = i$$

Note that the above constraint implies that the set variables in **SA** are all disjoint.

An example that satisfies the constraint is (assuming that all ranges starts from 1):

```
SA = [{1,2},{3}]
X = [1,1,2]
```

We can make a set variable equal to the union of an array of integer variables by using the following constraint:

```
cp.post(unionOf(X,S));
```

which enforces:

$$S = \bigcup_i X_i$$

Finally, if a set variable is a singleton, i.e., has cardinality 1, and it is equal to a specific integer variable, we employ:

```
cp.post(singleton(x,S));
```

where x is an integer variable, and it constrains S such that: $S = \{x\}$

11.3.5 Set Global Cardinality

This constraint is the natural extension of the global cardinality constraint, applied to set variables. Here, by cardinality we refer to the number of occurrences of an element in an array of set variables (rather than the cardinality of a set as in Section 11.3.2). The constraint bounds the minimum and maximum number of occurrences of the elements that appear in an array of sets. Basically, if an element e needs to appear between l_v and u_v times, then it will be contained by at least l_v set variables and at most u_v set variables.

More formally, let \mathcal{S} be a set of set variables, U the universe (i.e., the union of all the possible elements of all the set variables) and l_v and u_v respectively the lower and upper bound for each $v \in U$. Then, the constraint enforces:

$$\forall v \in U : l_v \leq |\mathcal{S}_v| \leq u_v$$

where \mathcal{S}_v is the set of set variables that contain the element v , i.e., $\mathcal{S}_v = \{s \in \mathcal{S} : v \in s\}$

The syntax of this constraint is:

```
cp.post(setGlobalCardinality(l,SA,u));
```

where l and u are the arrays of lower and upper bounds and SA is the array of set variables.

Consider the following lower and upper bounds (ranges start from 1):

$$l = [1, 2, 1, 2, 0]$$

$$u = [1, 3, 2, 4, 1]$$

meaning that the element 1 must appear exactly once, value 2 must appear between two and three times, value 3 between once and twice, value 4 at least two and at most four times, and value 5 at most once. Then a feasible solution is the following:

$$SA = [\{1, 2\}, \{2, 4\}, \{3\}, \{2, 3, 4\}]$$

Note that each element inside the set variables satisfies the bounds defined above: the element 1 appears in one set variable, the element 2 in three variables, the elements 3 and 4 appear in two variables each, and the element 5 does not appear in any variable.

11.4 The SONET Problem

This chapter concludes with presenting the SONET problem and describing a model that solves it using set variables. The SONET problem is a network design optimization problem: we are given a set of nodes along with communication demands between pairs of nodes. Two nodes can communicate, only if they join the same ring; nodes may join more than one ring. The total number of rings is given and each ring has a maximum capacity, both in terms of the number of client nodes it can host, and in terms of the total demand it can manage.

The minimization objective derives from the cost incurred in making a node join a ring. Therefore, we need to minimize the sum, over all rings, of the number of nodes that join each ring. In this section, we solve a simpler version of the problem, in which rings can manage unlimited demand, and we are only concerned with the node capacity of each ring, i.e., the maximum number of nodes it can host. The complete code for this version of the problem is given in Statement 11.1. The specific instance we deal with, has four rings with node capacities 3,2,2 and 3, respectively. There are five nodes and five pairs of nodes that need to communicate.

The Model To model the set of nodes that join a ring, we can employ a set variable for each ring; the possible elements are the set of nodes and the cardinality is the capacity of the ring:

```
range Rings = 1..4;
int capacity[Rings] = [3,2,2,3];
range Nodes = 1..5;
Solver<CP> cp();
var<CP>{set{int}} nodesInRing[r in Rings](cp,Nodes,0..capacity[r]);
```

Note that bounding the cardinality of these set variables automatically enforces the node capacity constraint. However, it is not easy to express, on these variables, the constraint that two communicating nodes must share at least a ring. To do so, we introduce an array of dual set variables representing the set of rings that a node joins.

```
var<CP>{set{int}} ringsOfNode[Nodes](cp,Rings);
```

```

import cotfd;
Solver<CP> cp();
range Rings = 1..4; // upper bound for amount of rings
range Nodes = 1..5; // amount of clients
int demand[Nodes,Nodes] = [[0,1,0,1,1],
                             [1,0,1,0,0],
                             [0,1,0,0,1],
                             [1,0,0,0,0],
                             [1,0,1,0,0]];
int capacity[Rings] = [3,2,2,3]; // capacity of possible rings

var<CP>{set{int}} nodesInRing[r in Rings](cp,Nodes,0..capacity[r]);
var<CP>{set{int}} ringsOfNode[Nodes](cp,Rings);
minimize<cp>
    sum(r in Rings) nodesInRing[r].getCardinalityVariable()
subject to {
    // at least two nodes in each ring
    forall (r in Rings)
        cp.post(nodesInRing[r].getCardinalityVariable() != 1);

    forall (n1 in Nodes, n2 in Nodes: n1 < n2 && demand[n1,n2] == 1)
        cp.post(atleastIntersection(ringsOfNode[n1], ringsOfNode[n2], 1));
    cp.post(channeling(nodesInRing, ringsOfNode));
}
using {
    while (!and(i in Rings)(nodesInRing[i].bound())) {
        selectMin (i in Rings: !nodesInRing[i].bound())
            (nodesInRing[i].getCardinalityVariable().getSize()) {
            set{int} S = nodesInRing[i].getPossibleSet();
            set{int} R = nodesInRing[i].getRequiredSet();
            Solver<CP> cp = nodesInRing[i].getSolver();
            selectMax (e in S: !R.contains(e)) (sum(n in R) (demand[e,n])) {
                try<cp> cp.requires(nodesInRing[i], e); | cp.excludes(nodesInRing[i], e);
            }
        }
    }
    forall (r in Rings) cout << nodesInRing[r] << endl;
}

```

Statement 11.1: Model for the SONET Problem (sonet-cp.co)

We link the two arrays of variables using the channeling constraint described in [11.3.4](#):

```
cp.post(channeling(nodesInRing,ringsOfNode));
```

We can find the rings shared between any two nodes by intersecting their respective `ringsOfNode` variables. Then, it suffices to force the intersection to have a cardinality of at least one, if the two needs to communicate, as shown in the following code fragment:

```
forall (n1 in Nodes,n2 in Nodes: n1<n2 && demand[n1,n2]==1)
  cp.post(atleastIntersection(ringsOfNode[n1],ringsOfNode[n2],1));
```

Note that `demand[n1,n2]` is equal to 1, iff `n1` communicates with `n2`.

At this point, we have modeled all the constraints; however, we can achieve better filtering by adding some redundant constraints. In particular, there is no point in having rings that host just a single node: this would increase the objective function, but it would not help with satisfying the problem constraints, since there must be at least two nodes in a ring, to satisfy a communication demand.

```
forall (r in Rings)
  cp.post(nodesInRing[r].getCardinalityVariable()!=1);
```

Note that, since a solution may include empty rings (that are not used to host any node), we cannot rule out cardinality 0 for the `nodesInRing` variables.

The objective function to minimize can be expressed as follows:

```
minimize<cp>
  sum(r in Rings) nodesInRing[r].getCardinalityVariable()
```

which basically takes the sum, over all rings, of the number of nodes each ring hosts. Equivalently, we could have used:

```
minimize<cp>
  sum(n in Nodes) ringsOfNode[n].getCardinalityVariable()
```

We now focus on the heuristics that can be used for performing the search.

Simple Heuristic The simplest heuristic we could employ is the following, which is one of the search heuristics already provided by COMET:

```
label(nodesInRing);
```

This non-deterministically assigns, one by one, in lexicographical order, the set variables in the array `nodesInRing`. The value-ordering heuristic, i.e., the assignment of each individual set variable, proceeds by trying to include one of the possible elements in the set and, in case of a failure, it excludes it from the set.

The equivalent code for assigning the set variable in position `i`, `nodesInRing[i]`, resembles the following snippet of code:

```
set{int} S = nodesInRing[i].getPossibleSet();
set{int} R = nodesInRing[i].getRequiredSet();
forall (e in S: !R.contains(e))
  try<cp> cp.requires(nodesInRing[i],e); | cp.excludes(nodesInRing[i],e);
```

Note the use of the methods `requires` and `excludes` of class `Solver<CP>`. These methods may be employed to design a customized search heuristic.

COMET also offers a search heuristic called `labelFF`, inspired by the first-fail principle. This heuristic orders the variables by increasing set cardinality, breaking ties by choosing the variable with the smallest domain size (i.e., the variable that minimizes the difference between the size of the possible element set and the size of the required element set).

Custom Heuristic Of course, it is possible to develop a more efficient and fine-grained heuristic for this problem. In the same way with `labelFF`, we first order the variables by increasing set cardinality, but we now take into account the structure of the problem, in order to assign values to variables.

Once a variable `nodesInRing[i]` has been chosen, we first try to include the node that has the most communication with the nodes already placed in ring `i`, as in the following code:

```
set{int} S = nodesInRing[i].getPossibleSet();
set{int} R = nodesInRing[i].getRequiredSet();
forall (e in S) by (-sum(n in R)(demand[e,n]))
    try<cp> cp.requires(nodesInRing[i],e); | cp.excludes(nodesInRing[i],e);
```

This solution, though, has the drawback that, once we choose a set variable, we branch on all its possible elements, before choosing the next set variable. A more fine-grained solution would be to review the variable selection each time we assign a node to a ring.

The full code of the modified heuristic is shown below:

```
while (!and(i in Rings)(nodesInRing[i].bound())) {
    selectMin (i in Rings: !nodesInRing[i].bound())
        (nodesInRing[i].getCardinalityVariable().getSize()) {
        set{int} S = nodesInRing[i].getPossibleSet();
        set{int} R = nodesInRing[i].getRequiredSet();
        Solver<CP> cp = nodesInRing[i].getSolver();
        selectMax (e in S: !R.contains(e)) (sum(n in R)(demand[e,n])) {
            try<cp> cp.requires(nodesInRing[i],e); | cp.excludes(nodesInRing[i],e);
        }
    }
}
```

The outer **while** loop serves to ensure that, at the end of the heuristic procedure, all variables are bound (assigned). Each time, we select a ring corresponding to a variable with minimum cardinality and, then, try to include one single node into that ring; after that, a possibly new variable is selected.

Chapter 12

Constraint Programming Examples

This chapter shows a wide range of successful applications of Constraint Programming to optimization problems, offering a better understanding of modeling and search techniques in CP. Also, several of the constraints described in earlier chapters of the tutorial are now used in the context of actual problems.

12.1 Labeled Dice

In this section we will see:

- an example using the `alldifferent` and the `exactly` global constraints
- the use of `solveall` to enumerate all solutions
- a problem with intrinsic symmetries and the use of constraints to avoid enumerating symmetrical solutions

The Labeled Dice Problem is an interesting puzzle, created by Humphrey Dudley and proposed on Jim Orlin's blog.¹ Here is a description of the problem. We are given the following 12 words: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA. There are 24 different letters appearing in these 12 words. The question is: can we assign the 24 letters to 4 different cubes so that the four letters of each word appears on different cubes? (There is one letter from each word on each face of the cubes.) After three out of the five solutions posted on the blog used Constraint Programming, Jim Orlin made the following comment: *"The ease for solving it using Constraint Programming makes me think that Constraint Programming should be considered a fundamental tool in the OR toolkit."* The complete COMET model is given in Statement 12.1.

One variable is created for each letter of the alphabet in the `cube` array. It represents the cube where the corresponding letter is placed. Since not all letters of the alphabet may be present in a word, we collect, in an array `cube_`, the variables, whose corresponding letter appears in at least one word. Only the variables in the `cube_` array are decision variables.

```
var<CP>{int}    cube[ALPHABET] (cp,Cubes);
var<CP>{int}[]  cube_ = all(1 in letters) cube[1];
```

The first constraint is that the letters of each word are placed on different cubes. This is ensured with an `alldifferent` constraint over the letter variables of each word. Note that we ask COMET to use the domain consistent filtering algorithm with the `onDomains` parameter.

```
forall (w in Words)
  cp.post(alldifferent(all(i in Cubes)(cube[words[w,i]])),onDomains);
```

¹<http://jimorlin.wordpress.com/2009/02/17/colored-letters-labeled-dice-a-logic-puzzle/>

```

enum ALPHABET = {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z};
range Words = 1..12;
range Cubes = 1..4;
ALPHABET words[Words,Cubes] = [[B,U,O,Y],[C,A,V,E],[C,E,L,T],[F,L,U,B],[F,O,R,K],
                                [H,E,M,P],[J,U,D,Y],[J,U,N,K],[L,I,M,N],[Q,U,I,P],[S,W,A,G],[V,I,S,A]];
set{ALPHABET} letters = union(w in Words) (collect(i in Cubes) words[w,i]);

import cotfd;
Solver<CP> cp();
var<CP>{int} cube[ALPHABET](cp,Cubes);
var<CP>{int}[] cube_ = all(1 in letters) cube[1];
solveall<cp> {
    forall(w in Words)
        cp.post(alldifferent(all(i in Cubes)(cube[words[w,i]])),onDomains);
    cp.post(exactly(all(i in Cubes)(6), cube_));
    forall(i in Cubes)
        cp.label(cube[words[1,i]],i);
}
using {
    labelFF(cube_);
    cout << "-----" << endl;
    forall(i in Cubes)
        cout << filter(j in letters)(cube[j].bound() && cube[j]==i) << endl;
}
cout << "#fails:" << cp.getNFail() << endl;

```

Statement 12.1: CP Model for the Labeled Dice Problem (labeled-dice-cp.co)

Since each cube only has six faces, we must also enforce that each cube is used exactly six times in the decision variables. The first argument corresponds to the number of occurrences of values from the range `Cubes` in the array of variables `cube_`. We use array comprehension to build, on the fly, the array `[6,6,6,6]` with `range Cubes`, which is given as parameter to the `exactly` constraint:

```
cp.post(exactly(all(i in Cubes) (6), cube_));
```

Note that, for any given solution to this problem, we can build 23 other symmetrical solutions, by simply permuting the four cubes. To avoid exploring symmetrical solutions, we add constraints to arbitrarily assign the letters of the first word to the different cubes:

```
forall (i in Cubes)
    cp.label(cube[words[1,i]], i);
```

A standard first-fail search strategy is sufficient for this problem: `labelFF(cube_)`. We also add some extra lines into the `using` block, that print each solution found. (Two solutions in this case)

12.2 Bin-Packing

This section shows how to solve a bin-packing problem with constraint programming in COMET. The bin packing problem consists in placing items of given sizes into bins of fixed capacity, using as few bins as possible. For problems of this kind, it is often more convenient to solve a series of satisfaction problems, in each of which we fix the number of available bins. Let b_0 be a lower bound on the number of bins necessary to place all the items. We first try to find a solution that uses only b_0 bins. If the search fails, then we try to find a solution that uses at most $b_0 + 1$ bins. We then keep increasing the number of available bins, until we find a solution. By construction, this will be a solution that uses the smallest possible number of bins.

The most interesting COMET features of this example include:

- the use of `restartOnCompletion` to transparently increase the number of available bins
- the global constraints `multiknapsack` and `binpackingLB`
- the use of the `onFailure` block to implement a search that dynamically breaks symmetries

The complete COMET model is given in Statement [12.2](#).

The first lines introduce the problem's data: the weight of each item and the capacity of the bins. Then, the following lines declare and initialize the only two sets of variables for this problem: `x[i]`, the index of the bin where item `i` is placed, and `l[j]`, the load of bin `j` (i.e., the sum of weights of the items placed into bin `j`).

```
Solver<CP> cp();
var<CP>{int} x[Items](cp,Bins);
var<CP>{int} l[Bins](cp,0..capacity);
```

The number of available bins is represented by the `Integer` object `nbBins`, that is initialized with the simple lower bound $\lceil \text{totalWeight}/\text{capacity} \rceil$.

```
Integer nbBins((int) ceil(totalWeight/capacity));
```

```

import cotfd;
range Items = 1..50;
int weights [Items] = [95,92,87,87,85,84,83,79,77,77,75,73,69,68,65,63,63,62,
                      61,58,57,52,50,44,43,40,40,38,38,38,35,33,33,32,31,29,
                      27,24,24,22,19,19,18,16,14,11,6,4,3,2];
float totalWeight = sum(i in Items) weights[i];
range Bins = 1..50;
int capacity = 100;

Solver<CP> cp();
var<CP>{int} x[Items](cp,Bins);
var<CP>{int} l[Bins](cp,0..capacity);

Integer nbBins((int) ceil(totalWeight/capacity));
cp.restartOnCompletion();
solve<cp> {
    cp.post(multiknapsack(x,weights,l));
    cp.post(binpackingLB(x,weights,l));
}
using {
    forall (b in Bins: b > nbBins)
        cp.label(l[b],0);
    while(!bound(x)) {
        int currentLoad[b in Bins] =
            sum(i in Items : x[i].bound() && x[i].getValue() == b) weights[i];
        selectMin (i in Items : !x[i].bound()) (i) {
            tryall<cp> (b in 1..nbBins : x[i].memberOf(b))
                cp.label(x[i],b);
            onFailure {
                forall (j in Items: !x[j].bound() && weights[j] == weights[i])
                    forall (b_ in 1..nbBins : currentLoad[b_] == currentLoad[b])
                        cp.diff(x[j],b_);
            }
        }
    }
}
onRestart {
    nbBins := nbBins + 1;
}
cout << "solution using " << nbBins << " bins " << endl;
cout << "#fails: " << cp.getTotalNumberFailures() << endl;

```

Statement 12.2: CP Model for the Bin Packing Problem (bin-packing-cp.co)

The `cp.restartOnCompletion()` instruction is explained later together with the **onRestart** block. For now, assume that the number of available bins is fixed to `nbBins`. The main constraint of the problem is a **multiknapsack** constraint, linking the placement variables, the weights and the bin loads. The constraint `binpackingLB` is a redundant constraint that can be used to add more pruning in bin-packing problems.

```
cp.post(multiknapsack(x,weights,1));
cp.post(binpackingLB(x,weights,1));
```

The **using** block contains the search for the current satisfaction problem with `nbBins` available bins. It starts by ensuring that only the first `nbBins` bins are used, by constraining the load all other bins to zero.

```
forall (b in Bins : b > nbBins)
  cp.label(1[b],0);
```

During the search, the item chosen to be placed next by the variable heuristic is the one with the smallest index among the items not placed yet. Since the items are sorted in decreasing order in the data, this corresponds to selecting first the largest item not yet assigned to a bin. The search then tries to assign this item to each of the bins from 1 to `nbBins`:

```
selectMin (i in Items : !x[i].bound()) (i)
  tryall<cp> (b in 1..nbBins : x[i].memberOf(b))
    cp.label(x[i],b);
```

There are several symmetries in the bin-packing problem:

- Similar items can be swapped in any solution and produce another valid solution
- The items of any two bins in a solution can be also be swapped

We *break these symmetries dynamically*, to avoid exploring symmetrical parts of the search tree. This dynamic symmetric breaking scheme is implemented in the **onFailure** block. This block is executed on backtracking, just before trying the next value of the **tryall** instruction. If this block is

executed right after trying to place item *i* into bin *b*, this means that we cannot extend the current partial solution (that consists of the already placed items) to a valid solution, in which item *i* is placed into bin *b*.

Hence, it would be pointless to try placing the item into any other bin with the same current load as bin *b*. It would also be pointless to try placing any other item with the same weight as item *i* into any bin, whose current load is the same with the current load of bin *b*. The current bin load is computed with

```
int currentLoad[b in Bins] =
    sum(i in Items : x[i].bound() && x[i].getValue() == b) weights[i];
```

and the `onFailure` block looks like:

```
onFailure {
    forall (j in Items : !x[j].bound() && weights[j] == weights[i])
        forall (b_ in 1..nbBins : currentLoad[b_] == currentLoad[b])
            cp.diff(x[j], b_);
}
```

So far, we have ignored the `restartOnCompletion` instruction and the `onRestart` block and treated the number of available bins, `nbBins`, as fixed. Let's see now focus on these parts of the code. The instruction `restartOnCompletion` results into a restart, whenever the search has completed without finding any solution, and causes the `onRestart` block to be executed before each restart.

If the `onRestart` block is executed, this means that there is no solution using only `nbBins` bins, and we simply need to increase the number of available bins by 1 before the restarting. Note that, when restarting, any change that took place inside the `using` block is reversed, before executing the `using` block again. This means that any effect of the labeling is undone before restarting. On the other hand, the increase to the value of the `Integer` object `nbBins` carries through the restart.

When a solution is found, the search continues without executing the `onRestart` block, and the last lines print information about the number of bins used in the solution and the number of failures during the search.

12.3 Euler Knight

The main objective of this section is to illustrate the use of the `circuit` constraint. The Euler Knight problem consists in moving a knight through all the squares of a chessboard, starting and finishing on the same square and passing from each square exactly once. This is called an Euler circuit. The complete model is given on Statement [12.3](#).

The only decision variables are contained in the successor array `jump`:

```
var<CP>{int} jump[i in Chessboard](cp,Knightmoves(i));
```

where `jump[i]` is the identifier of the square visited after square `i`. The valid knight moves are encoded in the domain. The function `Knightmoves(i)` returns the set of squares reachable from square `i`. The only constraint of the problem is:

```
cp.post(circuit(jump));
```

which avoids sub-tours and ensures that the successor array represents a valid circuit. The search used is a default first-fail heuristic on the `jump` variables.

```

import cotfd;
int t0 = System.getCPUTime();
Solver<CP> cp();
function set{int} Knightmoves(int i) {
    set{int} S;
    if (i % 8 == 1)
        S = {i-15,i-6,i+10,i+17};
    else if (i % 8 == 2)
        S = {i-17,i-15,i-6,i+10,i+15,i+17};
    else if (i % 8 == 7)
        S = {i-17,i-15,i-10,i+6,i+15,i+17};
    else if (i % 8 == 0)
        S = {i-17,i-10,i+6,i+15};
    else
        S = {i-17,i-15,i-10,i-6,i+6,i+10,i+15,i+17};
    return filter(v in S)(v >= 1 && v <= 64);
}
range Chessboard = 1..64;
var<CP>{int} jump[i in Chessboard](cp,Knightmoves(i));
solve<cp>
    cp.post(circuit(jump));
using {
    labelFF(jump);
    cout << jump << endl;
}
cout << "#fail   = " << cp.getNFail() << endl;
cout << "time    = " << System.getCPUTime() - t0 << endl;

```

Statement 12.3: CP Model for the Euler Knight Problem (euler-cp.co)

12.4 Perfect Square

The problem is to fully cover a big 112×112 square with 21 different smaller squares with no overlap between squares. This problem illustrates

- how to model non-overlap constraints
- how to use redundant constraints for better pruning of the search space
- a sophisticated search, where the value is chosen before the variable and the `onFailure` block is used for more pruning of the search space

The complete COMET model is shown in Statement 12.4. The input data of the problem consists of the side length of each square and the side length of the big square. The decision variables are the bottom left x and y positions of each square:

```
var<CP>{int} x[i in Square](cp,1..s-side[i]+1);
var<CP>{int} y[i in Square](cp,1..s-side[i]+1);
```

Note that the domain of square i ranges from 1 to `s-side[i]+1`, since a x or y position larger than `s-side[i]+1` would lead to extending beyond the limits of the big square.

The first set of constraints states that there can be no overlap in space between any two squares i and j . This constraint must be literally read as: “square i is on the left, on the right, below or above square j .”

```
forall (i in Square, j in Square : i < j)
  cp.post(x[i] + side[i] <= x[j]
    || x[j] + side[j] <= x[i]
    || y[i] + side[i] <= y[j]
    || y[j] + side[j] <= y[i]);
```

The last set of constraints are not necessary, in order to find a feasible solution, since they express a property that is satisfied by all the solutions. This kind of constraints are called *redundant* or *surrogate*, because they are not necessary, but help to prune the search space.

```

import cotfd;
Solver<CP> cp();
int    s          = 112;
range Side        = 1..s;
range Square      = 1..21;
int    side[Square] = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];

var<CP>{int} x[i in Square](cp,1..s-side[i]+1);
var<CP>{int} y[i in Square](cp,1..s-side[i]+1);
solve<cp> {
    forall (i in Square, j in Square : i < j)
        cp.post(x[i] + side[i] <= x[j]
            || x[j] + side[j] <= x[i]
            || y[i] + side[i] <= y[j]
            || y[j] + side[j] <= y[i]);
    forall (p in Side) {
        cp.post(sum(i in Square)
            side[i] * ((x[i] <= p) && (x[i] >= p - side[i] + 1)) == s);
        cp.post(sum(i in Square)
            side[i] * ((y[i] <= p) && (y[i] >= p - side[i] + 1)) == s);
    }
}
using {
    forall (k in Square)
        selectMin(p in Square : !x[p].bound(), r = x[p].getMin()) (r) {
            tryall<cp>(i in filter(k in Square) (!x[k].bound() && x[k].memberOf(r)))
                cp.label(x[i],r);
            onFailure
                cp.diff(x[i],r);
        }
    forall (k in Square)
        selectMin (p in Square : !y[p].bound(), r = y[p].getMin()) (r) {
            tryall<cp>(i in filter(i in Square) (!y[i].bound() && y[i].memberOf(r)))
                cp.label(y[i],r);
            onFailure
                cp.diff(y[i],r);
        }
}
cout << x << endl;
cout << y << endl;
cout << "#choices = " << cp.getNChoice() << endl;
cout << "#fail   = " << cp.getNFail() << endl;

```

Statement 12.4: CP Model for the Perfect Square Problem (squares-cp.co)

In our example, they state that the sum of size lengths of all the squares overlapping a vertical (or horizontal) line be equal to the size length of the big square. This holds for any valid solution, since every position of the big square has to be covered.

```
forall (p in Side) {
  cp.post(sum(i in Square)
    side[i] * ((x[i] <= p) && (x[i] >= p - side[i] + 1)) == s);
  cp.post(sum(i in Square)
    side[i] * ((y[i] <= p) && (y[i] >= p - side[i] + 1)) == s);
}
```

The search strategy is to first assign all the x positions of the squares and then assign the y positions. Variables are assigned in a different way than usual. The value is chosen first, and then this value is tried on all the variables containing this value inside their domain. The value chosen is the leftmost possible position p , for the x variable of any of the squares that are not yet placed. Then this value p is tried for all the x variables that have it in their domain.

Note that, since variables (rather than values) are successively tried in the `tryall`, when backtracking and trying the next variable, it is important to inform the constraint store that it doesn't have to consider the labeling $x[i] == r$ again. Indeed, this information was acquired after completely exploring, without success, the sub-tree that corresponds to that labeling. By posting the constraint `cp.diff(x[i], r)` in the `onFailure` block, we pass this valuable information to the next sub-tree exploration.

The part of the search pertaining to the x variables is the following:

```
forall (k in Square)
  selectMin(p in Square : !x[p].bound(), r = x[p].getMin()) (r) {
    tryall<cp>(i in filter(k in Square) (!x[k].bound() && x[k].memberOf(r)))
      cp.label(x[i], r);
    onFailure
      cp.diff(x[i], r);
  }
```

```

import cotfd;
Solver<CP> cp();

range Cars                = 1..100;
range Configs             = 1..18;
range Options             = 1..5;
int lb[Options]           = [1,2,1,2,1];
int ub[Options]           = [2,3,3,5,5];
int demand[Configs]       = [5,3,7,1,10,2,11,5,4,6,12,1,1,5,9,5,12,1];
int requires[Configs,Options] = [[1,1,0,0,1],[1,1,0,1,0],...,[0,0,1,0,0]];
set{int} options[o in Options] = filter (c in Configs)(requires[c,o] == 1);

var<CP>{int} line[Cars](cp,Configs);
solve<cp>
  forall (o in Options)
    cp.post(sequence(line,demand,lb[o],ub[o],options[o]));
using
  labelFF(line);

cout << "#choices = " << cp.getNChoice() << endl;
cout << "#fail   = " << cp.getNFail() << endl;

```

Statement 12.5: CP Model for the Car Sequencing Problem ([carsequencing-cp.co](#))

12.5 Car Sequencing

This section illustrates the use of the global constraint **sequence** on a car sequencing problem. We start with some intuition on the problem. Cars in production are placed on an assembly line. Different options (radio, sun-roof, air-conditioning, etc.) are installed to the cars by moving them through the corresponding production units that install these options. The unit for each option has a production capacity that is expressed as “*m out of n*”, meaning that no more than *m* out of *n* consecutive ones can require that option.

The input of the problem is a set of demands for cars with different configurations (sets of options). The problem is to put these cars in sequence on the assembly line, while satisfying the capacity constraints for each option. The complete COMET model for solving this problem is shown in Statement 12.5, which also contains the data for a particular instance.

The data of the problem is given in the beginning of the model:

```

range Cars                = 1..100;
range Configs             = 1..18;
range Options             = 1..5;
int lb[Options]           = [1,2,1,2,1];
int ub[Options]           = [2,3,3,5,5];
int demand[Configs]       = [5,3,7,1,10,2,11,5,4,6,12,1,1,5,9,5,12,1];
int requires[Configs,Options] = [[1,1,0,0,1],[1,1,0,1,0],...,[0,0,1,0,0]];
set{int} options[o in Options] = filter (c in Configs)(requires[c,o] == 1);

```

We can see that 100 cars must be sequenced. There are 18 different configurations with five possible options. The option capacities are given in the arrays `lb` and `ub`: option `o` cannot be required by more than `lb[o]` out of `ub[o]` consecutive cars. The array `options` contains, for each option, the set of configurations requiring that option.

The decision variables of the problem represent the configuration in each position of the car sequence:

```

var<CP>{int} line[Cars](cp,Configs);

```

For each option, we use one global `sequence` constraint to express its capacity.

```

forall (o in Options)
  cp.post(sequence(line,demand,lb[o],ub[o],options[o]));

```

It imposes that:

- at most `lb[o]` out of `ub[o]` consecutive variables in the array `line` take their value from the set `options[o]`
- the number of `c` values in the sequence is equal to `demand[c]`.

The search used in the `using` block is a standard first-fail labeling (`labelFF`) on the `line` variables.

12.6 Sport Scheduling

This section shows how to solve an interesting Sport Scheduling problem. The most interesting features of the solution presented here include:

- the use of `table`, `exactly`, and `alldifferent` constraints
- the introduction of redundant variables to ease the expression of constraints
- the use static symmetry breaking constraints

The problem is to schedule an even number n of teams over $n/2$ periods and $n - 1$ weeks, under the following constraints:

- Each team must play against every other team
- A team plays exactly one game per week
- A team can play at most twice in the same period

The complete COMET model is given in Statement [12.6](#).

We now give a more detailed explanation of the model. Among the three constraints above, the most difficult to express is the constraint that each team must play against every other team. So, we create a unique identifier for each pair of teams, that represents the game between the two teams. We then store all triples (team1,team2,identifier) into a `Table<CP>` object, that will be used later, when stating the constraints:

```
tuple triple {int a1;int a2;int a3;}
set{triple} Triples();
forall (i in Teams,j in Teams: i < j)
    Triples.insert(triple(i,j,(i-1)*n + j));
Table<CP> t(all(t in Triples) t.a1,all(t in Triples) t.a2,all(t in Triples) t.a3);
```

```

import cotfd;
Solver<CP> cp();
int    n      = 12;
range Periods = 1..n/2;
range Teams   = 1..n;
range Weeks   = 1..n-1;
enum Location = {home,away};
range Games   = 1..(n/2)*n-1;

tuple triple {int a1;int a2;int a3;}
set{tuple} Triples();
forall (i in Teams, j in Teams: i < j)
    Triples.insert(triple(i,j,(i-1)*n + j));
Table<CP> t(all(t in Triples) t.a1,
            all(t in Triples) t.a2,
            all(t in Triples) t.a3);

var<CP>{int} team[Periods,Weeks,Location](cp,Teams);
var<CP>{int} game[p in Periods,w in Weeks](cp,1..n^2);
solve<cp> {
    forall (p in Periods,w in Weeks)
        cp.post(table(team[p,w,home],team[p,w,away],game[p,w],t));
    forall (w in Weeks)
        cp.post(alldifferent(all(p in Periods,l in Location) team[p,w,l]),onDomains);
    forall (p in Periods)
        cp.post(cardinality(all(i in Teams) 1,
                               all(w in Weeks,l in Location) team[p,w,l],
                               all(i in Teams) 2), onDomains);
    cp.post(alldifferent(all(p in Periods,w in Weeks) game[p,w]),onDomains);
    forall (w in Weeks.getLow()..Weeks.getUp()-1)
        cp.post(game[Periods.getLow(),w] < game[Periods.getLow(),w+1]);
}
using
    labelFF(all(p in Periods,w in Weeks) game[p,w]);

```

Statement 12.6: CP Model for the Sport Scheduling Problem (sport-scheduling-cp.co)

We then create two families of variables:

```
var<CP>{int} team[Periods,Weeks,Location] (cp,Teams);
var<CP>{int} game[p in Periods,w in Weeks] (cp,1..n^2);
```

The interpretation of these variables is the following:

- `team[p,w,home]`: the team that plays at home during period `p` and week `w`.
- `team[p,w,away]`: the team that plays away during period `p` and week `w`
- `game[p,w]`: the unique identifier for the team pair (`team[p,w,home]`,`team[p,w,away]`).

Note that, in a valid solution, for any period `p` and week `w`, `team[p,w,home]` must play against `team[p,w,away]`. Also, note that the second set of variables is not strictly necessary, since the `game[p,w]` is a function `team[p,w,home]` and `team[p,w,away]`); however, `game` variables make it much easier to express the constraint that every team must play against every other team.

The first set of constraints establishes the link between `team[p,w,home]`, `team[p,w,away]` and `game[p,w]` by stating that these variables must constitute one of the precomputed triples contained in `Table<CP> t`:

```
forall (p in Periods,w in Weeks)
  cp.post(table(team[p,w,home],team[p,w,away],game[p,w],t));
```

The following constraints enforce that each team plays exactly once during each week:

```
forall (w in Weeks)
  cp.post(alldifferent(all(p in Periods,l in Location) team[p,w,l]),onDomains);
```

Then a **cardinality** constraint is posted on each period, to express the fact a team can play at most twice at in any given period.

```
forall (p in Periods)
  cp.post(cardinality(all(i in Teams) 1,
                        all(w in Weeks,l in Location) team[p,w,l],
                        all(i in Teams) 2), onDomains);
```

At this point, the constraint that each team plays against every other team can easily expressed on the **game** variables:

```
cp.post(alldifferent(all(p in Periods,w in Weeks) game[p,w]),onDomains);
```

Note that this problem has two kind of symmetries:

- the home/away status of the two teams in a game is interchangeable
- the weeks are interchangeable

The home/away symmetries were avoided at the creation of the triples. Indeed, the triples were created so that the identifier of the home team is smaller that the identifier of the away team. The week symmetries are avoided by posting the following constraints:

```
forall (w in Weeks.getLow()..Weeks.getUp()-1)
  cp.post(game[Periods.getLow(),w] < game[Periods.getLow(),w+1]);
```

Finally, the search used is a first-fail heuristic on the **game** variables.

12.7 Radio Link Frequency Assignment

This section illustrates the use of events on CP integer variables, while solving a radio link frequency allocation problem. Frequency allocation typically aims at eliminating interference between frequencies, producing distance constraints and complex generalizations of graph coloring. A frequency must be assigned to each transmitter. Transmitters are grouped into cells and there are intra- and inter-cell distance constraints between the frequencies. The complete COMET model is given in Statement 12.7.

This is a summary of the problem's data: `Cells` and `Freqs` are the ranges of cells possible frequencies, `trans[Cells]` stores the number of transmitters in each cell, and `dist[c1,c2]` gives the minimum distance between the frequencies of transmitters in cells `c1` and `c2`.

The tuple `TT {int c; int t;}` defines the transmitter `t` inside cell `c`. The `set{TT} strans()` collects all these tuples. We also define a mapping from these tuples to the corresponding frequency variable:

```
dict{TT -> var<CP>{int}} freq();
```

The array `occ` maintains incrementally the number of transmitters using each frequency. This is done during the search, through the event mechanism. Whenever the transmitter defined by a tuple `t` is assigned a frequency `v`, in other words, `freq{t}` is bound to value `v`, the corresponding frequency counter is incremented. On the other hand, when the domain of the `freq{t}` is restored, i.e., frequencies other than `v` are also added to its domain, the counter of frequency `v` is decremented. This counter information is used later for implementing a value heuristic.

```
int occ[i in Freqs] = 0;
forall (t in strans) {
    whenever freq{t}@onValueBind(int v) occ[v]++;
    whenever freq{t}@onUnbind(int v)    occ[v]--;
}
```

The only constraints of the problem are the minimum inter- and intra cell distance constraints. In the search, variables are chosen according to a first-fail heuristic: the transmitter with the smallest domain is assigned first; in case of ties, we prefer transmitters from cells with few transmitters. The value heuristic maximizes the chance of success by trying first the most used frequencies. Recall that this information is maintained in the `occ` array.

```

import cotfd;
Solver<CP> cp();
range Cells          = 1..25;
range Freqs          = 1..256;
int   trans[Cells]   = [8,6,6,1,4,4,8,8,8,4,9,8,4,4,10,8,9,8,4,5,4,8,1,1];
int   dist[Cells,Cells] = [[16,1,1,0,0,0,0,0,1,1,1,1,1,2,2,1,1,0,0,0,2,2,1,1,1],
                           [1,16,2,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0],
                           [1,2,16,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0],
                           ...
                           [1,0,0,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,1,2,2,1,2,16]];

tuple TT {int c; int t;}
set{TT} strans();
forall (c in Cells, t in 1..trans[c])
    strans.insert(new TT(c,t));
dict{TT -> var<CP>{int}} freq();
forall (t in strans)
    freq{t} = new var<CP>{int}(cp,Freqs);
int occ[i in Freqs] = 0;
forall (t in strans) {
    whenever freq{t}@onValueBind(int v) occ[v]++;
    whenever freq{t}@onUnbind(int v)   occ[v]--;
}
solve<cp> {
    forall (c in Cells, t1 in 1..trans[c], t2 in t1+1..trans[c])
        cp.post(abs(freq{new TT(c,t1)}-freq{new TT(c,t2)}) >= dist[c,c]);
    forall (c1 in Cells, c2 in Cells : c1 < c2 && dist[c1,c2] > 0)
        forall (t1 in 1..trans[c1], t2 in 1..trans[c2])
            cp.post(abs(freq{new TT(c1,t1)}-freq{new TT(c2,t2)}) >= dist[c1,c2]);
}
using {
    forall (t in strans) by (freq{t}.getSize(), trans[t.c])
        tryall<cp> (f in Freqs : freq{t}.memberOf(f)) by (-occ[f])
            cp.label(freq{t},f);
}
cout << freq << endl;
cout << "#freqs = " << sum(i in Freqs) (occ[i]!=0) << endl;
cout << "#fails = " << cp.getNFail() << endl;

```

Statement 12.7: CP Model for the Radio Link Frequency Assignment Problem ([frequency-cp.co](#))

12.8 Steel Mill Slab Design

In this problem we are asked to assign orders of different colors and sizes to slabs of different capacities, so that the total loss of steel incurred is minimized and so that at most two different colors are present in each slab.

We now describe a small instance. The available capacity options are 7, 10 and 12. There are five orders with colors and sizes given in the following table.

| | | | | | |
|-------|---|---|---|---|---|
| Order | 1 | 2 | 3 | 4 | 5 |
| Size | 5 | 8 | 3 | 4 | 6 |
| Color | 1 | 3 | 2 | 1 | 2 |

An important observation in solving this problem is that the capacity chosen for each slab is the minimum capacity that can accommodate its load (since we minimize the total loss). A solution using three slabs is given next (the same capacity could be used more than once). The total loss of this solution is 3, because there is a loss of 2 in the second slab and a loss of 1 in the third slab.

| | | | |
|---------------|-------|----|---|
| Slab Capacity | 12 | 10 | 7 |
| Orders | 1,3,4 | 2 | 5 |
| Load | 12 | 8 | 6 |
| Loss | 0 | 2 | 1 |

The data of the problem can be summarized as follows:

```
range Orders;
range Slabs;
range Colors;
range Caps;
int    weight[Orders];
int    color[Orders];
int    capacities[Caps];
```

We preprocess these data, to compute the set of orders corresponding to each color (`colorOrders`), the maximum capacity (`maxCap`) and the resulting steel loss for each possible load.

```
set{int} colorOrders[c in Colors] = filter(o in Orders) (color[o] == c);
int maxCap = max(i in Caps) capacities[i];
int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;
```

For the small example given in the problem description, the `loss` array would contain:

| | | | | | | | | | | | | | |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| load | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| loss | 0 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 2 | 1 | 0 | 1 | 0 |

This array is very useful, since it gives in constant time the slab for a given load. A first CP model is given in Statement 12.8. Note that the data and preprocessing parts are omitted.

For each order, we introduce a decision variable `x`, giving the slab it is placed into. We also introduce a variable `l`, that represents the load of each slab, i.e., the sum of sizes of the orders placed into the slab. Note that we only need to branch on the decision variables `x`. The other set of variables is used with the `loss` array, to easily express the objective function to minimize:

```
minimize<cp>
    sum(s in Slabs) loss[l[s]]
```

The `subject to` block contains the problem constraints. A `multiknapsack` constraint links the order placement variables, `x[o]`, their size `weight[o]`, and the load of the slabs:

```
cp.post(multiknapsack(x,weight,l));
```

```

import cotfd;
Solver<CP> cp();
var<CP>{int} x[Orders](cp,Slabs);
var<CP>{int} l[Slabs](cp,0..maxCap);
minimize<cp>
    sum(s in Slabs) loss[l[s]]
subject to {
    cp.post(multiknapsack(x,weight,l));
    forall (s in Slabs)
        cp.post(sum(c in Colors) (or(o in colorOrders[c])(x[o] == s)) <= 2);
}
using {
    forall (o in Orders : !x[o].bound()) by (x[o].getSize(),-weight[o])
        tryall<cp> (s in Slabs)
            cp.label(x[o],s);
    cout << "#choices = " << cp.getNChoice() << endl;
    cout << "#fails   = " << cp.getNFail() << endl;
}
cout << x << endl;

```

Statement 12.8: CP Model for the Steel Mill Slab Problem ([steelmillslab-cp.co](#))

The other set of constraints ensures that at most two colors are present in each slab:

```
forall (s in Slabs)
  cp.post(sum(c in Colors) (or(o in colorOrders[c])(x[o] == s)) <= 2);
```

The `using` block implements the non-deterministic search. We request that each variable `x[o]` be labeled. The variable heuristic is implemented with the `by`: the next order to place is the one with the smallest domain size and ties are broken in favor of larger orders. Then, an alternative is created for each slab. Slab values are tried in increasing order.

```
forall (o in Orders : !x[o].bound()) by (x[o].getSize(),-weight[o])
  tryall<cp> (s in Slabs)
    cp.label(x[o],s);
```

Breaking Symmetries

Given any solution, we can construct several different solutions by simply changing the order of the slabs. These symmetries can be avoided by posting constraints to enforce that the loads or the losses of the slabs be decreasing. The problem with this approach is the risk of bad interaction with the heuristic.

A better approach is to break symmetries dynamically during the search: For a given order, rather than trying all possible slabs, we only try the non-empty slabs and at most one empty slab. The idea is that empty slabs are completely equivalent, and trying several of them would lead to completely equivalent sub-problems. The new modified search, with dynamic symmetry breaking is given next:

```
forall (o in Orders : !x[o].bound()) by (x[o].getSize(),-weight[o]) {
  int ms = max(0,maxBound(x));
  tryall<cp> (s in Slabs : s <= ms + 1)
    cp.label(x[o],s);
}
```

Improving the Value Heuristic with Events

The value heuristic of trying the slabs in arbitrary order can fail to quickly drive the search towards good solutions. For this particular problem, a better idea is to place the next order so that the loss of the resulting partial solution is minimized.

To efficiently compute the delta of loss for each alternative, we need to incrementally maintain the load of each slab in the current solution, and restore it upon backtracking. This can be done declaratively using the event notifications `onValueBind` and `onUnbind` for CP variables:

```
int load[Slabs] = 0;
forall (o in Orders) {
  whenever x[o].onValueBind(int s)
    load[s] += weight[o];
  whenever x[o].onUnbind(int s)
    load[s] -= weight[o];
}
```

The `tryall` is then modified with a `by` to sort the tried values, `s`, according to the value of $(\text{loss}[\text{load}[s] + \text{weight}[o]] - \text{loss}[\text{load}[s]])$, which represents the delta loss in the partial solution, if order `o` is placed into slab `s`.

```
minimize<cp>
  sum(s in Slabs) loss[l[s]]
subject to {
  ...
}
using
forall (o in Orders : !x[o].bound()) by (x[o].getSize(), -weight[o]) {
  int ms = max(0, maxBound(x));
  tryall<cp> (s in Slabs : s <= ms + 1 && x[o].memberOf(s))
    by (loss[load[s] + weight[o]] - loss[load[s]])
    cp.label(x[o], s);
}
```

Transforming the Model into a Large Neighborhood Search (LNS)

LNS is a local search method using constraint programming to explore a *large* neighborhood. The neighborhood of the current solution is obtained by relaxing the problem, usually by restoring the domain of some of the variables. COMET offers an easy way of transforming a CP model into a LNS: The only thing necessary to implement is a **restart** block, in which the relaxation procedure is implemented. Of course, we need to ask the solver to do LNS, rather than a complete depth first search. We do this using the **lnsOnFailure** method, and specifying the failure limit for each restart.

In this example, the relaxation procedure is to fix a random 90% of the orders in the last solution. Notice the **getSnapshot** method, which returns a variable's labeling in a given **Solution** object **s**.

```
UniformDistribution dist(1..100);
cp.lnsOnFailure(1000);
minimize<cp>
    sum(s in Slabs) loss[l[s]]
subject to {
    ...
}
using {
    ...
}
onRestart {
    cout << "restart" << endl;
    Solution s = cp.getSolution();
    if (s != null)
        forall (o in Orders)
            if (dist.get() <= 90)
                cp.post(x[o] == x[o].getSnapshot(s));
}
```

The improved model with LNS is implemented in `steelmillslab-cp-lns.co`.

12.9 Eternity II

In this section, we are focusing on Eternity II (E2), an edge matching puzzle. We will see:

- how to write a CP model for this edge matching puzzle
- an application of `table` constraints
- a heuristic that fixes two kinds of variables in alternating order
- how to make a dual model, channel it with element constraints, and use it in the value heuristic
- the use of `restart` to rapidly reconsider the first choices in the search tree

We now give a description of the puzzle. We are given $n \times m$ square pieces of side 1 and we are asked to place them on a $n \times m$ rectangular board. Each piece has a colored side and any two adjacent sides must have a matching color. The border color must be 0. The data, along with some preprocessing of the problem, is given in Statement [12.9](#).

Each piece is described by a tuple

```
tuple piece {int up; int right; int down; int left;}
```

giving the colors of the piece, in clockwise order (up, right, down, left), when the piece is not rotated. The `piece` tuples of all the pieces are stored in an array:

```
piece pieces[ Pieces ] = null; //to be read from input file
```

```

import cotfd;
Solver<CP> cp();

int      nbLines; //to be read from input file
int      nbCols;  //to be read from input file
range    Lines   = 0..nbLines-1;
range    Cols    = 0..nbCols-1;
range    Pieces  = 0..nbLines*nbCols-1;

tuple    piece {int up; int right; int down; int left;}
piece    pieces[Pieces] = null; //to be read from input file
set{int}  sides(); //to be read from input file

tuple triplet {int a; int b; int c;}

set{triplet} upT();
set{triplet} rightT();
set{triplet} downT();
set{triplet} leftT();

forall (p in Pieces) {
    upT.insert(new triplet(p,0,pieces[p].up));
    upT.insert(new triplet(p,1,pieces[p].right));
    upT.insert(new triplet(p,2,pieces[p].down));
    upT.insert(new triplet(p,3,pieces[p].left));

    rightT.insert(new triplet(p,0,pieces[p].right));
    rightT.insert(new triplet(p,1,pieces[p].down));
    rightT.insert(new triplet(p,2,pieces[p].left));
    rightT.insert(new triplet(p,3,pieces[p].up));

    downT.insert(new triplet(p,0,pieces[p].down));
    downT.insert(new triplet(p,1,pieces[p].left));
    downT.insert(new triplet(p,2,pieces[p].up));
    downT.insert(new triplet(p,3,pieces[p].right));

    leftT.insert(new triplet(p,0,pieces[p].left));
    leftT.insert(new triplet(p,1,pieces[p].up));
    leftT.insert(new triplet(p,2,pieces[p].right));
    leftT.insert(new triplet(p,3,pieces[p].down));
}

```

Statement 12.9: CP Model for Eternity II (Part 1/3) Data and Preprocessing (eternity-cp.co)

The following set of triples contains some precomputation on the pieces, that will be useful in the CP model. For each piece, `upT` gives the top color, if the piece is rotated 0,1,2 or 3 quarter(s) counter-clockwise. This is stored in a triple with fields `a`: the id of the piece, `b`: the number of quarter counter-clockwise rotations, `c`: the color. The same computation is done for the right, down and left colors.

```
tuple triplet {int a; int b; int c;}

set{triplet} upT();
set{triplet} rightT();
set{triplet} downT();
set{triplet} leftT();

forall (p in Pieces) {
    upT.insert(new triplet(p,0,pieces[p].up));
    upT.insert(new triplet(p,1,pieces[p].right));
    upT.insert(new triplet(p,2,pieces[p].down));
    upT.insert(new triplet(p,3,pieces[p].left));

    rightT.insert(new triplet(p,0,pieces[p].right));
    rightT.insert(new triplet(p,1,pieces[p].down));
    rightT.insert(new triplet(p,2,pieces[p].left));
    rightT.insert(new triplet(p,3,pieces[p].up));

    downT.insert(new triplet(p,0,pieces[p].down));
    downT.insert(new triplet(p,1,pieces[p].left));
    downT.insert(new triplet(p,2,pieces[p].up));
    downT.insert(new triplet(p,3,pieces[p].right));

    leftT.insert(new triplet(p,0,pieces[p].left));
    leftT.insert(new triplet(p,1,pieces[p].up));
    leftT.insert(new triplet(p,2,pieces[p].right));
    leftT.insert(new triplet(p,3,pieces[p].down));
}
```

In what follows we give a detailed explanation of the COMET model for this problem shown in Statements 12.10 and 12.11. The second statement contains the **solve** block of the model, which is omitted from the first statement due to space limitations.

```

var<CP>{int}    up[Lines,Cols](cp,sides);
var<CP>{int}    right[Lines,Cols](cp,sides);
var<CP>{int}    down[Lines,Cols](cp,sides);
var<CP>{int}    left[Lines,Cols](cp,sides);
var<CP>{int}    ori[Lines,Cols](cp,0..3);
var<CP>{int}    id[Lines,Cols](cp,Pieces);

var<CP>{int}[] ori_ = all(1 in Lines,c in Cols) ori[l,c];
var<CP>{int}[] id_  = all(1 in Lines,c in Cols) id[l,c];
var<CP>{int}    pos_[Pieces](cp,Pieces);

cp.restartOnFailureLimit(3000);
solve<cp> {
    ...
}
using {
    while (!bound(id))
        selectMin (i in id_.getRange() : !id_[i].bound()) (id_[i].getSize()) {
            tryall<cp> (p in Pieces : id_[i].memberOf(p)) by(pos_[p].getSize())
                cp.label(id_[i],p);
            tryall<cp> (o in 0..3)
                cp.label(ori_[i],o);
        }
}
onRestart
    cout << "restart, total #fails:" << cp.getTotalNumberFailures() << endl;
    cout << "#fails since last restart: " << cp.getNFail() << endl;
    cout << "#fails total:" << cp.getTotalNumberFailures() << endl;

```

Statement 12.10: CP Model for Eternity II (Part 2/3) (eternity-cp.co)

```

solve<cp> {
  forall (l in Lines, c in Cols) {
    Table<CP> tableUp(all(t in upT) (t.a),
                     all(t in upT) (t.b),
                     all(t in upT) (t.c));
    Table<CP> tableRight(all(t in rightT) (t.a),
                        all(t in rightT) (t.b),
                        all(t in rightT) (t.c));
    Table<CP> tableDown(all(t in downT) (t.a),
                       all(t in downT) (t.b),
                       all(t in downT) (t.c));
    Table<CP> tableLeft(all(t in leftT) (t.a),
                       all(t in leftT) (t.b),
                       all(t in leftT) (t.c));
    cp.post(table(id[l,c],ori[l,c],up[l,c],tableUp));
    cp.post(table(id[l,c],ori[l,c],right[l,c],tableRight));
    cp.post(table(id[l,c],ori[l,c],down[l,c],tableDown));
    cp.post(table(id[l,c],ori[l,c],left[l,c],tableLeft));
  }

  forall (l in Lines, c in Cols.getLow()..Cols.getUp()-1)
    cp.post(right[l,c] == left[l,c+1]);
  forall (l in Lines.getLow()..Lines.getUp()-1, c in Cols)
    cp.post(down[l,c] == up[l+1,c]);

  forall (l in Lines) {
    cp.label(left[l,Cols.getLow()],0);
    cp.label(right[l,Cols.getUp()],0);
  }

  forall (c in Cols) {
    cp.label(up[Lines.getLow()],c,0);
    cp.label(down[Lines.getUp()],c,0);
  }

  cp.post(bijection(id_),onDomains);
  forall (i in Pieces)
    cp.post(id_[pos_[i]] == i);
}

```

Statement 12.11: CP Model for Eternity II (Part 3/3) (eternity-cp.co)

The variables characterizing the problem are:

```
var<CP>{int} up[Lines,Cols](cp,sides);
var<CP>{int} right[Lines,Cols](cp,sides);
var<CP>{int} down[Lines,Cols](cp,sides);
var<CP>{int} left[Lines,Cols](cp,sides);
var<CP>{int} ori[Lines,Cols](cp,0..3);
var<CP>{int} id[Lines,Cols](cp,Pieces);
```

Their semantics are quite intuitive:

- `up[l,c]` represents the top color of the piece coming in position `l,c` of the board
- `right[l,c]`, `down[l,c]`, `left[l,c]` have similar semantics
- `ori[l,c]` represents the orientation of the piece in position `l,c`
- `id[l,c]` is the identifier of the piece in position `l,c`

To facilitate the search, we introduce variables `id_` and `ori_`, which are simply flattened versions of the corresponding matrix variables, generated with the **all** operator:

```
var<CP>{int}[] ori_ = all(l in Lines,c in Cols) ori[l,c];
var<CP>{int}[] id_ = all(l in Lines,c in Cols) id[l,c];
```

We also introduce the dual viewpoint variables `pos_`:

```
var<CP>{int} pos_[Pieces](cp,Pieces);
```

Variable `pos_[p]` is the position on the board of the piece `p`. Positions are uniquely identified from top left to bottom right on the range $0..(n \times m) - 1$.

The decision variables are, essentially, the two vectors `id_` and `ori_`. All other variables are linked to them through the constraints in the **solve** block, in Statement [12.11](#). This means that, in a feasible solution, if `id_` and `ori_` are bound, all remaining variables are also bound.

The first set of constraints makes the link between the decision variables and the color variables `up`, `right`, `down`, `left`, using `table` constraints. The first `table` constraint enforces that the three variables (`id[l,c]`, `ori[l,c]`, `up[l,c]`) related to the position `l,c` constitute one of the triples given in the table object `tableUp`. The next three constraints establish the equivalent for the right, down and left colors.

```
forall (l in Lines,c in Cols) {
    Table<CP> tableUp(all(t in upT) (t.a),
                      all(t in upT) (t.b),
                      all(t in upT) (t.c));
    Table<CP> tableRight(all(t in rightT) (t.a),
                         all(t in rightT) (t.b),
                         all(t in rightT) (t.c));
    Table<CP> tableDown(all(t in downT) (t.a),
                        all(t in downT) (t.b),
                        all(t in downT) (t.c));
    Table<CP> tableLeft(all(t in leftT) (t.a),
                        all(t in leftT) (t.b),
                        all(t in leftT) (t.c));
    cp.post(table(id[l,c],ori[l,c],up[l,c],tableUp));
    cp.post(table(id[l,c],ori[l,c],right[l,c],tableRight));
    cp.post(table(id[l,c],ori[l,c],down[l,c],tableDown));
    cp.post(table(id[l,c],ori[l,c],left[l,c],tableLeft));
}
```

The constraints that follow are the edge matching constraints between adjacent pieces:

```
forall (l in Lines, c in Cols.getLow()..Cols.getUp()-1)
    cp.post(right[l,c] == left[l,c+1]);
forall (l in Lines.getLow()..Lines.getUp()-1, c in Cols)
    cp.post(down[l,c] == up[l+1,c]);
```

The constraint that the border color is zero is expressed with the lines:

```
forall (l in Lines) {
    cp.label(left[l, Cols.getLow()], 0);
    cp.label(right[l, Cols.getUp()], 0);
}

forall (c in Cols) {
    cp.label(up[Lines.getLow(), c], 0);
    cp.label(down[Lines.getUp(), c], 0);
}
```

The constraint that every piece is used only once is expressed with a `bijection` constraint, stating that `id_` constitutes a permutation of the range `id_.getRange()`:

```
cp.post(bijection(id_), onDomains);
```

Finally the channeling between the dual and the primal model is achieved with element constraints:

```
forall (i in Pieces)
    cp.post(id_[pos_[i]] == i);
```

The search used is a first-fail strategy. At every iteration of the search, we attempt to place a piece into a position of the board and fix its orientation (similarly to playing the puzzle by hand). The position is selected according to its domain size (first-fail strategy):

- select the position that can accommodate the fewest number of piece
- try to place first in this position the pieces that can be placed in the fewest number of positions (using the domain size of dual variables `pos_[p].getSize()`)

The following segment of the **using** block implements this strategy:

```
selectMin (i in id_.getRange() : !id_[i].bound()) (id_[i].getSize()) {  
  tryall<cp> (p in Pieces : id_[i].memberOf(p)) by(pos_[p].getSize())  
    cp.label(id_[i],p);  
}
```

Every time a piece is chosen for the selected position, its orientation is immediately fixed with the following lines, that basically tries all four possible rotations:

```
tryall<cp> (o in 0..3)  
  cp.label(ori_[i],o);
```

Note that, for the early choices in the search tree, we don't have a lot of information about the right position to select and/or the right piece to instantiate. Inside a **while** loop, it is preferable to use a **selectMin** rather than a **forall**, so that ties are broken randomly. Randomized heuristics are useful when combined with a restart strategy.

In our example, the search restarts every 3000 failures. This means that two searches will never be the same. The idea of a restart is to avoid being trapped in a bad region of the search tree for too long, and never reconsider the early choices of a previous search. The **restart** block on [Statement 12.10](#) is simply used to print information on the number of failures (backtracks) encountered so far.

Chapter 13

Constraint Programming Techniques

This chapter attempts to offer a better understanding of different techniques for efficiently solving constraint programming problems. It starts by describing some of the fundamental issues, such as tree exploration, choosing the right decision variables and variable and value selection heuristics, and then discusses issues such as dynamic symmetry breaking, large neighborhood search, restarts and hybridization. When solving a problem with Constraint Programming one should always keep in mind the equation

$$CP = Modeling + Search$$

Modeling, i.e., choosing the right variables and constraints, is a key component to successfully solve a problem. Unfortunately, most of the time, it is not enough: knowledge of the problem and its structure must also be incorporated, to rapidly drive the search towards (good) solutions. CP is an exhaustive exact method based on search tree exploration. The search is responsible for the shape of the tree and the way to explore it.

The next few sections draw from our experience in designing efficient search for satisfaction and optimization problems. They present some principles or guidelines that we try to follow, when faced with the design of a heuristic for a non-deterministic search. More precisely, we focus our attention on what to write in the **using** block, to efficiently solve a problem:

```
import cotfd;
Solver<CP> cp();
// declare variables and domains
solve<cp> {
    // state constraints
}
using {
    // non-deterministic search
}
```

13.1 Non-Deterministic Search

Assume we want to assign the vector \mathbf{x} of binary variables and to find the first feasible assignment. The following example illustrates how to do this using the **try** instruction.

```
import cotfd;
Solver<CP> cp();
range n = 1..3;
var<CP>{int} x[n](cp,0..1);
solve<cp> {
}
using {
  forall (i in n)
    try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
  cout << x << endl;
}
```

The output is

$\mathbf{x}[1,1,1]$

The **try** instruction defines a choice point with two alternatives (binary search tree). In the left alternative, variable $\mathbf{x}[i]$ is assigned the value 1, whereas in the right alternative it assigned the value 0. Since there are no constraints in the previous example, the depth first search, which is the default, dives directly to the leftmost solution, that is $\mathbf{x}[1,1,1]$.

The concept of non-deterministic search will become more clear in the next example. Non-deterministic means that the choice of alternatives, and variables and values implied in them, is not fixed in advance, but determined on the fly by a heuristic branching algorithm. This branching algorithm may or may not be randomized, but non-deterministic search does not necessarily mean randomized search; it rather means that the branching decision is taken inside each node, not necessarily based on past decisions.

The following example illustrates a randomized non-deterministic search, where the value for the left and right alternative is chosen randomly. This search is randomized, because of the use of a random distribution to decide on the values (0 or 1) to assign to the left and right alternatives. It is also non-deterministic because this decision is taken inside each node.

```
import cotfd;
Solver<CP> cp();
range n = 1..3;
var<CP>{int} x[n](cp,0..1);
UniformDistribution dist(0..1);
solve<cp> {
}
using {
  forall (i in n) {
    int v = dist.get();
    try<cp> cp.post(x[i] == v);| cp.post(x[i] != v);
  }
  cout << x << endl;
}
```

One of the possible random outputs is:

`x[1,0,1]`

A solution is a node where every variable has a non-empty domain and with no alternative left to extend the node. The depth-first search stops, as soon as either a solution is found, or the search is exhausted because of the `solve` instruction. Alternatively, one can request all possible solutions with the `solveall` instruction.

The following example illustrates the effect of the `solveall` using the default depth-first search exploration strategy.

```
import cotfd;
Solver<CP> cp();
range n = 1..3;
var<CP>{int} x[n](cp,0..1);
solveall<cp> {
}
using {
  forall (i in n)
    try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
  cout << x << endl;
}
```

The output of the example above is:

```
x[1,1,1]
x[1,1,0]
x[1,0,1]
x[1,0,0]
x[0,1,1]
x[0,1,0]
x[0,0,1]
x[0,0,0]
```

The search tree and the order, in which the nodes are visited, is represented on Figure [13.1](#). The constraints of the alternatives are shown on the branches. The leaves of this tree correspond to the solutions and, every time a solution is found, it is printed to the screen.

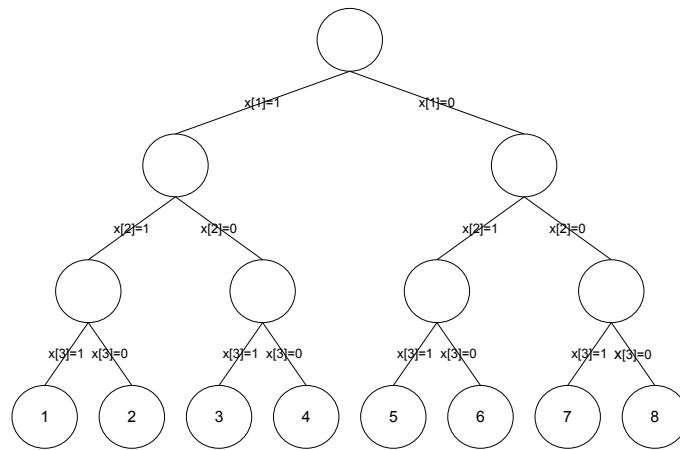


Figure 13.1: Depth-First Search Exploration: Numbers inside leaf nodes show the order, in which the corresponding solution is encountered

A search tree can be viewed as an and-or tree. This means that logical constraints, such as the **or** constraint, can be enforced by creating alternatives during the search. Consider the following program with the single logical constraint: there are two or four binary variables equal to 1. This is stated with the following **post** command:

```
cp.post((sum(i in n) x[i] == 2) || (sum(i in n) x[i] == 4))
```

This is the complete COMET code:

```
import cotfd;
Solver<CP> cp();
range n = 1..5;
var<CP>{int} x[n](cp,0..1);
solveall<cp>
  cp.post((sum(i in n) x[i] == 2) || (sum(i in n) x[i] == 4));
using {
  forall (i in n : !x[i].bound())
    try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
  cout << x << endl;
}
```

The same logical **or** constraint can also be enforced by creating two alternatives at the root node of the tree. Each member of the **or** constraint is posted as a different alternative:

- in the left alternative the sum of the variables is 2
- in the right alternative the sum is 4

The set of solutions created by the following program is exactly the same as in the previous example. Also, note that the effect of posting a constraint during the search is reversible: the constraint is removed upon backtracking.

```
import cotfd;
Solver<CP> cp();
range n = 1..5;
var<CP>{int} x[n](cp,0..1);
solveall<cp> {
}
using{
  try<cp> cp.post(sum(i in n) x[i] == 2);| cp.post(sum(i in n) x[i] == 4);
  forall (i in n : !x[i].bound())
    try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
  cout << x << endl;
}
```

The **forall** structure is not the only way to assign a vector of variables. For example, one could use the **while** loop, until all variables are bound:

```
import cotfd;
Solver<CP> cp();
range n = 1..3;
var<CP>{int} x[n](cp,0..1);
solveall<cp> {
}
using {
  while (!bound(x))
    select (i in n : !x[i].bound())
      try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
  cout << x << endl;
}
```

The **tryall** instruction should be used instead of **try**, to create more than one alternatives, for example when labeling a variable with all its possible values:

```
import cotfd;
Solver<CP> cp();
range n = 1..3;
range d = 0..2;
var<CP>{int} x[n](cp,d);
solveall<cp>
  cp.post(sum(i in n) x[i] == 5);
using {
  forall (i in n : !x[i].bound())
    tryall<cp> (v in d)
      cp.post(x[i] == v);
  cout << x << endl;
}
```

The corresponding output is:

```
x[1,2,2]
x[2,1,2]
x[2,2,1]
```

Depth-first Search is not the only strategy available in COMET, for exploring the search tree. The exploration strategy to be used is determined by the search controller of the CP solver. The actual strategy used in the search can be retrieved with the method `getSearchController`. In COMET, all search controllers implement the `SearchController` interface.

As illustrated in the following code fragment, the default search controller is the depth-first search controller, `DFSController`.

```
import cotfd;
Solver<CP> cp();
DFSController dfs = (DFSController) cp.getSearchController(); //default DFS
```

The user can specify different search strategies, using the `setSearchController` instruction. For

example, we can easily specify that bounded discrepancy search (BDS) should be performed rather than the default depth-first search. The principle of bounded discrepancy search is to successively increase the number of right alternatives (“discrepancies”) allowed along a branch.

The following program illustrates bounded discrepancy search for a binary tree of three variables. The solutions are discovered and printed by increasing number of discrepancies. The discrepancy value of each solution node is represented in Figure [13.2](#).

```
import cotfd;
Solver<CP> cp();
cp.setSearchController(BDSController(cp)); //replace DFS by BDS
range n = 1..3;
var<CP>{int} x[n](cp,0..1);
solveall<cp> {
}
using {
  forall (i in n)
    try<cp> cp.post(x[i] == 1);| cp.post(x[i] == 0);
    cout << x << endl;
}
```

The output of the above fragment is:

```
x[1,1,1] discrepancy: 0
x[1,1,0] discrepancy: 1
x[1,0,1] discrepancy: 1
x[0,1,1] discrepancy: 1
x[0,1,0] discrepancy: 2
x[0,0,1] discrepancy: 2
x[1,0,0] discrepancy: 2
x[0,0,0] discrepancy: 3
```

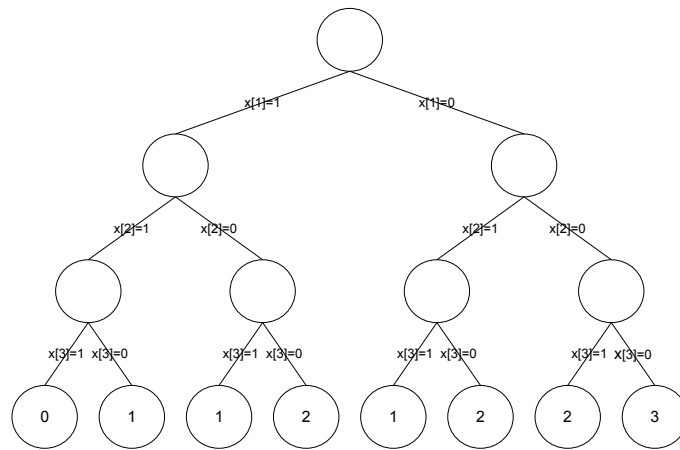


Figure 13.2: Bounded Discrepancy Search: Numbers inside leaf nodes gives the number of discrepancies (the number of right alternatives) leading to the corresponding solution

13.2 Choosing the Right Decision Variables

In modeling a problem, the choice of variables should obviously facilitate the expression of the problem's constraints, but it should also facilitate the search process. Usually, several choices are possible, but the design of the search makes it easy to argue in favor of one or the other model. We illustrate this through two examples: a bin-packing problem and the queens problem.

Ideally, when designing a search, each variable should be assigned a single value in the end. It is often a good idea to introduce redundant variables, to facilitate the constraint statement. This does not mean that all variables declared in the model should be branching variables (decision variables) during the search. Typically, redundant variables are bound by the constraints of the problem, once the decision variables become bound. This is also illustrated in the examples of this section.

13.2.1 Bin-Packing Problem

Statement 13.1 gives a model for a small bin-packing problem, where 16 items must be packed into six bins of capacity 8. The variables of the problem are

```
var<CP>{int} P[Items,Bins](cp,0..1);
var<CP>{int} L[Bins](cp,0..capacity);
```

where $P[i,b]$ is equal to 1, if item i is placed into bin b , and 0 otherwise, and $L[b]$ represents the load of bin b . The constraints express that each item must be placed in exactly one bin:

```
forall (i in Items)
  cp.post(sum(b in Bins) P[i,b] == 1);
```

and that the load of each bin must equal the sum of weights of the items placed into the bin:

```
forall (b in Bins)
  cp.post(L[b] == sum(i in Items) P[i,b]*weight[i]);
```

```
import cotfd;
range Items = 1..16;
range Bins = 1..6;
int capacity = 8;
int weight[Items] = [1,1,2,2,2,2,2,3,3,3,3,4,4,5,5,6];
Solver<CP> cp();
var<CP>{int} P[Items,Bins](cp,0..1);
var<CP>{int} L[Bins](cp,0..capacity);
solve<cp> {
    forall (i in Items)
        cp.post(sum(b in Bins) P[i,b] == 1);
    forall (b in Bins)
        cp.post(L[b] == sum(i in Items) P[i,b]*weight[i]);
}
using
    forall (b in Bins, i in Items)
        try<cp> cp.post(P[i,b] == 0); | cp.post(P[i,b] == 1);
```

Statement 13.1: Naive CP Model for a Small Bin-Packing Problem

The non-deterministic search tries to assign all binary variables of matrix P . This search is (almost) equivalent to using the instruction `label(P)`.

```
forall (i in Items, b in Bins)
  try<cp> cp.post(P[i,b] == 0); | cp.post(P[i,b] == 1);
```

Note that it would be pointless to assign the load variables, L , during the search. Indeed, the L variables are linked to the P variables, through the constraints

$$L[b] == \text{sum}(i \text{ in } \text{Items}) P[i,b] * \text{weight}[i]$$

This means that, when the P are bound the L will also be bound. We say that the P variables are the *decision variables*, while L variables are *dependent variables*.

If no constraints were posted, the size of the complete search tree would be $2^{(16 * 8)} \approx 8 * 10^{28}$. Fully exploring such a tree is totally intractable. Actually, a decision taken by an alternative `cp.post(P[i,b] == 0)` has almost no impact on the propagation; on the other hand, a decision `cp.post(P[i,b] == 1)` is much more radical, since it causes all variables $P[i,b']$ with $b' \neq b$ to be set to 0 by the constraint:

$$\text{sum}(b \text{ in } \text{Bins}) P[i,b] == 1$$

This observation leads to the design of a different search scheme, that uses a **tryall** structure to decide for each item, the bin it must be placed into:

```
forall (i in Items)
  tryall<cp> (b in Bins) cp.post(P[i,b] == 1);
```

This search can potentially create a search tree of size $6^{16} \approx 3 * 10^{12}$, which is much smaller than the previous search that assigned all the P variables.

This improved search scheme leads in turn to a more efficient model, in which binary variables P are replaced by a vector X of decision variables, so that variable $X[i]$ corresponds to the bin into which item i is placed. There are many advantages of this model over the binary variable model:

- it doesn't need constraints to state that each item is placed into exactly one bin
- as we saw in the bin-packing model of Section 12.2, decision variables $X[i]$ make it possible to design a first-fail heuristic and to dynamically break symmetries during the search
- moreover, using a default search, such as `label` and `labelFF`, would not blow up on X as it would on P

We are going to revisit first-fail heuristics and dynamic symmetry breaking in Sections 13.3 and 13.5. The following block of code shows the modifications to the original naive model:

```
var<CP>{int} L[Bins](cp,0..capacity);
var<CP>{int} X[Items](cp,Bins);
solve<cp>
  forall (b in Bins)
    cp.post(L[b] == sum(i in Items) (X[i]==b) * weight[i]);
using
  forall (i in Items)
    tryall<cp> (b in Bins) cp.post(X[i]==b);
```

13.2.2 Queens Problem

We now revisit the queens problem presented in Section 9.5. The problem is to place n queens on a $n \times n$ chess-board such that no two queens attack each other, i.e., no two queens can lie on the same line, column or diagonal. A first COMET model is given in Statement 13.2. For each position of the chess-board, it introduces a binary variable that is equal to 1 if a queen is placed there:

```
var<CP>{int} Q[i in S, j in S](cp, 0..1);
```

The search simply labels all these variables

```
forall (i in S, j in S)
  try<cp> cp.post(Q[i,j] == 0); | cp.post(Q[i,j] == 1);
```

Assigning the binary variables can potentially lead to the generation of 2^{400} alternatives. Once again, note that assigning a binary variable to 0 does not have a significant impact on the propagation. A much better choice of decision variables follows from the problem structure: since we have n queens and n rows, and no two queens can be placed into the same row, each row must contain exactly one queen. Therefore, we can create one variable for each row, to represent the column of the row's queen.

These decision variables allow to simplify and improve the way constraints are expressed:

- the constraint “exactly one queen in each row” is now enforced by the choice of variables
- the 120 sum constraints posted before are replaced by three **alldifferent** constraints resulting to a stronger pruning

The improved model is shown on Statement 13.3. From the search perspective, the potential size of the search tree is reduced to 20^{20} instead of 2^{400} , and each alternative chosen during the search has a strong impact on the propagation of the domains.

```

import cotfd;
Solver<CP> cp();
int n = 20;
range S = 1..n;
var<CP>{int} Q[i in S, j in S](cp, 0..1);
solve<cp> {
  forall (i in S) {
    cp.post(sum(j in S) Q[i, j] == 1 );
    cp.post(sum(j in S) Q[j, i] == 1 );
  }
  forall (i in S) {
    cp.post(sum(j in 1..i) Q[j, j+(n-i)] <= 1);
    cp.post(sum(j in 1..i) Q[j+(n-i), j] <= 1);
    cp.post(sum(j in 1..i) Q[-j+i+1, j] <= 1);
    cp.post(sum(j in 1..i) Q[n-j+1, n-i+j] <= 1);
  }
}
using
  forall (i in S, j in S)
    try<cp> cp.post(Q[i, j] == 0); | cp.post(Q[i, j] == 1);

```

Statement 13.2: Naive CP Model for the Queens Problem (queens-cp-naive.co)

```

import cotfd;
Solver<CP> cp();
int n = 20;
range S = 1..n;
var<CP>{int} C[i in S](cp,S);
solve<cp> {
    cp.post(alldifferent(all(r in S) C[r] + r),onDomains);
    cp.post(alldifferent(all(r in S) C[r] - r),onDomains);
    cp.post(alldifferent(C),onDomains);
}
using
forall (i in S)
    tryall<cp> (v in S)
        cp.post(C[i] == v);

```

Statement 13.3: Improved CP Model for the Queens Problem

Conclusion When a CP model is built with binary variables, one should always check if it is possible to formulate it in a different way. This is particularly the case, when binary variables are used to reflect placement of items into particular locations: for example, queens on a board, item into bins, etc. The decision variables we branch on should ensure that every variable of the problem is assigned. Furthermore, each branching decision should have the strongest possible impact on the propagation.

The following section illustrates that, in addition to the choice of decision variables, the order, in which variables and values are considered, can also have an impact on the size of the search tree and, consequently, on the time to find a solution.

13.3 Variable and Value Heuristic

A non-deterministic search generally follows the next two steps, at each node of the search tree:

1. choose the next variable to instantiate
2. try the values in the variable's domain, to create alternatives

The choice of variable is called the *variable heuristic* and the value ordering is called the *value heuristic*.

In the previous section, variables were considered in a static order along a path of the search tree and their values were also tried in a static order. However, a dynamic selection of the next variable to instantiate, along with trying its values in a dynamical order, can significantly impact the size of the search tree. In what follows, we give some principles that can guide the modeler in implementing an efficient search scheme.

13.3.1 Variable Heuristic

A principle for choosing the next variable to instantiate is the *first-fail* principle:

Since all unbound variables must be assigned, choose to instantiate first the unbound variable with the largest chance of creating the smallest possible sub-tree, if the current partial solution cannot be extended into a full solution.

Indeed, we prefer to backtrack as soon as possible, if the current node is a dead end. In practice, this principle is often implemented by preferring the variable with the smallest domain size. This is exactly the strategy implemented by the default search `labelFF(X)`. A possible implementation of a first-fail heuristic to assign an array of variables `X` is the following:

```
function void labelFirstFail(var<CP>{int}[] X) {
  Solver<CP> cp = X[X.getLow()].getSolver();
  forall (i in X.getRange()) by (X[i].getSize())
    tryall<cp> (v in X[i].getMin()..X[i].getMax() : X[i].memberOf(v))
      cp.post(X[i] == v);
}
```


13.3.2 Value Heuristic

Ordering the values, once a variable has been selected, is more problem dependent. The general principle to follow is often the opposite of the first-fail principle used for variable selection. It is called the *first-success* principle for the value heuristic:

If the partial solution can be extended to a solution, we prefer to discover it as soon as possible. Thus, choose first the value with the most chances of leading into a solution.

When it is not clear which heuristic can rapidly lead to a solution, the modeler can use, among others, one of the following heuristics:

→ Try the values in increasing order:

```
tryall<cp> (v in X[i].getMin()..X[i].getMax() : X[i].memberOf(v))
  cp.post(X[i] == v);
```

→ Try the values in decreasing order:

```
tryall<cp> (v in X[i].getMin()..X[i].getMax() : X[i].memberOf(v)) by (-v)
  cp.post(X[i] == v);
```

→ Try first the values closer to the middle of the domain:

```
int mid = (X[i].getMin() + X[i].getMax()) / 2;
tryall<cp> (v in X[i].getMin()..X[i].getMax() : X[i].memberOf(v)) by (abs(v-mid))
  cp.post(X[i] == v);
```

→ Try the values in a randomized order:

```
range dom = X[i].getMin()..X[i].getMax();
RandomPermutation perm(dom);
int rand[dom] = perm.get();
tryall<cp> (v in dom : X[i].memberOf(v)) by (rand[v])
  cp.post(X[i] == v);
```

13.3.3 Domain Splitting

Trying all values of the domain as alternatives is called a *labeling*. Assigning a variable to a value may be a too radical decision for some problems. Domain splitting is a popular branching strategy, that consists of creating two alternatives by splitting the domain in two parts at the middle value. The following function is a possible implementation of a splitting strategy:

```
function void splitFirstFail(var<CP>{int}[] X) {
    Solver<CP> cp = X[X.getLow()].getSolver();
    while (!bound(X)) {
        selectMin (i in X.getRange() : !X[i].bound()) (X[i].getSize()) {
            int mid = (X[i].getMin() + X[i].getMax()) / 2;
            try<cp> cp.post(X[i] <= mid); | cp.post(X[i] > mid);
        }
    }
}
```

13.3.4 Design of a Branching Heuristic for the Queens Problem

We visit once again the queens model of Statement 13.3, in order to illustrate the first-fail and first-success principles for the variable and value heuristics. An interesting feature of the Queens Problem is that the variables and the values have a similar role. In fact, placing one queen in each row, and placing one queen in each column are dual viewpoints of the problem. The row variables R , introduced in Statement 13.4, represent, for each column, the row where its queen is placed.

Channeling between the two viewpoints is done with the following element constraints:

```
forall (c in S)
  cp.post(C[R[c]] == c);
```

As in Statement 13.3, the heuristic is trying to assign the column variables, but it now implements a first-fail variable and first-success value heuristic on top of that:

```
forall (r in S) by (C[r].getSize(),abs(r-n/2))
  tryall<cp> (c in S : C[r].memberOf(c)) by (R[c].getSize(),abs(r-n/2))
    cp.post(C[r] == c);
```

In more detail, the selection/assignment procedure is the following:

- The row that has the fewer possibilities is selected first. Ties are broken in favor of the central rows, the idea being that placing a queen in the middle rather than on a side has a higher impact on propagation.
- Once a row is select, we first try the available columns with the fewest number of possibilities. The number of possibilities for a column is precisely given by the dual viewpoint variables R . This is a first-success value heuristic, since we maximize the chances of success in the branch. Ties are also broken in favor of columns in the center of the board.

```

var<CP>{int} C[i in S](cp,S);
var<CP>{int} R[i in S](cp,S);
solve<cp> {
  forall (c in S)
    cp.post(C[R[c]] == c);
    cp.post(alldifferent(all(r in S) C[r] + r),onDomains);
    cp.post(alldifferent(all(r in S) C[r] - r),onDomains);
    cp.post(alldifferent(C),onDomains);
}
using {
  forall (r in S) by (C[r].getSize(),abs(r-n/2))
    tryall<cp> (c in S : C[r].memberOf(c)) by (R[c].getSize(),abs(r-n/2))
      cp.post(C[r] == c);
}

```

Statement 13.4: CP Model for the Queens Problem with First-Fail Search
(queens-cp-firstfaildual.co)

13.4 Designing Heuristics for Optimization Problems

Optimization problems using the **minimize** or the **optimize** blocks of COMET are solved with CP using a branch-and-bound search. Whenever a solution is encountered during the search, a constraint stating that the next solution must be better is dynamically added. The process of optimization in CP can be seen as solving a sequence of satisfaction problems with a constraint on the objective which becomes tighter and tighter, until optimality is proven.

For a large class of optimization problems, the difficulty is not to satisfy the constraints but to optimize the objective. For these problems, it is very important for the heuristic to drive the search quickly towards good solutions, so that the objective becomes tight early on. The tighter the objective, the stronger the pruning of the search tree, which is essential for proving optimality in reasonable time. This section illustrates two general techniques for designing a good search for such optimization problems:

- transforming a greedy algorithm into a non-deterministic search
- using a good initial solution to improve it with a non-deterministic search.

Good initial solutions don't have to be found using CP. One could use, for example, a greedy or a local search algorithm.

13.4.1 From a Greedy to a Good Non-Deterministic Search

We use the quadratic assignment problem to illustrate that designing a heuristic for an optimization problem is very similar to the implementation of a greedy algorithm for the same problem. The quadratic assignment problem (QAP) consists in finding an assignment of n facilities to n locations. The input consists of a distance function, defined for every pair of locations, and a weight function, defined for every pair of facilities. The weight can correspond, for example, to the amount of supplies transported between the two facilities. The problem is then to assign each facility to a different location, so that the sum of weighted distances between facilities is minimized. More formally, if W is the weight matrix and D the distance matrix, the objective is to find a permutation vector p (the location where each facility is assigned), minimizing the following sum:

$$\sum_{\substack{i \in \{1 \dots n\} \\ j \in \{1 \dots n\}}} W_{i,j} \cdot D_{p_i,p_j}$$

A greedy algorithm for this problem, it to proceed in n iterations: in each iteration, a facility is placed to a free location, trying to minimize the largest term of the objective function:

- choose the unassigned facility i with the largest weight with respect to some other facility i
- place facility i to a location that minimizes its potential distance from j

An implementation of this greedy algorithm is given in Statement [13.5](#).

```

int n;
range N = 1..n;
int W[N,N];
int D[N,N];
set{int} freei = filter(i in N) (true); // free slots
int perm[N] = -1;
forall (f in N)
    selectMax (i in N : perm[i] == -1, j in N) (W[i,j]) {
        set{int} freej = perm[j] != -1 ? {perm[j]} : freei;
        selectMin (n in freei) (min(l in freej : l != n) D[n,l]) {
            perm[i] = n;
            freei.delete(n);
        }
    }
cout << "objective:" << sum(i in N, j in N) W[i,j] * D[perm[i],perm[j]] << endl;

```

Statement 13.5: Greedy Algorithm for the Quadratic Assignment Problem

We transform this greedy algorithm into a non-deterministic search for the QAP in Statement 13.6. The left most branch of the non-deterministic search corresponds exactly to the greedy algorithm. One can see that the main difference between the greedy and the non-deterministic search is when assigning facility *i*. In the greedy search, it is assigned to *n* and no backtracking occurs after that:

```
selectMin (n in freei) (min(l in freej : l != n) D[n,l]) {
  perm[i] = n;
  freei.delete(n);
}
```

In the non-deterministic search, all possible values are tried with the **tryall** instruction starting with the values that would be chosen by the greedy algorithm:

```
tryall<cp> (n in N : p[i].memberOf(n))
          by (min(l in N : p[j].memberOf(l) && l != n) D[n,l])
cp.post(p[i] == n);
```

The **by** instruction specifies that the possible values in the **tryall** are tried in increasing order, according to the evaluation

$$\min(l \text{ in } N : p[j].memberOf(l) \ \&\& \ l \neq n) \ D[n,l]$$

which is the smallest potential distance between facilities *i* and *j*.

```

import cotfd;
Solver<CP> cp();
var<CP>{int} p[N](cp,N); // location of each facility
minimize<cp>
    2 * sum(i in N, j in N : j > i) W[i,j] * D[p[i],p[j]]
subject to
    cp.post(alldifferent(p));
using
    while (!bound(p))
        selectMax (i in N : !p[i].bound(), j in N) (W[i,j])
        tryall<cp> (n in N : p[i].memberOf(n))
            by (min(l in N : p[j].memberOf(l) && l != n) D[n,l])
        cp.post(p[i] == n);

```

Statement 13.6: CP Model for QAP: Search Adapted from Greedy Algorithm of Statement [13.5](#)

13.4.2 From a Good Initial Solution to a Non-Deterministic Search

Constraint Based Local Search or other methods can produce good solutions fast. These solutions can then be used to guide a non-deterministic search in labeling a vector of variables \mathbf{x} , given an initial solution represented by `init`.

The idea is quite simple:

- Choose a variable according to the first-fail heuristic, i.e., choose first the variable with the smallest domain
- If the value v of the chosen variable in the initial solution, is present in the variable's domain, the leftmost alternative assigns that value v to the variable and states that the variable take a different value in the other alternatives
- Otherwise, all values present in the domains are tried in increasing order, since it is not clear which one is worth trying first

Statement 13.7 defines a function `labelFirstFailFromInit` that implements this idea.

Combined with this heuristic, it often a good idea to change the default search controller to a bounded discrepancy search controller. Indeed, it is natural to expect that better solutions should not be too far from the initial solution. For this reason, we want to explore first the parts of the search tree with only one different entry compared to the initial solution, then parts of the tree with two different entries, and so on, until we reach a complete exploration. This is specified in Statement 13.7 right before the `minimize` block, where the default search controller is changed to `BDSCController`:

```
cp.setSearchController(BDSCController(cp));
```

The code included in the statement solves a small instance of twelve locations with an initial solution [7,5,12,2,1,3,9,11,8,6,10,4] using the function `labelFirstFailFromInit` for the labeling. Of course, the idea can be generalized to different problems and instances.

```

function void labelFirstFailFromInit(var<CP>{int}[] x,int[] init) {
    Solver<CP> cp = x[x.getRange().getLow()].getSolver();
    while (!bound(x))
        selectMin (i in x.getRange() : !x[i].bound()) (x[i].getSize())
            if (x[i].memberOf(init[i]))
                try<cp> cp.post(x[i] == init[i]); | cp.post(x[i] != init[i]);
            else
                tryall<cp> (v in x[i].getMin()..x[i].getMax() : x[i].memberOf(v))
                    cp.post(x[i] == v);
}

import cotfd;
Solver<CP> cp();
range N = 1..12;
var<CP>{int} p[N](cp,N); // location of facility
int pInit[N] = [7,5,12,2,1,3,9,11,8,6,10,4]; // initial solution
cp.setSearchController(BDSController(cp)); // set a BDS search controller
minimize<cp>
    2 * sum(i in N, j in N : j > i) W[i,j] * D[p[i],p[j]]
subject to
    cp.post(alldifferent(p));
using
    labelFirstFailFromInit(p,pInit);

```

Statement 13.7: CP Model for QAP with Labeling Based on Initial Solution (qap-cp-init.co)

13.5 Dynamic Symmetry Breaking During Search

Symmetries are not desirable in Constraint Programming models, because they result into exploring larger search trees, only to discover equivalent solutions. Symmetries can sometimes be avoided by posting constraints to remove equivalent solutions. Unfortunately, this easy technique of adding symmetry-breaking constraints can have a negative interaction with the heuristic: solutions that were discovered early on during the search may disappear, when including the new constraints. This means that it may take a longer time to discover the first solution with the symmetry-breaking constraints rather than without.

For some problems, symmetries can be avoided during the search, if we avoid creating equivalent alternatives. This *dynamic symmetry breaking during search* (DSBDS) technique has the advantage that it does not interact with the heuristic. We illustrate DSBDS on a movie scene scheduling problem, that can be summarized as follows:

- A number of scenes must be scheduled for the next days of the week
- For each scene, we are given a set of playing actors
- Every actor is paid a fixed fee for each day he is playing
- At most five scenes can be scheduled per day
- The objective is to minimize the total cost

The complete CP model is given at Statement 13.8. The data of the problem includes the number of scenes and days, the actors and their fees. For each scene, we define a set **appears**, which stores the actors that are present in the scene. We also compute, for each actor, a set **which** of the scenes, where the actor appears. The decision variables correspond to the day each scene is scheduled for:

```
var<CP>{int} shoot[Scenes](cp,Days);
```

```

import cotfd;
Solver<CP> cp();
int      maxScene   = 19;
range    Scenes     = 0..maxScene-1;
range    Days       = 0..4;
enum     Actor      = {Patt, Casta, Scolaro, Murphy, Brown, Hacket,
                      Anderson, McDougal, Mercer, Spring, Thompson};
int      fee[Actor] = [26481, 25043, 30310, 4085, 7562, 9381,
                      8770, 5788, 7423, 3303, 9593];

set{Actor} appears[Scenes];
appears[0] = {Hacket};
appears[1] = {Patt,Hacket,Brown,Murphy};
appears[2] = {McDougal,Scolaro,Mercer,Brown};
...
appears[17] = {Scolaro,McDougal,Hacket,Thompson};
appears[18] = {Casta};
set{int}   which[a in Actor] = filter(i in Scenes) member(a,appears[i]);

var<CP>{int} shoot[Scenes] (cp,Days);
minimize<cp>
    sum(a in Actor) sum(d in Days) fee[a] * (or(s in which[a]) shoot[s] == d)
subject to
    cp.post(atmost(all(k in Days) 5,shoot),onDomains);
using
    while (!bound(shoot)) {
        int mday = max(-1,maxBound(shoot));
        selectMin (s in shoot.getRange() : !shoot[s].bound())
            (shoot[s].getSize(),-sum(a in appears[s]) fee[a])
        tryall<cp> (d in Days : d <= mday + 1 && shoot[s].memberOf(d))
            cp.label(shoot[s],d);
        onFailure
            cp.diff(shoot[s],d);
    }

```

Statement 13.8: Dynamic Symmetry Breaking During Search on the Scene Scheduling Problem
(scene-cp.co)

The objective to minimize is:

```
sum(a in Actor) sum(d in Days) fee[a] * (or(s in which[a]) shoot[s] == d)
```

where `(or(s in which[a]) shoot[s] == d)` is reified to 1, if actor `a` plays on day `d`, and 0 otherwise. The only constraint posted is that at most 5 scenes are scheduled per day:

```
cp.post(atmost(all(k in Days) 5,shoot),onDomains);
```

The search is a first-fail labeling search on the `shoot` variables, where ties on domain size are broken by assigning first the scenes with many expensive actors. The most interesting part of this search is that not all values are tried for a variable. Assume, for instance, that currently some of the days have no scene. Then it doesn't make sense to try all the free days, since this would lead to equivalent sub-problems. Instead, at most one free day is tried at every node, which dynamically breaks symmetries while remaining a complete search.

```
while (!bound(shoot)) {
  int mday = max(-1,maxBound(shoot));
  selectMin (s in shoot.getRange() : !shoot[s].bound())
    (shoot[s].getSize(),-sum(a in appears[s]) fee[a])
  tryall<cp> (d in Days : d <= mday + 1 && shoot[s].memberOf(d))
    cp.label(shoot[s],d);
  onFailure
    cp.diff(shoot[s],d);
}
```

13.6 Restarts

Restarts can help solving satisfaction problems with a huge search tree that cannot be fully explored in a reasonable time. For such problems, early decisions, near the root of the tree are of capital importance. The search tree is so big that they have little chance of being reconsidered later, if they were wrong: the search is then stuck in a bad region of the search tree that contains no solution. Furthermore, the earliest decisions are often arbitrary and uninformed. To avoid this phenomenon, it may be useful to restart the search from scratch after some time. This allows to explore other parts of the search tree and reconsider the early choices at each restart. COMET offers the capability to implement a restart strategy without modifying the search or the model.

The restart technique is illustrated on the magic square example. A magic square of order n is an $n \times n$ matrix, that contains all numbers of the range $[1..n^2]$, so that the sum of numbers along each row, column and main diagonal has the same value, equal to $n \cdot (n^2 + 1)/2$. An example of magic square of order 8 is the following:

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 6 | 22 | 49 | 23 | 46 | 45 | 62 | 7 |
| 11 | 29 | 52 | 53 | 60 | 44 | 8 | 3 |
| 41 | 16 | 30 | 42 | 17 | 25 | 50 | 39 |
| 43 | 58 | 27 | 31 | 28 | 38 | 14 | 21 |
| 51 | 13 | 56 | 32 | 34 | 54 | 5 | 15 |
| 26 | 19 | 33 | 20 | 24 | 35 | 48 | 55 |
| 18 | 63 | 12 | 57 | 4 | 9 | 36 | 61 |
| 64 | 40 | 1 | 2 | 47 | 10 | 37 | 59 |

Statement 13.9 solves the magic square problem using a first-fail splitting strategy. The decision variables are

```
var<CP>{int} s[R,R] (cp,D)
```

representing the number in each entry of the matrix. The constraints posted are simply the arithmetic constraints that characterize the problem, with the addition of a constraint that breaks symmetries by forcing numbers along one main diagonal to be increasing:

```
forall (i in 1..n-1)
  cp.post(s[i,i] < s[i+1,i+1]);
```

```

import cotfd;
Solver<CP> cp();
int    n = 8;
range R = 1..n;
range D = 1..n^2;
int    T = n * (n^2 + 1) / 2;

var<CP>{int} s[R,R] (cp,D);
solve<cp> {
  forall (i in R) {
    cp.post(sum(j in R) s[i,j] == T);
    cp.post(sum(j in R) s[j,i] == T);
  }
  cp.post(sum(i in R) s[i,i] == T);
  cp.post(sum(i in R) s[i,n-i+1] == T);
  cp.post(alldifferent(all(i in R,j in R) s[i,j]));
  forall (i in 1..n-1) //symmetry breaking
    cp.post(s[i,i] < s[i+1,i+1]);
}
using {
  var<CP>{int}[] vars = all(i in R,j in R) s[i,j];
  range V = vars.getRange();
  while (!bound(vars))
    selectMin (i in V : !vars[i].bound()) (vars[i].getSize()) {
      int mid = (vars[i].getMin() + vars[i].getMax()) / 2;
      try<cp> cp.post(vars[i] <= mid); | cp.post(vars[i] > mid);
    }
}

```

Statement 13.9: Magic Square Problem (magicsquare-cp.co)

The model of Statement [13.9](#) can easily find a magic square of order 9, but has difficulty finding one of order 10. We show that it is much easier to solve the problem for order 10, by applying a restart strategy combined with a randomized search. We only need to make the following modifications to the model:

- restart the search upon reaching a given failure limit
- at every restart, increase the failure limit by a constant
- randomize the heuristic, in order to explore a different search tree at each restart

We now get into more details. We choose to set up a restart when reaching the limit of 1000 failures without finding a solution. This is done with the line:

```
cp.restartOnFailureLimit(1000);
```

The user can add an (optional) **onRestart** block to better control the restart process. If such a block is implemented, the code inside the block is executed prior to each restart. In this example, we use this block to increase the failure limit by 100 at each restart:

```
cp.setRestartFailureLimit(cp.getRestartFailureLimit() + 100);
```

A successful restart strategy should use a randomized heuristic, otherwise the tree explored is the same at every restart. At this end, we make two modifications to the model of Statement [13.9](#): we replace `selectMin` with `selectMin[3]`, so that a variable is randomly chosen among those with the three smallest domain sizes; and we randomize the splitting order:

```
select (j in 0..1)
  if (j==1)
    try<cp> cp.post(vars[i] <= mid); | cp.post(vars[i] > mid);
  else
    try<cp> cp.post(vars[i] >= mid); | cp.post(vars[i] < mid);
```

These simple modifications make it relatively easy to find an order 10 magic square within a few restarts. The complete modified version of the model is shown in Statement [13.10](#)

```

import cotfd;
Solver<CP> cp();
int    n = 10;
range R = 1..n;
range D = 1..n^2;
int    T = n * (n^2 + 1) / 2;

var<CP>{int} s[R,R] (cp,D);
solve<cp> {
  forall (i in R) {
    cp.post(sum(j in R) s[i,j] == T);
    cp.post(sum(j in R) s[j,i] == T);
  }
  cp.post(sum(i in R) s[i,i] == T);
  cp.post(sum(i in R) s[i,n-i+1] == T);
  cp.post(alldifferent(all(i in R,j in R) s[i,j]));
  forall (i in 1..n-1) //symmetry breaking
    cp.post(s[i,i] < s[i+1,i+1]);
}
using {
  var<CP>{int}[] vars = all(i in R,j in R) s[i,j];
  range V = vars.getRange();
  while (!bound(vars))
    selectMin[3] (i in V : !vars[i].bound()) (vars[i].getSize()) {
      int mid = (vars[i].getMin() + vars[i].getMax()) / 2;
      select (j in 0..1)
        if (j==1)
          try<cp> cp.post(vars[i] <= mid); | cp.post(vars[i] > mid);
        else
          try<cp> cp.post(vars[i] >= mid); | cp.post(vars[i] < mid);
    }
}

```

Statement 13.10: Magic Square Problem with Restarts (magicsquare-cp-restart.co)

13.7 Large Neighborhood Search (LNS)

Large Neighborhood Search (LNS) is a powerful technique, that hybridizes Constraint Programming and Local Search to solve optimization problems. In each iteration, a neighborhood is explored with CP trying to improve the current best solution. The LNS process is summarized as follows:

1. Find a feasible solution using CP
2. Relax a part of the current solution:
 - restore the domain of some variables
 - fix the remaining variables to their current value
3. Re-optimize the restricted problem using CP with a limit on the number of failures
4. If the current best solution cannot be improved or a failure limit occurs, go back to step 2
5. Repeat steps 2–4, until a stopping criterion is met

The reason for setting a failure limit is to avoid exploring a neighborhood for too long, allowing the search to explore a variety of neighborhoods. Here are some of the benefits of using LNS:

- With very few changes, it allows to adapt a CP model into a hybridized procedure that scales very well with the problem size
- In each iteration, LNS explores a neighborhood large enough to avoid the need for a meta-heuristic, such as tabu search
- It requires minimal parameter tuning

13.7.1 Choosing Relaxation Procedure and Failure Limits

One advantage of LNS is the ease of implementation, since very few parameters need to be tuned. The two main design decisions in transforming a CP model into an LNS procedure are:

- Choosing the right relaxation procedure. A basic relaxation procedure, such as relaxing a randomly chosen subset of the variables, usually works well in practice.
- Choosing a limit on the number of failures or the time allowed per iteration

These design choices are closely related: if the relaxed part of the problem is too large and the number of allowed failures per iteration too small, there is little hope to improve the current best solution. Indeed, in this case, the neighborhood to explore is almost as large as solving the initial problem.

A good combination of relaxation procedure and failure limit should allow to almost completely explore the neighborhood: as a rule of thumb, a full neighborhood exploration should occur once every two iterations. Following this idea, we only need to tune the time we are willing to spend in every LNS iteration. The skeleton a CP optimization model in COMET is given on the left fragment, while its LNS version is shown on the right:

```
Solver<CP> cp();

//variable declaration
minimize<cp>
    // some variable or expression
subject to {
    // post the constraints
}
using {
    // non-deterministic search
}
```

```
Solver<CP> cp();
cp.lnsOnFailure(maxNumberOfFailures);
// variable declaration
minimize<cp>
    // some variable or expression
subject to {
    // post the constraints
}
using {
    // non-deterministic search
}
onRestart {
    // relaxation procedure
}
```

Observe that no change to the CP model is necessary to transform it into an LNS: we only need to add a call to the method `lnsOnFailure` to put in place an LNS with a given number of failure limit per iteration. The relaxation procedure must be implemented in the **onRestart** block. If

nothing is implemented in the restart block, all variables are restored to their initial domain upon each restart. To restrict the size of the relaxed problem, we need to fix some variables to their values in the current best solution.

Remark Due to the constraint solver’s architecture, each assignment of value to a variable is followed by a propagation step. This could lead to a very time-consuming **onRestart** block, since typically we fix many variables to their current values. In cases like that, it is often a good idea to include the code that fixes the variables inside a **with atomic()** block. This will postpone propagation until all variables are fixed.

13.7.2 Starting from an Initial Solution

The initial solution used for starting the LNS is produced by default using CP, but we could conceivably produce it with a different technique, for example using CBLS. The **startWithRestart** instruction is particularly useful in cases like that. It states that the **onRestart** block should execute, before the **using** block is entered for the first time.

When the **onRestart** block runs for the first time, all variables are fixed to the values they take in the given initial solution. In subsequent iterations of the **onRestart** block, a relaxation is created as in the standard LNS. An implementation on the QAP, that starts the LNS with an initial solution `initSol[]`, is illustrated on Statement 13.11. A `Boolean` object variable `init` is used to determine whether it is the first time the **onRestart** block runs.

```

import cotfd;
Solver<CP> cp();
int n;
range N = 1..n;
int W[N,N];
int D[N,N];
Boolean init(true);
UniformDistribution dist(1..100);
int initSol[N] = ...; //an initial solution

var<CP>{int} p[N](cp,N);
cp.lnsOnFailure(50);
cp.startWithRestart();

minimize<cp>
  2 * sum(i in N, j in N : j > i) W[i,j] * D[p[i],p[j]]
subject to
  cp.post(alldifferent(p));
using {
  labelFF(p);
  cout << "fails:" << cp.getNFail() << endl;
}
onRestart {
  with atomic(cp) {
    if (init) {
      forall (i in N)
        cp.post(p[i] == initSol[i]);
      init := false;
    }
    else {
      Solution s = cp.getSolution();
      forall (i in N : dist.get() <= 90)
        cp.post(p[i] == p[i].getSnapshot(s));
    }
  }
}

```

Statement 13.11: Starting LNS with an Initial Solution: Illustration on the QAP
(qap-cp-lns-init.co)

13.7.3 Combining LNS with Restarts

For some optimization problems, it is beneficial to combine LNS with restarts, to increase the diversification of the search. With the standard LNS, after every restart, the solver posts an implicit constraint that the objective function must improve the objective of the current best solution. This may be quite restrictive in some cases, since it could prevent the search from reaching particular regions of the search space with good solution. COMET offers a variant of LNS that addresses this issue. One simply has to use the following form of the `lnsOnFailure` method:

```
void lnsOnFailure(int nbFailures,int nbStable,int nbStarts)
```

This will perform an LNS iteration, whenever the number of failures reaches `nbFailures`. It will keep performing LNS iterations, until the objective has not improved for `nbStable` consecutive iterations. At that point, we perform a more radical restart: after the LNS relaxation procedure, the objective function is now relaxed, i.e., no implicit constraint is posted. This can lead to a potential degradation, resulting in better diversification. The search terminates after `nbStarts` such restarts. Note that, in the end, the best solution found during the whole search process is restored. Statement 13.12 gives a version of the QAP model that uses this form of LNS. The line

```
cp.lnsOnFailure(20,3,15);
```

in this example, runs LNS whenever the search encounters 20 failures. If LNS does not improve for 3 iterations, it performs a restart. The search terminates after 15 restarts.

```

import cotfd;
Solver<CP> cp();
int n;
range N = 1..n;
int W[N,N];
int D[N,N];
UniformDistribution dist(1..100);

var<CP>{int} p[N](cp,N);
cp.lnsOnFailure(20,3,15);

minimize<cp>
  2 * sum(i in N, j in N : j > i) W[i,j] * D[p[i],p[j]]
subject to
  cp.post(alldifferent(p));
using {
  labelFF(p);
  cout << p << endl;
  cout << "fails:" << cp.getNFail() << endl;
}
onRestart {
  Solution s = cp.getSolution();
  with atomic(cp)
    forall (i in N : dist.get() <= 90)
      cp.post(p[i] == p[i].getSnapshot(s));
}
Solution s = cp.getSolution();
cout << "Best Solution found:" << endl;
cout << all (i in N) p[i].getSnapshot(s) << endl;
cout << "Solution value: " <<
  2 * sum(i in N, j in N : j > i)
    W[i,j] * D[p[i].getSnapshot(s),p[j].getSnapshot(s)] << endl;

```

Statement 13.12: LNS with Restarts for the QAP (qap-cp-`lns2.co`)

13.7.4 Adaptive LNS

As mentioned earlier, a LNS restart can occur for two reasons:

- a. because the failure/time limit is reached
- b. because the search is exhausted

A well-chosen combination of relaxation procedure and failure limit should, ideally, lead to roughly the same number of restarts caused by each of these reasons. Therefore, it is often a good idea to dynamically adapt the LNS failure or time limit, or the parameters of the relaxation procedure, based on the cause of the last restart. The method `isLastLNSRestartCompleted()` of the solver allows to test the origin of the LNS restart. It returns `true` in case of a complete search (case b. above), and `false` otherwise.

Statement [13.13](#) dynamically increases the failure limit by 10%, whenever the restart is caused by a failure limit, and decreases it by 10%, whenever it is caused by a complete search. It makes use of the solver methods `getLNSFailureLimit`, that returns the current failure limit, and `setLNSFailureLimit` that updates the failure limit. The corresponding methods for time-based LNS are `getLNSTimeLimit` and `setLNSTimeLimit` (time is measured in seconds).

```

import cotfd;
Solver<CP> cp();
int n;
range N = 1..n;
int W[N,N];
int D[N,N];
UniformDistribution dist(1..100);

var<CP>{int} p[N](cp,N);
cp.lnsOnFailure(20);

minimize<cp>
  2 * sum(i in N, j in N : j > i) W[i,j] * D[p[i],p[j]]
subject to
  cp.post(alldifferent(p));
using {
  labelFF(p);
  cout << "fails:" << cp.getNFail() << endl;
}
onRestart {
  Solution s = cp.getSolution();
  with atomic(cp)
    forall (i in N : dist.get() <= 90)
      cp.post(p[i] == p[i].getSnapshot(s));
  if (!cp.isLastLNSRestartCompleted()) // restart due to failure limit
    cp.setLNSFailureLimit(cp.getLNSFailureLimit() * 110 / 100);
  else // restart due to complete search
    cp.setLNSFailureLimit(max(20, cp.getLNSFailureLimit() * 90 / 100));
}

```

Statement 13.13: Adaptive LNS for the QAP (qap-cp-adaptivelns.co)

13.7.5 Differences between Restarts and LNS

Restarts and LNS might seem similar techniques, but they have some fundamental differences:

- LNS is well-suited for optimization problems, where an initial feasible solution is generally not too difficult to find. Restarts are more useful for satisfaction problems
- The restarting process stops, as soon as a solution is found; on the other hand, LNS keeps running until the solver is asked to exit
- In the restart strategy, a restart occurs, only if the time/number of failures reaches the specified limit. If a solution is found or if the search space is fully explored, no more restart take place. In LNS, instead, a restart occurs if the limit is reached, but also if the search is complete.
- The restart strategy, as implemented in the magic square example, is a complete method, since the number of failures is increased at each iteration. LNS is a local search technique, not really intended to prove optimality, but rather to rapidly find a solution. Even if we tried to tweak LNS by adjusting the failure limit before every LNS restart, we have to keep in mind that LNS may restart even when the tree search is complete.
- Both techniques share the idea of exploring different parts of the search tree to avoid being trapped for too long in bad regions, but diversification is achieved in a different way for the restart strategy: In the restart approach, we randomize the heuristic, whereas in LNS it is the relaxation that is randomized and not necessarily the heuristic.

13.8 Speeding Up Branch and Bound with CBLS*

LNS can be seen as an hybridization technique between Constraint Programming and Local Search (LS), where CP is used to explore the neighborhood of the current solution. In this section, we use a different approach for optimization problems, in which CP is the master solver, and CBLS is used as the slave technique. Users not very familiar with Local Search are encouraged to consult the CBLS part of this tutorial, for more details on using local search in COMET.

Let us first review the Branch and Bound process for CP optimization. During the search tree exploration, every time a solution is found, a constraint is dynamically added to the store, enforcing that the next solution found is better. Hence, the last solution found is proven to be optimal, by construction. The time required to find an optimal solution and prove it optimal is strongly influenced by the time necessary to reach good solutions, that allow to prune the search tree.

In this section's scheme, we are using CBLS to improve a solution found during the Branch and Bound Search in an opportunistic way. The hope is to improve the objective faster, and thus reduce the size of the search tree. Once again, we illustrate this hybridization technique on the Quadratic Assignment Problem. The complete solution is given on Statement 13.14, which we now explain in more detail. Like before, the data includes the range of locations N and the weight and distance matrices, W and D :

```
range N;
int W[N,N]; //weights
int D[N,N]; //distances
```

A CBLS model is declared next:

```
import cotls;
Solver<LS> ls();
RandomPermutation perm(N);
var{int} x[N](ls,N) := perm.get(); // initial random permutation
FunctionExpr<LS> O(sum(i in N,j in N) W[i,j]*D[x[i],x[j]]); // objective function
ls.close();
Solution solution = new Solution(ls); //store the best solution
```

The decision variables of the CBLS model are in the permutation vector **x** that gives the location where each facility is placed. Locations are randomly initialized with a **RandomPermutation**. A differentiable expression function **O** is then built to express the objective function. The best CBLS solution is stored in a **Solution** object that can be restored at any time. After that, we introduce the following CP model (we only show what follows the **using** block):

```
using {
  ...
  // improve current solution with CBLS
  forall (i in N) x[i] := p[i].getValue();
  while (true) {
    int old = O.evaluation();
    selectMin (i in N, j in i..n) (O.getSwapDelta(x[i],x[j]))
    x[i] := x[j];
    if (O.evaluation() >= old) {
      solution.refresh(ls); // update best CBLS solution
      cp.setPrimalBound(MinimizeIntValue(O.evaluation())); // set new B&B bound
      break; // no more greedy improvement possible with CBLS
    }
  }
}
solution.restore();
```

The objective, the constraints and the search are the same as in Statement 13.6, but we now add a block of local search code, that executes every time a solution is found during the branch-and-bound search. After the **while(!bound(p))** loop, a new improving solution is available and all CP variables in **p** are bound. So their values can be copied into the incremental (local search) variables **x**.

Then, we apply a greedy best-improvement search that keeps swapping the locations of pairs of facilities, until no further improvement is possible. At that point, we refresh the **Solution** object, to store the state of the incremental variables, and use the **setPrimalBound** method to add a new bound for the branch-and-bound according to the value discovered with CBLS. Finally, once optimality is proven, we restore the CBLS solution.

Note that, since we use a non-degrading CBLS search, it is not strictly necessary to store the state of incremental variables into a **Solution**: the incremental variables always contain the solution corresponding to the best branch-and-bound bound. But since we might want to use a different heuristic, such as tabu search, that could degrade the current objective in some of its iterations, it makes sense to store the best solution to be able to restore it at the end of the branch-and-bound.

```

range N;
int W[N,N]; // weights
int D[N,N]; // distances

// CBLS Model
import cotls;
Solver<LS> ls();
RandomPermutation perm(N);
var{int} x[N](ls,N) := perm.get(); // initial random permutation
FunctionExpr<LS> O(sum(i in N, j in N) W[i,j] * D[x[i],x[j]]);
ls.close();
Solution solution = new Solution(ls); // store the best solution
// CP Model
import cotfd;
Solver<CP> cp();
var<CP>{int} p[N](cp,N); // location of each facility
minimize<cp>
    sum(i in N, j in N) W[i,j] * D[p[i],p[j]]
subject to
    cp.post(alldifferent(p));
using {
    while (!bound(p))
        selectMax (i in N : !p[i].bound(), j in N) (W[i,j])
        tryall<cp> (n in N : p[i].memberOf(n))
            by(min(l in N : p[j].memberOf(l) && l != n) D[n,l])
            cp.post(p[i] == n);
    // improve current solution with CBLS
    forall (i in N) x[i] := p[i].getValue();
    while (true) {
        int old = O.evaluation();
        selectMin (i in N, j in i..n) (O.getSwapDelta(x[i],x[j]))
        x[i] := x[j];
        if (O.evaluation() >= old) {
            solution.refresh(ls); // update best CBLS solution
            cp.setPrimalBound(MinimizeIntValue(O.evaluation())); // set new B&B bound
            break; // no more greedy improvement possible with CBLS
        }
    }
}
}
solution.restore();
cout << "permutation:" << x << " objective:" << O.evaluation() << endl;
cout << "#fails: " << cp.getNFail() << endl;

```

Statement 13.14: Hybridization of CP (Master) and CBLS (Slave) for QAP (qap-cp-cbils.co)

Chapter 14

Over-Constrained Problems

This chapter discusses how to deal with situations, in which the CP model complains that there is no solution, because the problem is over-constrained. Note that we do not discuss here how to debug a CP model: this can be done, for example, by trying the model on small instances that can be solved by hand, or on instances with a known solution.

14.1 Dropping-Then-Relaxing Constraints

In general, not all constraints of a CP model have the same importance and, in fact, some of them can be treated as preferences rather than hard constraints.

For instance, in a problem of scheduling exams in a university, students might express preferences of the form:

- *a student should not have two consecutive exams scheduled*
- *a student should not have more than two exams each week*

In any case, nobody is in a better position than the modeler himself, to determine which constraints can be treated as preferences. Once the preferences have been identified, a reasonable strategy to solve the problem is the following:

1. Start dropping constraints by decreasing importance, until the problem becomes feasible
2. Once the problem is feasible, introduce soft versions of the dropped constraints
3. Try to minimize the violation variables of the soft constraints, so that preferences are optimized

For a detailed account of soft constraints in COMET, please refer to Section [10.4.10](#). Section [14.2](#) illustrates this approach on an over-constrained time-tabling problem, in which we need to relax an **alldifferent** constraint. Then, section [14.3](#) focuses on an over-constrained personnel-scheduling problem, in which we decide to relax an **atleast** constraint.

14.2 An Over-Constrained Time-Tabling Problem

We consider the problem of allocating events to rooms and time slots. There are 16 events, 4 rooms and 4 time slots, and we need to assign each event to a different (room,slot) pair. Each of ten students has registered to four events. Since a student can attend at most one event in each time-slot, the events of a student must be scheduled in different time slots.

An instance of this problem and the complete model to solve it is given in Statement 14.1. The data consists of the ranges `Students`, `Rooms`, `Slots` and `Events`, and of the set of events attended by each student, given by the array of sets `eventsAttendedByStudent`. For instance, the first student attends events 1, 3, 4, and 6.

```
range Students = 1..10;
range Rooms   = 1..4;
range Slots    = 1..4;
range Events   = 1..16;

set<int> eventsAttendedByStudent[Students] = [{1,3,4,6},
                                             ...
                                             {2,3,9,10}];
```

The variables `events[r,s]` represent the event scheduled for room `r` and time-slot `s`. Variables `room[e]` and `slots[t]` give, respectively, the room and slot, in which event `e` is scheduled.

```
var<CP>{int} events[Rooms,Slots] (cp,Events);
var<CP>{int} rooms[Events] (cp,Rooms);
var<CP>{int} slots[Events] (cp,Slots);
```

The first constraint ensures that each event is placed exactly once:

```
cp.post(alldifferent(all(i in Rooms,j in Slots) events[i,j]), onDomains);
```

```

range    Students = 1..10;
range    Rooms    = 1..4;
range    Slots    = 1..4;
range    Events   = 1..16;

set{int} eventsAttendedByStudent[Students] = [{1,3,4,6},
                                                {1,5,6,13},
                                                {3,7,9,10},
                                                {5,8,4,15},
                                                {6,7,8,16},
                                                {8,9,10,12},
                                                {6,9,13,14},
                                                {12,13,15,16},
                                                {3,7,8,10},
                                                {2,3,9,10}];

import cotfd;
Solver<CP> cp();

var<CP>{int} events[Rooms,Slots](cp,Events);
var<CP>{int} rooms[Events](cp,Rooms);
var<CP>{int} slots[Events](cp,Slots);

solve<cp> {
  cp.post(alldifferent(all(i in Rooms, j in Slots) events[i,j]), onDomains);
  forall (e in Events)
    cp.post(events[rooms[e],slots[e]] == e);
  forall (s in Students)
    cp.post(alldifferent(all(e in eventsAttendedByStudent[s]) slots[e]), onDomains);

  //symmetry breaking constraints
  forall (i in 1..Rooms.getUp()-1, j in Slots)
    cp.post(events[i,j] < events[i+1,j]);
  forall (i in 1..Rooms.getUp()-1)
    cp.post(events[1,i] < events[1,i+1]);
}
using
  labelFF(events);

```

Statement 14.1: Time-Tabling with Hard Constraints (timetabling-cp-hard.co)

The next block links the `rooms`, `slots` and `events` variables through element constraints. This kind of constraints are called channeling constraints between different viewpoints:

```
forall (e in Events)
  cp.post(events[rooms[e],slots[e]] == e);
```

We then state the constraint that, for each student, his events are scheduled in different time-slots:

```
forall (s in Students)
  cp.post(alldifferent(all(e in eventsAttendedByStudent[s]) slots[e]), onDomains);
```

Finally, since rooms and slots are equivalent, we can break some symmetries with the constraints:

```
forall (i in 1..Rooms.getUp()-1, j in Slots)
  cp.post(events[i,j] < events[i+1,j]);
forall (i in 1..Rooms.getUp()-1)
  cp.post(events[1,i] < events[1,i+1]);
```

For the search, in the `using` block, we perform a default first-fail labeling on the vector `events` of decision variables. Unfortunately, the model of Statement 14.1 does not return a solution because the problem is over-constrained. The infeasibility is caused by the constraint that:

The four events of each student are scheduled at different time-slots

Without this constraint, any permutation of the events would be a solution to the problem. We choose to relax this constraint into:

*The four events of each student are **preferably** scheduled at different time-slots*

The `atLeastNValue` global constraint, which is the soft version of the `alldifferent` constraint, and counts the number of different values in a vector of variables, achieves exactly the required relaxation. Recall the form of the constraint from Section 10.4.10:

```
AtLeastNValue<CP> atLeastNValue(var<CP>{int}[] x,var<CP>{int} numberOfValue)
```

For any given student, the `numberOfValue` variable represents the number of different slots, into which his events are scheduled. We post the soft constraints for all students with:

```
forall (s in Students) {  
    var<CP>{int}[] slotsOfs = all(e in eventsAttendedByStudent[s]) slots[e];  
    cp.post(atLeastNValue(slotsOfs,nbEventsByStudent[s]), onDomains);  
}
```

Note that we use the `onDomains` level of consistency for this constraint (the default for this constraint is the weaker consistency level `onValue`). The objective function is the total number of events attended by the students in the whole schedule. The complete model with soft global constraints maximizing this objective is given in Statement [14.2](#).

```

import cotfd;
Solver<CP> cp();

// Read data

var<CP>{int} events[Rooms,Slots] (cp,Events);
var<CP>{int} rooms[Events] (cp,Rooms);
var<CP>{int} slots[Events] (cp,Slots);
var<CP>{int} nbEventsByStudent[Students] (cp,0..Slots.getUp());

maximize<cp>
    sum(s in Students) nbEventsByStudent[s]
subject to {
    cp.post(alldifferent(all(i in Rooms, j in Slots) events[i,j]), onDomains);
    forall (e in Events)
        cp.post(events[rooms[e],slots[e]] == e);
    forall (s in Students) {
        var<CP>{int}[] slotsOfs = all(e in eventsAttendedByStudent[s]) slots[e];
        cp.post(atLeastNValue(slotsOfs,nbEventsByStudent[s]), onDomains);
    }
    forall (i in 1..Rooms.getUp()-1, j in Slots)
        cp.post(events[i,j] < events[i+1,j]);
    forall (i in 1..Rooms.getUp()-1)
        cp.post(events[1,i] < events[1,i+1]);
}
using
    labelFF(events);

```

Statement 14.2: Time-Tabling with Soft Constraints (timetabling-cp-soft.co)

14.3 An Over-Constrained Personnel Scheduling Problem

The next problem is to schedule activities for 5 persons on 6 different time-slots. There is a set of 10 possible activities and each person chooses at least 6 different activities from that set, according to their skills and preferences. In order to diversify the work, all activities of a person must be different in the course of the schedule. In addition to this strong constraint, for every time-slot we are given some demand requirements on each activity (stating that at least a number of persons should perform the activity in the given slot). Ideally, the schedule of activities should cover these demands in each time-slot.

The complete model is given in Statement 14.3. The data of the problem consists of the ranges **Persons**, **Slots** and **Activities**, along with possible activities per person and demand requirements. For each person p , the set of possible activities is given by the array `possibleActivity[p]`. For each time-slot t , the demand for each activity is specified in the corresponding row t of the matrix `demand`:

```
range  Persons    = 1..5;
range  Slots      = 1..6;
range  Activities  = 1..10;
set{int} possibleActivity[Persons] = [{1,2,3,4,5,6,8},
                                     ...
                                     {1,4,5,7,8,10}];
int     demand[Slots,Activities] = [[1,0,2,1,0,0,0,0,1,0],
                                     ...
                                     [0,0,1,0,0,1,1,0,1,0]];
```

The only variables introduced for this problem represent the activity assigned to each person in each time-slot:

```
var<CP>{int} activities[p in Persons,t in Slots](cp,possibleActivity[p]);
```

```

import cotfd;
Solver<CP> cp();

range Persons = 1..5;
range Slots = 1..6;
range Activities = 1..10;
set<int> possibleActivity[Persons] = [{1,2,3,4,5,6,8},
                                     {1,3,4,7,9,10},
                                     {2,4,5,8,9,10},
                                     {1,3,5,6,7,8,9,10},
                                     {1,4,5,7,8,10}];

int demand[Slots,Activities] = [[1,0,2,1,0,0,0,0,1,0],
                                 [2,0,0,0,0,1,0,0,0,1],
                                 [1,0,0,1,0,0,1,0,0,2],
                                 [0,1,0,1,0,0,0,0,1,0],
                                 [1,0,1,0,0,1,0,1,0,1],
                                 [0,0,1,0,0,1,1,0,1,0]];

var<CP>{int} activities[p in Persons,t in Slots](cp,possibleActivity[p]);
solve<cp> {
  forall (p in Persons)
    cp.post(alldifferent(all(t in Slots) activities[p,t]), onDomains);
  forall (t in Slots) {
    int[] demand_t = all(a in Activities) demand[t,a];
    var<CP>{int}[] activities_t = all(p in Persons) activities[p,t];
    cp.post(atleast(demand_t,activities_t), onDomains);
  }
}
using
  labelFF(activities);

```

Statement 14.3: Personnel Scheduling with Hard Constraints ([personal-scheduling-cp-hard.co](#))

The first set of constraints enforces that all the activities assigned to a person are different along the schedule:

```
forall (p in Persons)
    cp.post(alldifferent(all(t in Slots) activities[p,t]), onDomains);
```

The second set of constraints enforces the minimum demand coverage, for each type of activity and each time-slot, using an `atleast` constraint:

```
forall (t in Slots) {
    int[] demand_t = all(a in Activities) demand[t,a];
    var<CP>{int}[] activities_t = all(p in Persons) activities[p,t];
    cp.post(atleast(demand_t,activities_t), onDomains);
}
```

Finally, the non-deterministic search used in the `using` block is a default first-fail labeling. Unfortunately, the model of Statement 14.3 terminates without finding any solution, which means that the problem is over-constrained. We choose to relax the `atleast` constraints that enforce the demand requirements per time-slot, by using their soft counter-part: the `softAtLeast` constraint.

The objective then is to minimize the sum of violations (shortage of demand) over all the periods, as expressed in Statement 14.4. Within a few seconds, this model is able to find an optimal solution, which features a total of 3 violations, with respect to the instance given in Statement 14.3.

```

import cotfd;
Solver<CP> cp();

// Read data

var<CP>{int} activities[p in Persons,t in Slots](cp,possibleActivity[p]);
var<CP>{int} underDemand[t in Slots](cp,0..(sum(a in Activities) demand[t,a]));

minimize<cp>
    sum(t in Slots) underDemand[t]
subject to {
    forall (p in Persons)
        cp.post(alldifferent(all(t in Slots) activities[p,t]), onDomains);
    forall (t in Slots) {
        int[] demand_t = all(a in Activities) demand[t,a];
        var<CP>{int}[] activities_t = all(p in Persons) activities[p,t];
        cp.post(softAtLeast(demand_t, activities_t, underDemand[t]));
    }
}
using
    labelFF(activities);

```

Statement 14.4: Personnel Scheduling with Soft Constraints (personal-scheduling-cp-soft.co)

Chapter 15

Propagators

COMET is an extensible system, that allows users to implement their own propagators in the COMET language. A propagator, also called a filtering algorithm, is in charge of removing inconsistent values, i.e., values that do not belong to any solution, from the domain of variables. From an object-oriented programming point of view, a propagator implements a constraint.

In this chapter, we first develop a propagator for the modulo constraint:

$$y = x \bmod m$$

where m is a positive integer and x, y are integer CP variables, i.e., of type `var<CP>{int}`. We start with a simple propagator, that implements this constraint using AC3 events. We then describe a more efficient, fully incremental, implementation, which uses two powerful features supported by COMET, trailable data structures and AC5 events. We conclude this chapter by showing how to build a reified constraint in COMET and then wrap a function around it.

15.1 Propagator for the Modulo Constraint using AC3 Events

Statement 15.1 depicts the general structure of a simple propagator that implements the modulo constraint. What is important to point out is that the propagator basically implements the `UserConstraint<CP>` interface, as indicated by its declaration:

```
class ModuloCstr extends UserConstraint<CP>
```

In what follows, we show how to fill the body of the constructor and the `post` and `propagate` methods:

- In the constructor we instantiate instance variables and give a state to the propagator
- The `post` method is called once, when the constraint is added to the model. Typically, this is where we first check the consistency of the constraint, i.e., if the constraint has a solution or not. This is also the right place to inform the solver about the events on which the `propagate` method should be called
- Finally, the `propagate` method will be in charge of removing inconsistent values from the domains of the variables involved in the constraint

Note that the result of `post` and `propagate` is of type `Outcome<CP>`. There are three possible outcomes:

Failure Returned when it is detected that the constraint is inconsistent, i.e., it has no solution. This is the case, each time one of the domains becomes empty

Success Returned when the constraint is satisfied for any value assignment. For example, this should be the return value of a propagator for the constraint $x < y$, with domains $x \in \{1, 2\}$ and $y \in \{6, 8\}$

Suspend Returned in all other cases. It means that the propagator needs to be called again, but it has removed all the values it could for now

The complete implementation of the modulo constraint is given in Statement 15.2. The parameters of the constraint are stored as instance variables of the constraint and are initialized in the constructor.

```
import cotfd;
class ModuloCstr extends UserConstraint<CP> {
    ModuloCstr(var<CP>{int} x,var<CP>{int} y,int m) : UserConstraint<CP>() { }

    Outcome<CP> post(Consistency<CP> c1) {
        return Suspend;
    }

    Outcome<CP> propagate() {
        return Suspend;
    }
}
```

Statement 15.1: General Structure of the Modulo Constraint

```

import cotfd;
class ModuloCstr extends UserConstraint<CP> {
    var<CP>{int} _x;
    var<CP>{int} _y;
    int _m;

    ModuloCstr(var<CP>{int} x,var<CP>{int} y,int m) : UserConstraint<CP>() {
        _x = x; _y = y; _m = m;
    }

    Outcome<CP> post(Consistency<CP> cl) {
        assert(_m>0);
        if (_y.updateMin(0) == Failure)
            return Failure;
        if (_y.updateMax(_m-1) == Failure)
            return Failure;
        if (propagate() == Failure)
            return Failure;
        _x.addDomain(this);
        _y.addDomain(this);
        return Suspend;
    }

    Outcome<CP> propagate() {
        forall (v in _x.getMin().._x.getMax() : _x.memberOf(v))
            if (!_y.memberOf(v%_m))
                if (_x.removeValue(v) == Failure)
                    return Failure;
        forall (v in _y.getMin().._y.getMax() : _y.memberOf(v)) {
            bool support = false;
            forall (w in 0.._x.getMax()/_m)
                if (_x.memberOf(w * _m + v)) {
                    support = true;
                    break;
                }
            if (!support && _y.removeValue(v) == Failure)
                return Failure;
        }
        return Suspend;
    }
}

```

Statement 15.2: Modulo Constraint Implementation (modulo-ac3-cp.co)

Propagate We describe the `propagate` method before the `post` method, since the former is called by the latter. There are two main loops in the body of `propagate`. They both use the method `removeValue(v)` defined for `var<CP>{int}`. This method removes the `int` value `v` from the domain of the variable, and should only be used in COMET propagators. It returns `Suspend`, if the domain does not become empty as a result of the removal, `Success`, if the domain becomes a singleton, and `Failure`, otherwise.

The first loop iterates over the values `v` in the domain of `_x`, and removes any inconsistent value. If `v % _m` does not belong to the domain of `_y`, we can remove `v` from the domain of `_x`. If `v` was the last value in the domain, `removeValue` will return `Failure` indicating that the domain is empty. This means that the constraint has no solution and the `propagate` method also returns `Failure`.

```
forall (v in _x.getMin().._x.getMax() : _x.memberOf(v))
  if (!_y.memberOf(v%_m))
    if (_x.removeValue(v) == Failure)
      return Failure;
```

The second loop removes inconsistent values from the domain of `_y`. For each value `v` in `_y`'s domain, the inner loop searches for a supporting value in `_x`'s domain. In other words, it looks in `x`'s domain for a value of the form $w * _m + v$.

```
bool support = false;
forall (w in 0.._x.getMax()/_m)
  if (_x.memberOf(w * _m + v)) {
    support = true;
    break;
  }
```

If no support is found, value `v` is removed from `_y`'s domain. A `Failure` is returned, if `y`'s domain becomes empty after the removal.

```
if (!support && _y.removeValue(v) == Failure)
  return Failure;
```


Post The `post` method of the propagator starts by setting the minimum and maximum of `_y` to 0 and `_m-1`, respectively. It uses the methods `updateMin` and `updateMax` for `var<CP>{int}`. Both methods should only be used in propagators, and return `Failure`, if they lead into an empty domain.

```
if (_y.updateMin(0) == Failure)
    return Failure;
if (_y.updateMax(_m-1) == Failure)
    return Failure;
```

Then it calls the `propagate` method, to check the consistency and make some filtering. If the constraint is not consistent, it immediately returns a `Failure` without going any further. Otherwise it registers the propagator with `addDomain`, so that the `propagate` method is called, after every modification of `_x` or `_y`'s domain.

```
if (propagate() == Failure)
    return Failure;
_x.addDomain(this);
_y.addDomain(this);
return Suspend;
```

15.2 Incremental Propagator for the Modulo Constraint with AC5 Events

The propagator of Statement 15.2 is not as efficient. We can implement a more efficient, incremental version of the propagator, by taking advantage of the following concepts:

- trailable sets, that are restored by the system upon backtracking
- AC5 events, a more fine-grained event mechanism originally introduced in [1]

These concepts will be illustrated with reference to the implementation shown in Statements 15.3 and 15.4. The first statement shows the class declaration and the constructor, while the second one contains the implementation of the rest of the methods. We now go into some more details.

Trailable sets are simply sets that are automatically restored by the system upon backtracking. Assume that a state has been introduced to a propagator. Using trailable rather than usual sets, we can focus on the descent along a branch of the search tree, without having to worry about restoring the state at every backtrack. This is automatically taken care of by the system. Thanks to the trailable sets, we can proceed with the implementation of a constraint, as if there were no backtracks. We introduce one trailable set for each value in the range $0..m-1$, to contain the support for this value in the domain of $_x$:

```
_supporty = new trail<set<int>>[0.._m-1](_x.getSolver());
```

```

import cotfd;
Solver<CP> cp();

class ModuloCstr extends UserConstraint<CP> {
    var<CP>{int} _y;
    var<CP>{int} _x;
    int _m;
    trail {set{int}}[] _supporty;

    ModuloCstr(var<CP>{int} x,var<CP>{int} y,int m) : UserConstraint<CP>() {
        _y = y;
        _x = x;
        _m = m;
        _supporty = new trail{set{int}}[0.._m-1](_x.getSolver());
    }

    Outcome<CP> post(Consistency<CP> c1);
    Outcome<CP> valRemove(var<CP>{int} v,int val);
}

```

Statement 15.3: Modulo Constraint with AC5 Events (Part 1/2) [Statement [15.4](#) shows the method implementation] (modulo-ac5-cp.co)

```

Outcome<CP> ModuloCstr::post(Consistency<CP> c1) {
    forall (v in _x.getMin().._x.getMax(): _x.memberOf(v))
        if (!_y.memberOf(v % _m)) {
            if (_x.removeValue(v) == Failure)
                return Failure;
        }
    else
        _supporty[v%_m].insert(v);
    if (_y.updateMin(0) == Failure)
        return Failure;
    if (_y.updateMax(_m-1) == Failure)
        return Failure;
    forall (v in _y.getMin().._y.getMax())
        if (_supporty[v].getSize() == 0)
            if (_y.removeValue(v) == Failure)
                return Failure;
    _y.addAC5(this);
    _x.addAC5(this);
    return Suspend;
}

Outcome<CP> ModuloCstr::valRemove(var<CP>{int} v,int val) {
    if (v.getId() == _y.getId()) {
        forall (i in _supporty[val])
            if (_x.removeValue(i) == Failure)
                return Failure;
    }
    else {
        _supporty[val%_m].del(val);
        if (_supporty[val%_m].getSize() == 0)
            if (_y.removeValue(val%_m) == Failure)
                return Failure;
    }
    return Suspend;
}

```

Statement 15.4: Modulo Constraint with AC5 Events (Part 2/2) (modulo-ac5-cp.co)

Post The `post` method, shown on Statement 15.4, first computes the supports sets and filters the domain of `_x`:

```
forall (v in _x.getMin().._x.getMax(): _x.memberOf(v))
  if (!_y.memberOf(v % _m)) {
    if (_x.removeValue(v) == Failure)
      return Failure;
  }
  else
    _supporty[v%_m].insert(v);
```

It then filters the domains, while checking the consistency of the constraint:

```
if (_y.updateMin(0) == Failure)
  return Failure;
if (_y.updateMax(_m-1) == Failure)
  return Failure;
forall (v in _y.getMin().._y.getMax())
  if (_supporty[v].getSize() == 0)
    if (_y.removeValue(v) == Failure)
      return Failure;
```

Finally, it subscribes the propagator to the AC5 events on the domains of variables `_x` and `_y`:

```
_y.addAC5(this);
_x.addAC5(this);
```

The following example illustrates the computed support sets. If $m = 4$ and x 's domain is $1..9$, the support set for 0 is $\{4, 8\}$, for 1 is $\{1, 5, 9\}$, for 2 is $\{2, 6\}$ and for 3 is $\{3, 7\}$. Support sets are very useful for filtering, because every time a value is suppressed (removed) from y 's domain, we know that the values of the corresponding support set must also be removed from x 's domain. Conversely, whenever the support set of a value in y 's domain becomes empty, this value must also be removed from the domain of y .

This fine-grained filtering is made possible through the use of AC5 events. In short, AC5 events allow to have more than one filtering method called, each time a domain is modified. Each method corresponds to a specific event on the domain such as

- removal of a value
- binding of a variable
- modification of the domain's bounds

Value Removal Moreover, these methods also receive information about the kind of modification to the domain that triggered it. In our modulo example, we subscribe the propagator to the AC5 events on variables `_x` and `_y`. Now, each time a value is removed from one of these variables' domain, the method `valRemove` is called, passing as arguments the variable (`_x` or `_y`) and the value suppressed from its domain. The implementation of `valRemove` is given in Statement 15.4. It deals with the following two cases: If the value `val` has been removed from the variable `_y`, then it removes from `_x`'s domain the values of the corresponding support set `_supporty[val]`:

```
if (v.getId() == _y.getId()) {
  forall (i in _supporty[val])
    if (_x.removeValue(i) == Failure)
      return Failure;
}
```

If the value `val` has been removed from variable `_x`, then it also removed from the support set `_supporty[val%_m]`. If this support set becomes empty, then the value `val%_m` can be safely removed from `_y`'s domain, since this value has no more support in `_x`'s domain.

```
else {
  _supporty[val%_m].del(val);
  if (_supporty[val%_m].getSize() == 0)
    if (_y.removeValue(val%_m) == Failure)
      return Failure;
}
```

Note that if, at any point, `_x` or `_y`'s domain becomes empty, a `Failure` is returned.

Example The following block of code shows a small example using the `ModuloCstr`. The constraint posted is $y = x \bmod 4$ with $x, y \in [1..10]$.

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x(cp,1..10);
var<CP>{int} y(cp,1..10);
explore<cp>
  cp.post(ModuloCstr(x,y,4));
using {
  label(x);
  label(y);
  cout << x << " , " << y << endl;
}
```

The output of this program consists of all possible solutions, namely:

```
1 , 1
2 , 2
3 , 3
5 , 1
6 , 2
7 , 3
9 , 1
10 , 2
```

15.3 Implementing a Reified Constraint

A reified constraint is a constraint that can be negated with `!` or used in logical expressions built with `||`, `&&`, `==`, `=>`. We explain how to build such a constraint in COMET and how to wrap it with a function. For the sake of clarity, we choose to build a very simple example: the reified equality between two integer variables.

In other words, for a `var<CP>{bool} b` and two `var<CP>{int} x1, x2`, we want to implement a constraint that is basically equivalent to the following COMET instruction:

```
cp.post(b == (x1==x2))
```

The class implementing this constraint is given in Statements 15.5 and 15.6 and it extends COMET's `UserConstraint<CP>` class. The first statement includes the constructor and the `post` method implementation, whereas the second one shows the implementation of the remaining methods.

The boolean variable `b` represents the truth value of the equality, i.e., `b = true`, if and only if `x1 = x2`. We implement an incremental propagator. The idea is that, when the intersection of the domains of `_x1` and `_x2` becomes empty, then `_b` must be set to `false`, since the equality cannot hold.

This can be done in an efficient, incremental way, using as instance variable the *trailable* integer `_interSize`, to represent the size of the intersection. Of course, this value will be correctly initialized and updated, whenever a modification of the domains occurs, as we'll explain later, when describing the implementation of `post`, `propagate` and `valRemove`.

```

class ReifiedEquality<CP> extends UserConstraint<CP> {
  Solver<CP> _cp;
  var<CP>{bool} _b;
  var<CP>{int} _x1;
  var<CP>{int} _x2;

  trail<int> _interSize; //size of the domain intersection of _x1 and _x2

  ReifiedEquality<CP>(var<CP>{bool} b, var<CP>{int} x1, var<CP>{int} x2)
    : UserConstraint<CP>() {
    _cp = x1.getSolver();
    _b = b;
    _x1 = x1;
    _x2 = x2;
  }

  Outcome<CP> post(Consistency<CP> c1) {
    //initialize the size of intersection of dom(x1) and dom(x2)
    _interSize = new trail<int>(_cp);
    set<int> dom1 = filter (v in _x1.getMin().._x1.getMax()) (_x1.memberOf(v));
    _interSize := sum(v in dom1) _x2.memberOf(v);

    //register to AC5 events
    _x1.addDomain(this);
    _x2.addDomain(this);
    _b.addBind(this);
    return Suspend;
  }

  Outcome<CP> propagate();

  Outcome<CP> valRemove(var<CP>{int} x,int v);
}

```

Statement 15.5: Reified Equality Constraint (Part 1/2) (reified-equality-cp.co)

Post The complete implementation of `post` is shown on Statement 15.5. The method works as follows: It first initializes correctly the size of the domain intersection:

```
_interSize = new trail<int>(_cp);
set<int> dom1 = filter(v in _x1.getMin().._x1.getMax())(_x1.memberOf(v));
_interSize := sum(v in dom1) _x2.memberOf(v);
```

It then subscribes the constraint to the AC5 events for `_x1` and `_x2` and, finally, it states that the `propagate()` method be also called, whenever variable `_b` becomes bound.

```
_x1.addDomain(this);
_x2.addDomain(this);
b.addBind(this);
```

Propagate The complete implementation of the `propagate` method is given in Statement 15.6. It takes care of with three special cases.

In the first case, variable `_b` is already bound. Depending on `_b`'s value, we need to dynamically add to the solver either an equality or a dis-equality constraint between `_x1` and `_x2`. Note that we can only do so using the `tryPost` method of `Solver<CP>`, since the usual `post` method cannot be used inside propagators. Either way, we then return a `Success`, so that the constraint is deactivated. This is because the constraint posted with `tryPost` will take over after this point.

```
if (_b.bound()) {
  if (_b) {
    if (!_cp.tryPost(_x1 == _x2))
      return Failure;
  }
  else
    if (!_cp.tryPost(_x1 != _x2))
      return Failure;
  return Success;
}
```

```

Outcome<CP> ReifiedEquality<CP>::propagate() {
  if (_b.bound()) {
    if (_b) {
      if (!_cp.tryPost(_x1 == _x2))
        return Failure;
    }
    else
      if (!_cp.tryPost(_x1 != _x2))
        return Failure;
    return Success;
  }
  if (_x1.bound() && _x2.bound()) {
    if (_b.bindValue(_x1 == _x2) == Failure)
      return Failure;
    return Success;
  }
  if (_interSize == 0) {
    if (_b.bindValue(false) == Failure)
      return Failure;
    return Success;
  }
  return Suspend;
}

Outcome<CP> ReifiedEquality<CP>::valRemove(var<CP>{int} x,int v) {
  if (x == _x1) {
    if (_x2.memberOf(v))
      _interSize := _interSize-1;
  }
  else // x == _x2
    if (_x1.memberOf(v))
      _interSize := _interSize-1;
  return Suspend;
}

```

Statement 15.6: Reified Equality Constraint (Part 2/2) (reified-equality-cp.co)

In the second case, both variables `_x1` and `_x2` are bound, we use the `bindValue` method, to bind variable `_b` to **true**, if `_x1` and `_x2` are bound to the same value, or to **false** otherwise.

```
if (_x1.bound() && _x2.bound()) {  
    if (_b.bindValue(_x1 == _x2) == Failure)  
        return Failure;  
    return Success;  
}
```

The last case considered is when the intersection becomes empty. In that case, variable `_b` is simply set to **false**:

```
if (_interSize == 0) {  
    if (_b.bindValue(false) == Failure)  
        return Failure;  
    return Success;  
}
```

Note that, as expected, the return value of the `bindValue` method is **Failure**, if we attempt to bind a variable to a value that is not any more in its domain. Also `bindValue` can only be used inside propagators.

Value Removal Finally, the `valRemove` method, which is executed whenever `_x1` or `_x2` loses a value, updates the size of the domain intersection. When the value is lost in one variable, but is still present in the other variables, `_interSize` is decremented. This is the excerpt of the code that deals with the case that value `v` is lost from `x1`'s domain (the other case is symmetric):

```
if (x == _x1) {  
    if (_x2.memberOf(v))  
        _interSize := _interSize-1;  
}
```

The complete code appears on Statement 15.6. We let the `propagate` method deal with the propagation, when the intersection becomes empty. Note that `propagate` is called after all `valRemove` calls. So, a slight optimization would be to also deal with this case in `valRemove`, to immediately act on `_b` when the empty intersection occurs.

Function Wrapper The final step, in order to make the reified constraint user-friendly in logical expressions, is to wrap it into a function that returns the boolean variable of the reified constraint. This way, that boolean variable is completely hidden and is created internally inside the function. The function also posts the reified user constraint:

```
function var<CP>{bool} equality(var<CP>{int} x1, var<CP>{int} x2) {
    var<CP>{bool} b(x1.getSolver());
    Constraint<CP> C = ReifiedEquality<CP>(b,x1,x2);
    x1.getSolver().post(C);
    return b;
}
```

We conclude this section with a complete small example, that illustrates the use of the reified equality constraint in a logical constraint:

```
Solver<CP> cp();
var<CP>{int} x(cp,1..3);
var<CP>{int} y(cp,2..5);
var<CP>{int} z(cp,1..2);
solve<cp>
    cp.post(equality(x,y) || (x!=y && x<=z) );
```

Chapter 16

Scheduling

This chapter gives an overview of COMET's Scheduling module, which is built on top of the Constraint Programming module, and offers high-level modeling and search abstractions specific to scheduling. COMET features a rich modeling ability to cover a wide variety of scheduling applications, as we'll see in the examples of this chapter.

Typically, in a COMET model all scheduling abstractions are contained in a `Scheduler<CP>` object. The two main actors in a scheduling problem are the *activities*, also called tasks, and the *resources*, also called machines. Activities are represented by objects of type `Activity<CP>` encapsulating three `var<CP>{int}` variables and the constraints linking them:

- the starting time, accessed through the `start()` method
- the ending time, accessed through the `end()` method
- the duration, accessed through the `duration()` method

In addition, COMET supports breakable activities to model, for example, week-end breaks.

Depending on the scheduling application's structure, different types of resources may be more appropriate for modeling and searching. For this reason, COMET supports several different types of resources, covering a variety of special cases.

We now give a summary of these types of resources. In the following sections, we are going to see examples of their application.

Unary Resources model machines on which at most one activity can execute at a time.

Unary Sequences extend the functionality of unary resources with successor/predecessor reasoning for scheduled activities.

Discrete Resources have a discrete capacity that may vary over time. More than one activity can execute in parallel on a discrete resource.

Reservoirs model situations in which some activities require resources provided by other activities.

State Resources model machines that need to be in a certain state, to perform some activity. A resource's state can change and we can specify transition times between different states.

Resource Pools model situations in which some activities can be served by more than one resource. Currently, COMET supports pools of unary resources, unary sequences and discrete resources.

16.1 Unary Resources: The Job Shop Problem

In this section we will see

- how to use activities together with unary resources and precedence constraints
- an example of non-deterministic search for unary resources using the **rank** method

We are focusing on a standard problem in Operations Research, the Job-Shop Scheduling Problem, that can be described as follows:

- We are given a set of jobs, each of which consists of a sequence of tasks
- Each task t has a duration $d(t)$ and can only be performed by a specified machine $m(t)$
- Each machine can only execute one task at a time
- The goal is to minimize the project completion time, also referred to as the makespan

The complete model is given in Statement 16.1. The input data of the problem is read from an instance file and is basically characterized by the following two matrices:

```
int duration[Jobs,Tasks];
int machine[Jobs,Tasks];
```

Each row of these matrices stores the data related to a job in the range **Jobs**:

- `duration[j,t]` is the duration of task `t` of job `j`
- `machine[j,t]` is the machine that needs to perform task `t` of job `j`

The model first creates a `Scheduler<CP>` object, which is an extension of `Solver<CP>`. This means that we can treat `Scheduler<CP>` as a CP solver, adding variables and constraints, in addition to scheduling-specific methods. A scheduler must be initialized with a time-horizon parameter that represents the time during which the schedule is active. In our case, we set the time-horizon to a trivial upper bound on the makespan, computed as the sum of durations of all tasks.

```
int          horizon = sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP> cp(horizon);
```

```

import cotfd;
int    nbJobs;      // read from file
int    nbTasks;     // read from file
range  Jobs         = 0..nbJobs-1;
range  Tasks        = 0..nbTasks-1;
range  Machines      = Tasks;
int    nbActivities = nbJobs * nbTasks;
range  Activities    = 0..nbActivities+1;

int    duration[Jobs,Tasks]; // read from file
int    machine[Jobs,Tasks];  // read from file

int          horizon = sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP> cp(horizon);
Activity<CP>  a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP>  makespan(cp,0);
UnaryResource<CP> r[Machines](cp);

minimize<cp>
  makespan.start()
subject to {
  forall (j in Jobs, t in Tasks : t != Tasks.getUp())
    a[j,t].precedes(a[j,t+1]);
  forall (j in Jobs)
    a[j,Tasks.getUp()].precedes(makespan);
  forall (j in Jobs, t in Tasks)
    a[j,t].requires(r[machine[j,t]]);
}
using {
  forall (m in Machines) by (r[m].localSlack(),r[m].globalSlack())
    r[m].rank();
  makespan.scheduleEarly();
  cout << "Makespan: " << makespan.start() << endl;
}

```

Statement 16.1: CP Model for Job-Shop Scheduling (jobshop-cp.co)

The model then creates an array **a** of **Activity<CP>** objects, one for each task in the schedule. It also defines a dummy activity of duration zero, called **makespan**. This activity represents the end of the schedule. Finally, the model instantiates an array **r** of **UnaryResource<CP>** objects, one for each machine, to model the restriction that at most one activity can use each resource at any given time.

```
Activity<CP>      a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP>      makespan(cp,0);
UnaryResource<CP> r[Machines](cp);
```

The objective function to minimize is the total duration of the schedule, which is equal to the starting date of the **makespan** activity:

```
minimize<cp>
    makespan.start()
```

Now let us describe the model's constraints. The first set of constraints enforces that the task of each job are performed in sequence:

```
forall (j in Jobs, t in Tasks : t != Tasks.getUp())
    a[j,t].precedes(a[j,t+1]);
```

The next set of constraints defines the semantics of the **makespan** activity, namely that it can only start after the last task of each job has finished.

```
forall (j in Jobs)
    a[j,Tasks.getUp()].precedes(makespan);
```

Finally, for each activity, we specify the machine on which it can be performed:

```
forall (j in Jobs, t in Tasks)
  a[j,t].requires(r[machine[j,t]]);
```

The search procedure relies on the fact that, once the tasks are completely ordered (ranked) within each job and machine, the smallest possible makespan is equal to the sum of durations of the tasks along the longest path of the precedence graph. Note that the corresponding schedule can be easily found, by assigning each task as early as possible in a topological sort on the precedence graph (PERT scheduling). The complete search procedure is concisely written as:

```
forall (m in Machines) by (r[m].localSlack(), r[m].globalSlack())
  r[m].rank();
makespan.scheduleEarly();
```

The next unary resource to be ranked is selected according to its `localSlack` and ties are broken with its `globalSlack`. The local slack of a unary resource is a precise measure of the tightness of the resource, whereas the global slack is a rough approximation, computed as follows: consider all activities that are not yet ranked, and compute the earliest starting date, `s`, the latest finishing date, `e`, and the total duration, `d`, of all these activities. Then the global slack is equal to $(e-s)-d$.

The non-deterministic instruction `r[m].rank()` ranks the activities of resource `r[m]`, i.e., it sets the order in which activities are performed on the resource (respecting, of course, any job precedences). Once all resources have been *ranked*, we end up with an instance of PERT scheduling, described earlier: the starting time of the `makespan` activity will correspond to the longest path in the precedence graph, and the instruction

```
makespan.scheduleEarly()
```

fixes its starting time to this minimum value.

16.2 Unary Sequence Resources

In many scheduling applications, it is useful to determine the activity that is scheduled immediately before or after a given task; in other words, the *predecessor* or the *successor* of a given activity. It is not very simple to determine this information when using the standard unary resource structure, `UnaryResource<CP>`, presented in Section 16.1.

Informally, the `UnarySequence<CP>` resource can be viewed as an extension of `UnaryResource<CP>` that allows for successor/predecessor reasoning. The resource utilizes two special `Activity<CP>` objects, that correspond to the start and the end of the schedule: the *source* and the *sink*, respectively. These can be retrieved with the following methods:

```
Activity<CP> getSource()
Activity<CP> getSink()
```

Most methods of `UnaryResource<CP>` are also available for `UnarySequence<CP>`. For example, one can still use the method `rank()` to schedule the different activities on the resource. However, there are additional functions supported for this purpose, that take advantage of the successor/predecessor logic. These are summarized here:

```
void sequence()
void sequenceForward()
void sequenceBackward()
```

The first two methods, `sequence` and `sequenceForward` are synonyms; they work as follows: First, they select the activity to schedule after the source activity, i.e., the first activity to use the resource. Once this is determined, they assign the successor of the first activity. This goes on, until all activities have been assigned a successor. Note that the successor of the last activity is the sink activity. Naturally, the precedence constraints imposed by the scheduling problem are taken into account in the above assignment.

The `sequenceBackward` method works in an analogous way: it first assigns a predecessor to the sink, and then continues assigning predecessors, until it reaches the source. Of course, in the end, this lead to a ranking of all the activities on the sequence resource, but the reasoning to reach that ranking is fundamentally different from that of the `rank` method.

On top of these methods, `UnarySequence<CP>` offers methods for retrieving the successor and the predecessor of each activity, once the resource is ranked. These are shown next.

```
Activity<CP> getSuccessor(Activity<CP>)
Activity<CP> getPredecessor(Activity<CP>)
Activity<CP> getSuccessor(Activity<CP>, Solution)
Activity<CP> getPredecessor(Activity<CP>, Solution)
```

Observe that the `getSuccessor` and `getPredecessor` methods can also take as argument a `Solution` object. This can be very useful, in cases in which it is desirable to perform LNS based on successors and predecessors, for example, in schemes based on the critical path.

Statement 16.2 shows how we can modify the `UnaryResource<CP>`-based model shown on Statement 16.1, in order to make use of `UnarySequence<CP>`. It also shows a trivial application of the `getSuccessor` method in printing the order of the activities on one of the machines.

Observe the fact that it takes only minimal changes, in order to adapt the model to the new resource: using `UnarySequence<CP>` instead of `UnaryResource<CP>` and `sequenceForward` instead of `rank`. Of course, the additional functionality comes with some overhead in efficiency, in order to maintain the additional information. For this reason, it is recommended to use unary sequence resources, only when it is really necessary to use predecessor and successor information, or if the actual problem at hand has a combinatorial structure that would allow the solver to benefit from using `sequence` rather than `rank`.

```

import cotfd;
range Jobs;
range Tasks;
range Machines;
range Activities;

int duration[Jobs,Tasks];
int machine[Jobs,Tasks];

int horizon = sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP> cp(horizon);
Activity<CP> a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP> makespan(cp,0);
UnarySequence<CP> r[Machines](cp);

minimize<cp>
    makespan.start()
subject to {
    forall (j in Jobs, t in Tasks : t != Tasks.getUp())
        a[j,t].precedes(a[j,t+1]);
    forall (j in Jobs)
        a[j,Tasks.getUp()].precedes(makespan);
    forall (j in Jobs, t in Tasks)
        a[j,t].requires(r[machine[j,t]]);
}
using {
    forall (m in Machines) by (r[m].localSlack(),r[m].globalSlack())
        r[m].sequenceForward();
    makespan.scheduleEarly();

    cout << "Makespan: " << makespan.start() << endl;
    Activity<CP> A = r[1].getSource();
    cout << "Source --> ";
    A = r[1].getSuccessor(A);
    while (A != r[1].getSink()) {
        cout << A << " ---> " ;
        A = r[1].getSuccessor(A);
    }
    cout << "Sink" << endl;
}

```

Statement 16.2: CP Model for Job-Shop Scheduling Using Unary Sequences
(jobshop-cp-sequence.co)

16.3 Discrete Resources: Cumulative Job Shop

In this section we will see:

- how to use activities together with discrete resources and precedence constraints
- an example of non-deterministic search for discrete resources using `setTimes`
- a large neighborhood search on a cumulative job-shop problem

16.3.1 Cumulative Job Shop Scheduling

The Cumulative Job-Shop Scheduling Problem is very similar to the standard Job-Shop Problem, with the difference that tasks execute on *discrete* rather unary resources. Discrete (or cumulative) resources have a discrete capacity that can vary over time. Each activity requires a number of units from some discrete resource. As long as the total demand is within the capacity limits of a discrete resource, more than one activity can use the resource at the same time. Note that unary resources can be viewed as special cases of discrete resource of capacity 1, which, of course, only makes sense if every activity also has a demand of 1.

In the Cumulative Job-Shop, tasks can also be grouped into jobs, so that each job is composed of a sequence of tasks, that must be performed in the given order. For the sake of simplicity, in the particular problem of this section, all jobs have the same number of tasks, all activities require only one unit from their chosen resource, and all discrete resources have the same capacity, but it easy to generalize each of these assumptions. Once again, the goal is to minimize the total duration of the project, i.e., the makespan.

The model in Statement 16.3 is not very different from the one in statement 16.1. There is only a slight difference in the way job precedences are dealt with. Instead of explicitly specifying the sequence of tasks within each job, they are stored into a set of `tuples` representing task precedences. This set of preferences, along with all the other data of the instance solved, is stored in a class `JobshopParser` initialized from a data file, and retrieved before stating the model. The definition of the precedence tuple and the code to initialize `JobshopParser` and retrieve the set of precedences is shown next:

```
tuple          PrecTuple { int o; int d; }
JobshopParser  parser(fName); // read input from file
set{PrecTuple} precedences = parser.getPrecedences();
```

The rest of the data is retrieved in a similar fashion.

```

import cotfd;
tuple      PrecTuple { int o; int d; }
JobshopParser  parser(fName); // read input from file

range      Activities      = parser.getActivities();
range      Machines       = parser.getMachines();
int        cap             = parser.getCapacity();
int []     duration       = parser.getDurationArray();
int []     machine         = parser.getMachineArray();
set{PrecTuple} precedences = parser.getPrecedences();

int        horizon = sum(a in Activities) duration[a];
Scheduler<CP>      cp(horizon);
Activity<CP>      act[a in Activities] (cp,duration[a]);
Activity<CP>      makespan(cp,0);
DiscreteResource<CP> r[Machines] (cp,cap);

minimize<cp>
    makespan.start()
subject to {
    forall (t in precedences)
        act[t.o].precedes(act[t.d]);
    forall (a in Activities)
        act[a].precedes(makespan);
    forall (a in Activities)
        act[a].requires(r[machine[a]],1);
}
using {
    setTimes(act);
    makespan.scheduleEarly();
    cout << "Makespan: " << makespan.start() << endl;
    cout << "fail: " << cp.getNFail() << endl;
}

```

Statement 16.3: CP Model for the Cumulative Job-Shop Problem ([cumulative-jobshop-cp.co](#))

Obviously, in terms of modeling, the only real difference compared to the simple job-shop problem is the fact that `DiscreteResource<CP>` objects are used instead of `UnaryResource<CP>` to represent the machines. However, the search with discrete resources is fundamentally different, compared to unary resource scheduling:

```
using {  
    setTimes(act);  
    makespan.scheduleEarly();  
}
```

16.3.2 Explanation of `setTimes`

The `setTimes` method used in the search non-deterministic assigns the starting time of the activities given in its argument. This is a brief explanation of how `setTimes` works, based on [3]. If `d` is the earliest date at which an activity can start, COMET selects a task that can be scheduled at date `d`. On backtracking, another task is selected to start at `d`, or execution fails (if no such task is available.) Tasks that are not scheduled at their earliest date are said to be *postponed*, and remain so, until their starting dates are updated. Given the nature of the problem and the properties of constraint propagation in COMET, one of these activities has to start at date `d` in an optimal solution.

Once such a task is chosen, COMET considers the next date `n` $\geq d$ in which a non-postponed activity can start and, also, all the tasks that can start at date `n`. Once again, one of these tasks is non-deterministically chosen to start at date `n`. Note that some tasks may have a minimal starting date smaller than `n`. These are, of course, the postponed activities, because there exists an optimal solution, in which these tasks do not take a value in the range `d+1..n`. If the current partial solution can be extended to an optimal solution, the starting dates of these tasks have to be updated, enabling COMET to reconsider them. Moreover, if the latest starting date of one of the postponed activities is smaller than `n`, then this postponed activity cannot have its starting date updated, meaning that the current partial solution cannot be extended to an optimal solution. This process is iterated until all the tasks have been assigned a starting date (a solution has been found) or only postponed activities remain (failure).

Non-deterministic search using `setTimes` is very powerful, since it encapsulates dominance rules that prune the search space. However, some solution could be missed, if the problem has negative precedence constraints, of the form `a.start >= b.end-3`. In situations like that, the method `scheduleTimes` should be used, since it also assigns starting dates of activities non-deterministically, but without using dominance rules.

In general, `setTimes` should only be used on an array of activities, if the following conditions hold:

- The activities require discrete or unary resources
- Activity durations are fixed
- There are no negative precedence constraints

16.3.3 Adding an LNS Component

When the problem becomes too difficult to solve exactly, it may be useful to transform it into a large neighborhood search. The idea is to use the CP model to explore exactly a large neighborhood obtained by relaxing the domain of some variables. The process of relaxing and solving is repeated until a stopping criterion is met (typically the time). An iteration terminates either because it has improved the solution, or because a time or failure limit is reached before restarting. Statement 16.4 shows how to adapt the model of Statement 16.3 to integrate a Large Neighborhood Search (LNS). LNS is controlled by the following parameters:

```
int prob = 20;
UniformDistribution distr(1..100);
cp.limitTime(60);
cp.lnsOnFailure(100);
```

In this code, `prob` is the probability that a task is relaxed in the next iteration and `distr` is a random uniform number generator in the range 1..100. A time limit of 60 seconds is allowed for the overall search, and if the solution does not improve within 100 failures, the **onRestart** block is run:

```
Solution s = cp.getSolution();
if (s != null) {
    set{Activity<CP>} R();
    forall (a in Activities)
        if (distr.get() <= prob)
            R.insert(act[a]);
    cp.relaxPOS(s,R);
}
```

This adds each activity, with probability 20%, to a set `R` of activities to be relaxed. Remember that

```

... // same data, activities, resources
int prob = 20;
UniformDistribution distr(1..100);
cp.limitTime(60);
cp.lnsOnFailure(100);

minimize<cp>
    makespan.start()
subject to
    ... // the same constraints
using
    ... // the same search
onRestart {
    Solution s = cp.getSolution();
    if (s != null) {
        set{Activity<CP>} R();
        forall (a in Activities)
            if (distr.get() <= prob)
                R.insert(act[a]);
        cp.relaxPOS(s,R);
    }
}

```

Statement 16.4: CP Model for Cumulative Job-Shop with LNS (cumulative-jobshop-cp-lns.co)

before each restart, all domains are restored. This means that in the beginning of the **onRestart** block, everything is restored, except for the constraints posted in the **subject to** block. The instruction `cp.relaxPOS(s,R)` constrains the problem by building a partial-order schedule on the solution `s`, in which all precedences involving activities in `R` are relaxed, by posting a set of constraints:

$$\text{act}[i].\text{end}() \leq \text{act}[j].\text{start}()$$

for all pairs of activities `act[i]` and `act[j]` not in `R`, for which `act[i]` precedes `act[j]` in solution `s`.

16.4 Reservoirs

As we saw in Discrete Resources, activities may use a number of units from the resource, but those units are returned to the resource upon termination of the activity. Generalizing this concept leads naturally to another type of resource, the **Reservoir**.

At any given moment during the schedule, reservoirs have a non-negative discrete capacity. Similar to discrete resources, activities can use a number of units from the reservoir, not exceeding capacity, that are returned when the activity finishes. However, reservoirs offer more modes of interaction with activities, depending on which of the following methods of **Activity<CP>** is used to specify the relation of the activity to the reservoir:

```
void requires(Reservoir<CP>,int)
void consumes(Reservoir<CP>,int)
void provides(Reservoir<CP>,int)
void produces(Reservoir<CP>,int)
```

Note that these four methods also have counterparts, in which the second argument is a **var<CP>{int}** rather than an **int**. What characterizes **Reservoir<CP>** is essentially the different modes of operation of activities. These are now explained in more detail.

requires works as in discrete resources. The activity uses a number of reservoir units that are returned to the resource after its completion.

consumes states that the activity is using a number of units that are not returned to the reservoir. This means that the reservoir has reduced capacity after the end of the activity.

provides corresponds to the situation, in which an activity temporarily adds a number of units to the reservoir's capacity. These units are only available during execution of the activity, and are removed when the activity ends.

produces results into permanently adding a number of units to the reservoir. These units are made available, as soon as the producing activity has concluded.

We now describe a very simple example, to clarify the above concepts. In this example, a 60-minute reception requires the presence of five waiters, throughout its duration. Initially, there is only one waiter present, but more waiters may become available in two different ways: a car can deliver five waiters to the reception hall, or back-up staff can provide two extra waiters. The car delivery takes 15 minutes, whereas the back-up staff is readily available at the reception hall, but only for a time-span of 40 minutes. In addition, three waiters need to be transferred to another event. The goal is to pick-up three waiters from the reception hall, as soon as possible.

```

import cotfd;

int horizon = 120;
Scheduler<CP> cp(0,horizon);
Reservoir<CP> waiters(cp,10,1);

Activity<CP> delivery(cp,15);
Activity<CP> pickup(cp,15);
Activity<CP> backupStaff(cp,40);
Activity<CP> reception(cp,60);

minimize<cp>
    pickup.start()
subject to {
    reception.requires(waiters,5);
    delivery.produces(waiters,5);
    pickup.consumes(waiters,3);
    backupStaff.provides(waiters,2);
}
using {
    Activity<CP>[] acts = cp.getActivities();
    label(all (a in cp.getActivities().getRange()) acts[a].end());

    cout << "Initial capacity: 1\t" << endl;
    cout << "delivery: (produces 5)\t" << delivery << endl;
    cout << "pick-up: (consumes 3)\t" << pickup << endl;
    cout << "backupStaff: (provides 2)\t" << backupStaff << endl;
    cout << "reception: (requires 5)\t" << reception << endl;
    cout << endl;
}

```

Statement 16.5: Simple Reservoir Example (reception-cp.co)

Statement 16.5 shows the use of reservoirs to model the problem. Waiters are modeled as a reservoir with maximum capacity 10 and initial capacity 1, while the reception is modeled as an activity that *requires* the reservoir:

```
Reservoir<CP> waiters(cp,10,1);
Activity<CP> reception(cp,60);
```

The waiter delivery and pick-up, and the supply of back-up staff are also modeled as activities, each of which has different behavior. The delivery *produces* 5 waiters after 15 minutes, the back-up supply temporarily *provides* 2 waiters for 40 minutes, and the pick-up *consumes* 3 waiters. These activities are declared in this block of code:

```
Activity<CP> delivery(cp,15);
Activity<CP> pickup(cp,15);
Activity<CP> backupStaff(cp,40);
```

The corresponding interactions with the reservoir, including the reception activity that requires 5 waiters, are posted as constraints in the **subject to** block

```
reception.requires(waiters,5);
delivery.produces(waiters,5);
pickup.consumes(waiters,3);
backupStaff.provides(waiters,2);
```

The objective function to minimize is simply `pickup.start()`. The **using** block does a simple labeling. Here is the resulting schedule:

```
Initial capacity: 1
delivery:  (produces 5)      [0 -- 15 --> 15]
pick-up:   (consumes 3)     [35 -- 15 --> 50]
backupStaff: (provides 2)   [35 -- 40 --> 75]
reception:  (requires 5)    [15 -- 60 --> 75]
```

objective tightened to: 35

As you can see, the reception starts at time 15, once the delivery has been performed. The back-up staff kicks in 20 minutes later, at time 35, at which point 3 waiters are ready to be picked-up.

16.5 State Resources: The Trolley Problem

We are now going to focus on a problem that illustrates the various modeling concepts illustrated so far and also introduces a new kind of resource: the **StateResource**. Each job in this problem corresponds to the production of an item, that needs to be processed on two different machines in a given order. The two machines required by each item are selected from a number of machines that are located in different areas of a factory, and items are moved around using a trolley. In addition to production time, a valid schedule also needs to take into account the time for loading and unloading items from the trolley, and the transportation times.

In more detail, each item has to go through the following steps, before its corresponding job is complete:

1. **Load** the item onto the trolley
2. Transport the item from area **A** to the first machine
3. **Unload** the item from the trolley
4. **Process** the item on the first machine
5. **Load** the item onto the trolley
6. Transport the item from the first to the second machine
7. **Unload** the item from the trolley
8. **Process** the item on the second machine
9. **Load** of the item onto the trolley
10. Transport the item from the second machine to the stock location **S**
11. **Unload** the item from the trolley

The objective is to minimize the time needed to complete all the jobs.

16.5.1 Trolley Problem Data

Statement 16.6 initializes all the data and structures that define the problem. For each job, we must schedule the eight tasks in the enumeration **Tasks**. Transportation is not considered a task and is modeled with state transitions on the trolley. There is only one trolley and five locations: the three available machines, **m1**, **m2** and **m3**, and the beginning and stock areas, **areaA** and **areaS**.

```
enum Tasks      = {loadA,unload1,process1,load1,unload2,process2,load2,unloadS};
enum Locations = {m1,m2,m3,areaA,areaS};
```

Each item's job is characterized by a tuple **jobRecord** containing the two machines visited by the item and the corresponding processing times. There are 6 jobs, stored in the array **jobRecord**.

```
tuple    jobRecord    {Locations machine1; int durations1;
                      Locations machine2; int durations2;}
enum     Jobs         = {j1,j2,j3,j4,j5,j6};
jobRecord job[Jobs]   = [...];
```

Loading (unloading) of item onto (from) the trolley takes 20 time units, and loading/unloading of several items can be done in parallel. We also assume, for now, that the trolley is uncapacitated. The time to move the trolley from location to location is given in the transition matrix **tt**. Finally, the total allowed duration (the horizon) for the whole schedule is 2000 time units.

```
int loadDuration      = 20;
int horizon           = 2000;
int tt[Locations,Locations] = [...];
```

Finally, for each job and task, the location, where the task must be processed, and the corresponding duration are stored in matrices **locations** and **durations**.

```
Locations location[Jobs,Tasks];
int duration[Jobs,Tasks];
```

```

enum Tasks      = {loadA,unload1,process1,load1,unload2,process2,load2,unloadS};
enum Locations = {m1,m2,m3,areaA,areaS};

tuple      jobRecord    {Locations machine1; int durations1;
                        Locations machine2; int durations2;}
enum      Jobs          = {j1,j2,j3,j4,j5,j6};
jobRecord job[Jobs]     = [jobRecord(m1,80,m2,60),
                        jobRecord(m2,120,m3,80),
                        jobRecord(m2,80,m1,60),
                        jobRecord(m1,160,m3,100),
                        jobRecord(m3,180,m2,80),
                        jobRecord(m2,140,m3,60)];

int loadDuration      = 20;
int horizon           = 2000;
int tt[Locations,Locations] = [[ 0, 50, 60, 50, 90 ],
                                [ 50, 0, 60, 90, 50 ],
                                [ 60, 60, 0, 80, 80 ],
                                [ 50, 90, 80, 0, 120 ],
                                [ 90, 50, 80, 120, 0 ]];

Locations location[Jobs,Tasks];
int      duration[Jobs,Tasks];

forall (j in Jobs) { location[j,loadA]      = areaA;
                    location[j,unload1]    = job[j].machine1;
                    location[j,process1]    = job[j].machine1;
                    location[j,load1]       = job[j].machine1;
                    location[j,unload2]     = job[j].machine2;
                    location[j,process2]    = job[j].machine2;
                    location[j,load2]       = job[j].machine2;
                    location[j,unloadS]     = areaS; }

forall (j in Jobs) { duration[j,loadA]      = loadDuration;
                    duration[j,unload1]     = loadDuration;
                    duration[j,process1]     = job[j].durations1;
                    duration[j,load1]       = loadDuration;
                    duration[j,unload2]     = loadDuration;
                    duration[j,process2]    = job[j].durations2;
                    duration[j,load2]       = loadDuration;
                    duration[j,unloadS]     = loadDuration; }

```

Statement 16.6: Data for the Trolley Problem

16.5.2 Modeling the Trolley Problem

The COMET model of the trolley problem is given in Statement 16.7. The model first creates a scheduler object `cp` with the given horizon. It then creates an activity for each task within each job, specifying the activity's duration and the location where it should be performed. It also creates a dummy activity `makespan`, with zero duration, to represent the project's duration.

```
Scheduler<CP>      cp(horizon);
Activity<CP>       act[j in Jobs,t in Tasks](cp,duration[j,t],location[j,t]);
Activity<CP>       makespan(cp,0);
```

The model then defines the resources used in the schedule. It creates a unary resource for each machine, along with a dictionary `machine` that maps machine locations to actual unary resources. It then creates a single `StateResource<CP>` to represent the trolley and the locations it visits. State resources are uncapacitated and can only process an activity, if they are in the state specified by the activity. It also possible to use a transition matrix to specify the transition times necessary for the state resource to move from one state to another. Note that no activity can be processed by the state resource during a state transition. In our model, states correspond to locations of the trolley and transition times to transportation times between location.

```
dict{Locations -> UnaryResource<CP>} machine();
machine{m1}      = UnaryResource<CP>(cp);
machine{m2}      = UnaryResource<CP>(cp);
machine{m3}      = UnaryResource<CP>(cp);
StateResource<CP> trolley(cp,Locations,tt);
```

The next set of constraints enforce that the tasks of each job are performed in the right order, i.e., they define job precedence constraints.

```
forall (j in Jobs, t1 in Tasks, t2 in Tasks : t1 < t2)
  act[j,t1].precedes(act[j,t2]);
```

```

import cotfd;
Scheduler<CP>      cp(horizon);
Activity<CP>       act[j in Jobs,t in Tasks](cp,duration[j,t],location[j,t]);
Activity<CP>       makespan(cp,0);

dict{Locations -> UnaryResource<CP>} machine();
machine{m1}        = UnaryResource<CP>(cp);
machine{m2}        = UnaryResource<CP>(cp);
machine{m3}        = UnaryResource<CP>(cp);
StateResource<CP>  trolley(cp,Locations,tt);

minimize<cp>
  makespan.start()
subject to {
  forall (j in Jobs, t1 in Tasks, t2 in Tasks : t1 < t2)
    act[j,t1].precedes(act[j,t2]);
  forall (j in Jobs) {
    act[j,process1].requires(machine[job[j].machine1]);
    act[j,process2].requires(machine[job[j].machine2]);
  }
  forall (j in Jobs, t in Tasks : t != process1 && t != process2)
    act[j,t].requires(trolley,location[j,t]);
  forall (j in Jobs)
    act[j,unloadS].precedes(makespan);
}
using {
  setTimes(all(j in Jobs,t in Tasks) act[j,t]);
  makespan.scheduleEarly();
  cout << "Makespan: " << makespan.start() << endl;
}

```

Statement 16.7: CP Model for the Uncapacitated Trolley Problem (trolley-cp.co)

The model then states which unary constraint should perform each task:

```
forall (j in Jobs) {  
    act[j,process1].requires(machine{job[j].machine1});  
    act[j,process2].requires(machine{job[j].machine2});  
}
```

It also specifies that the trolley must be in the right place for the whole duration of each load or unload activity:

```
forall (j in Jobs, t in Tasks : t != process1 && t != process2)  
    act[j,t].requires(trolley,location[j,t]);
```

The last group of constraints simply state that the **makespan** dummy activity cannot start before every job has finished.

```
forall (j in Jobs)  
    act[j,unloadS].precedes(makespan);
```

Finally, the search uses the non-deterministic search method **setTimes**, to assign a starting time to each activity, and after that it schedules the **makespan** as early as possible.

```
setTimes(all(j in Jobs,t in Tasks) act[j,t]);  
makespan.scheduleEarly();
```

16.5.3 Modeling Limited Trolley Capacity

We now show how to relax the assumption that the trolley is uncapacitated. In particular, we describe how to model the restriction that the trolley can carry at most three items at any given time. Since state resources don't have capacities, we introduce a discrete resource `trolleyCapacity` of maximum capacity 3, to model the trolley's capacity. We also need to introduce three additional types of tasks, corresponding to an item's stage of transportation:

onTrolleyA1 is the shipping from area A to the first machine

onTrolley12 is the shipping from the first to the second machine of the job

onTrolley2S is the shipping from the second machine to area S

Now, we can define three new `Activity<CP>` objects for each job, one of each task type, to track trolley usage. Essentially, each new activity will require 1 unit from the `trolleyCapacity` discrete resource. Note, though, that its duration will be variable, since it will start at the beginning of a load activity and end at the completion of an unload activity. So the minimum duration of any new activity will be $2 * \text{loadDuration}$. The new variables are stored in an array `tact`.

```
enum          TrolleyTasks = {onTrolleyA1, onTrolley12, onTrolley2S};
Activity<CP>   tact[j in Jobs,t in TrolleyTasks] (cp,2*loadDuration..horizon);
DiscreteResource<CP> trolleyCapacity(cp,3);
```

```

import cotfd;
Scheduler<CP>          cp(horizon);
enum                  TrolleyTasks = {onTrolleyA1, onTrolley12, onTrolley2S};
Activity<CP>          act[j in Jobs,t in Tasks](cp, duration[j,t], location[j,t]);
Activity<CP>          tact[j in Jobs,t in TrolleyTasks](cp, 2*loadDuration..horizon);
Activity<CP>          makespan(cp,0);

dict{Locations -> UnaryResource<CP>} machine();
machine{m1}           = UnaryResource<CP>(cp);
machine{m2}           = UnaryResource<CP>(cp);
machine{m3}           = UnaryResource<CP>(cp);
StateResource<CP>     trolley(cp,Locations,tt);
DiscreteResource<CP>  trolleyCapacity(cp,3);

minimize<cp>
  makespan.start()
subject to {
  forall (j in Jobs, t1 in Tasks, t2 in Tasks : t1 < t2)
    act[j,t1].precedes(act[j,t2]);
  forall (j in Jobs) {
    act[j,process1].requires(machine[job[j].machine1]);
    act[j,process2].requires(machine[job[j].machine2]);
  }
  forall (j in Jobs, t in Tasks : t != process1 && t != process2)
    act[j,t].requires(trolley,location[j,t]);
  forall (j in Jobs)
    act[j,unloadS].precedes(makespan);
  forall (j in Jobs) {
    cp.post(tact[j,onTrolleyA1].start() == act[j,loadA].start());
    cp.post(tact[j,onTrolleyA1].end()   == act[j,unload1].end());
    cp.post(tact[j,onTrolley12].start() == act[j,load1].start());
    cp.post(tact[j,onTrolley12].end()   == act[j,unload2].end());
    cp.post(tact[j,onTrolley2S].start() == act[j,load2].start());
    cp.post(tact[j,onTrolley2S].end()   == act[j,unloadS].end());
  }
  forall (j in Jobs, t in TrolleyTasks)
    tact[j,t].requires(trolleyCapacity,1);
}
using {
  setTimes(all(j in Jobs,t in Tasks) act[j,t]);
  makespan.scheduleEarly(); }

```

Statement 16.8: CP Model for the Trolley Problem with Capacity 3 (trolley-withcapa-cp.co)

The model of Statement 16.8 extends the constraint set of Statement 16.7, to take into account the new activities. The following group of constraints bind the start and end of each new activity to the start and end of the corresponding load and unload operations.

```
forall (j in Jobs) {  
  cp.post(tact[j,onTrolleyA1].start() == act[j,loadA].start());  
  cp.post(tact[j,onTrolleyA1].end() == act[j,unload1].end());  
  cp.post(tact[j,onTrolley12].start() == act[j,load1].start());  
  cp.post(tact[j,onTrolley12].end() == act[j,unload2].end());  
  cp.post(tact[j,onTrolley2S].start() == act[j,load2].start());  
  cp.post(tact[j,onTrolley2S].end() == act[j,unloadS].end());  
}
```

The last set of constraints specify that each transport activity in `tact` requires a single unit from the `trolleyCapacity` discrete resource:

```
forall (j in Jobs, t in TrolleyTasks)  
  tact[j,t].requires(trolleyCapacity,1);
```

Finally, the search used is the same as in the uncapacitated problem, since all new activities are dependant on the ones previously defined. This means that the start, end and duration variables of the new activities are automatically assigned, when the other problem variables become bound.

16.6 Resource Pools

This chapter concludes with a description of pool resources supported by COMET. Resources of this kind are invaluable for applications, in which some activities can be performed by one out of many possible resources. This is a very common situation in industrial production or in transportation. For example, we could have some cargo, that can be transferred either by boat or by truck, at different cost and speed, and we would like to take into account these alternatives when scheduling the transportation operations.

There are pool versions of three kinds of resources: Unary resources, unary sequences, and discrete resources. The following sections will explain the basic interface of the available pool resources and show some illustrative examples.

16.6.1 Unary Resource Pool

A unary resource pool, `UnaryResourcePool<CP>`, consists of a number of `UnaryResource<CP>` objects in a given range. In order to model activities that have one or more options for the unary resource that can process them, we can simply use the `Activity<CP>` method:

```
void requires(UnaryResourcePool<CP>,int [],int [])
```

The two `int` arrays in the arguments correspond to the indices of the unary resources that can process the activity, in the pool's range, and the corresponding durations. Note that the same activity may have different processing times on different unary resources. The most common methods of the unary resource pool are the following:

```
UnaryResourcePool<CP>(Scheduler<CP>,range)
UnaryResource<CP> getUnaryResource(int)
void close()
var<CP>{int} getSelectedResource(Activity<CP>)
void assignAlternatives()
void rank()
```

As mentioned above, one of the constructor arguments specifies the desired range for the unary resources in the pool. The constructor will then allocate one `UnaryResource<CP>` object for every position of the range. The method `getUnaryResource(int i)` can then be used to retrieve the

unary resource of index *i*. In order to allow better control over the unary resource assignment, the class supports a method `getSelectedResource(Activity<CP>)`, that returns a `var<CP>{int}` variable, whose value represents the index of the unary resource assigned to the given activity, among the possible options. These CP variables can only be accessed once the resource pool is closed with the `close` method.

The remaining two methods are used in scheduling the activities that use the pool. There are two ways to do that. One way is to first use the method `assignAlternatives`, to non-deterministically assign resources to activities, and then rank each individual unary resource, using the `rank()` method for `UnaryResource<CP>`. The other way is to simply call the method `rank()` on `UnaryResourcePool<CP>`; this automatically assigns resources to activities.

All this will become more clear, through the example on Statement 16.9, which solves a flexible job-shop scheduling problem. In this problem, we are given a number of activities, that can be performed on different machines, along with the duration of each activity on each machine, and we are looking for the minimum makespan.

The problem's data is first read into a number of ranges, `Jobs`, `Machines` and `Activities`, a two-dimensional array of activity durations on machines, and a set of precedences for activities within the same job. The parsing is omitted here, for simplicity. In order to simplify posting the constraints, we use a large integer value (BIG) for the duration of activities on machines that are not among their options.

Then, the code allocates a scheduler, an array of activities, a dummy makespan activity and a unary resource pool on the `Machine` range read in the data.

```
Scheduler<CP>          cp(horizon);
Activity<CP>          activity[Activities](cp,0..horizon);
Activity<CP>          makespan(cp,0);
UnaryResourcePool<CP> resourcePool(cp,Machines);
```

The allocated unary resources are retrieved into an array `machine` through the `getUnaryResource` method. In addition, an array `selectedMachine` of CP variables is allocated, to represent the assignment of activities to resource indices. The variables are later linked to the corresponding variables of the resource pool.

```
UnaryResource<CP>      machine[m in Machines] = resourcePool.getUnaryResource(m);
var<CP>{int}          selectedMachine[Activities](cp,Machines);
```

```

import          cotfd;
range           Jobs;
range           Machines;
range           Activities;
int             durations[Activities,Machines];
set{Precedence<CP>} precedences();

Scheduler<CP>   cp(horizon);
Activity<CP>    activity[Activities](cp,0..horizon);
Activity<CP>    makespan(cp,0);
UnaryResourcePool<CP> resourcePool(cp,Machines);
UnaryResource<CP> machine[m in Machines] = resourcePool.getUnaryResource(m);
var<CP>{int}     selectedMachine[Activities](cp,Machines);

minimize<cp>
    makespan.start()
subject to {
    forall (a in Activities) {
        set{int} aMachines = filter(m in Machines) (durations[a,m] != BIG);
        activity[a].requires(resourcePool,
                               all (m in aMachines) m,
                               all (m in aMachines) durations[a,m]);
    }
    resourcePool.close();
    forall (a in Activities)
        cp.post(selectedMachine[a] == resourcePool.getSelectedResource(activity[a]));

    forall (p in precedences)
        activity[p._o].precedes(activity[p._d]);
    forall (a in Activities)
        activity[a].precedes(makespan);
}
using {
    resourcePool.assignAlternatives();
    forall (m in Machines) by (machine[m].localSlack(),machine[m].globalSlack())
        machine[m].rank();
    makespan.scheduleEarly();
    cout << endl << all(a in Activities) selectedMachine[a] << endl;
}

```

Statement 16.9: CP Model for Flexible Job Shop Using Resource Pools (`resourcePool-cp.co`)

The **subject to** block first specifies, for each activity, the machines that can perform it and the corresponding durations:

```
set{int} aMachines = filter(m in Machines) (durations[a,m] != BIG);
activity[a].requires(resourcePool,
                    all (m in aMachines) m,
                    all (m in aMachines) durations[a,m]);
```

It then links the `selectedMachine` variables to the corresponding resource pool variables, retrieved using the method `getSelectedResource`, after closing the resource pool:

```
resourcePool.close();
forall (a in Activities)
    cp.post(selectedMachine[a] == resourcePool.getSelectedResource(activity[a]));
```

The rest of the constraints are straightforward, so can focus on the **using** block, which uses the `assignAlternatives` method to assign machines to activities, and then ranks each unary resource separately.

```
resourcePool.assignAlternatives();
forall (m in Machines) by (machine[m].localSlack(),machine[m].globalSlack())
    machine[m].rank();
```

This whole block could be replaced by a single line:

```
resourcePool.rank();
```

16.6.2 Unary Sequence Pool

Unary Sequence Pools are very similar to Unary Resource Pools, as can be seen by looking at the API of `UnarySequencePool<CP>`, a part of which is shown next:

```
UnarySequencePool<CP>(Scheduler<CP>,range)
UnarySequence<CP> getUnarySequence(int)
void close()
var<CP>{int} getSelectedResource(Activity<CP>)
void assignAlternatives()
void rank()

Activity<CP> getSource(int)
Activity<CP> getSink(int)
Activity<CP> getSuccessor(Activity<CP>)
Activity<CP> getPredecessor(Activity<CP>)
```

The corresponding `requires` method for an activity to use a unary sequence pool is almost identical:

```
void requires(UnarySequencePool<CP>,int [],int [])
```

Not surprisingly, the additional functionality of `UnarySequence<CP>` also extends to unary sequence pools. This means that it is still possible to retrieve the predecessor or the successor of an activity in the sequence it has been assigned to. Of course, given that the pool consists of multiple unary sequences, there a different source and sink for each one of them. This explains the integer argument in the `getSource` and `getSink` methods.

It takes only minor modifications to the code of Statement 16.9, to make it work with unary sequence pools, as can be verified by comparing the statement above with Statement 16.10, that uses unary sequences. The only significant difference is the use `sequenceForward` instead of `rank` on individual sequences.

```

import          cotfd;
range           Jobs;
range           Machines;
range           Activities;
int             durations[Activities,Machines];
set{Precedence<CP>} precedences();

Scheduler<CP>   cp(horizon);
Activity<CP>    activity[Activities](cp,0..horizon);
Activity<CP>    makespan(cp,0);
UnarySequencePool<CP> resourcePool(cp,Machines);
UnarySequence<CP> machine[m in Machines] = resourcePool.getUnarySequence(m);
var<CP>{int}     selectedMachine[Activities](cp,Machines);

minimize<cp>
    makespan.start()
subject to {
    forall (a in Activities) {
        set{int} aMachines = filter(m in Machines) (durations[a,m] != BIG);
        activity[a].requires(resourcePool,
                               all (m in aMachines) m,
                               all (m in aMachines) durations[a,m]);
    }
    resourcePool.close();
    forall (a in Activities)
        cp.post(selectedMachine[a] == resourcePool.getSelectedResource(activity[a]));

    forall (p in precedences)
        activity[p._o].precedes(activity[p._d]);
    forall (a in Activities)
        activity[a].precedes(makespan);
}
using {
    resourcePool.assignAlternatives();
    forall (m in Machines) by (machine[m].localSlack(),machine[m].globalSlack())
        machine[m].sequenceForward();
    makespan.scheduleEarly();
    cout << endl << all(a in Activities) selectedMachine[a] << endl;
}

```

Statement 16.10: CP Model for Flexible Job Shop Using Sequence Pools ([sequencePool-cp.co](#))

16.6.3 Discrete Resource Pool

The mechanism for the discrete resource pool, `DiscreteResourcePool<CP>` is similar to the other resource pools seen so far. In fact, the API is a little simpler; here are its most important methods

```
DiscreteResourcePool<CP>(Scheduler<CP>,range,int [])  
void close()  
var<CP>{int} getSelectedResource(Activity<CP>)  
void assignAlternatives()
```

There are some slight differences, though, compared to unary resource and unary sequence pools. For example, the constructor of a discrete resource pool must specify the capacity of each constituent discrete resource, in addition to the range. Another difference is that there is no direct access to the actual discrete resource assigned to an activity. There is only access to its index within the discrete resource pool. Finally, the corresponding `requires` method does not include duration information. It can only specify a discrete resource pool and the number of units required from it:

```
void requires(DiscreteResourcePool<CP> d,int c)
```


Chapter 17

Constraint Programming and Concurrency

In this chapter we describe the specific support for parallelizing constraint programming models written in `COMET`. Parallelization of a constraint program boils down to the exploration of the search tree with multiple threads. Naturally, a key objective is to carry out a work-efficient exploration, when compared to the sequential version of the same search. Namely, the total amount of work expended in the parallel case should be as close as possible to the sequential case and the tasks should be evenly spread out among the processors, to yield an ideal linear speedup.

This objective, however, cannot always be attained with optimization problems. Indeed, it is sometimes possible that the parallel version of a given search strategy ends up exploring subtrees that the sequential program might be able to prune (or fathom in the case of an optimization problem, thanks to the availability of a better bound). Similarly, optimization problems might display super-linear speedups, when excellent bounds are discovered early on by some threads enabling the others to discard entire subtrees that should otherwise be explored.

In any case, a parallel model is a sophisticated piece of software which, without support, would require substantial efforts for its production. `COMET` rises to the challenge and provides the necessary abstractions and tools to seamlessly turn a sequential constraint programming model into its parallel cousin, regardless of which search strategy and heuristics it implements. The remainder of this chapter describes the necessary primitives and illustrates how to parallelize a constraint program in `COMET`

17.1 Parallel Solving

From a modeling standpoint, the changes necessary to parallelize a sequential model are minimal. Essentially, one simply needs to embed the model statement in a `parall` loop to explore the search tree in parallel. Naturally, the model is not solved in its entirety by each thread contributing to the iterations of the parallel loop. Instead, the threads *share* a data-structure that holds the representation of the tree to explore and contribute to its examination.

The setup of this shared data structure consists in creating a shared problem pool and switching from a `Solver<CP>` to a `ParallelSolver<CP>`. For instance, consider the basic COMET program setup

```
Solver<CP> m();
minimize<m>
    <expression>
subject to {
    ... // state the constraints
}
using {
    ... // explore the search tree
}
```

A parallel version can be easily obtained with

```
SearchProblemPool pool(System.getNCPUS());
parall<pool> (p in 1..pool.getSize()) {
    ParallelSolver<CP> m();
    minimize<m>
        <expression>
    subject to {
        ... // state the constraints
    }
    using {
        ... // explore the search tree
    }
}
pool.close();
```

In the above, `pool` plays a dual role: it embodies the pool of threads that are going to contribute to the exploration of the search tree, and also serves as a repository of sub-problems (sub-trees) that are both generated and explored by the search procedure. The `parall` instruction iterates as many times as there are threads in the pool (thus guaranteeing that every thread in the pool will be working) and each iteration sets up a `ParallelSolver<CP>` and a model, and starts the search as usual. The very last line closes the thread pool, as customary, to reclaim the resources, i.e., the threads.

17.2 Parallel Graph Coloring

The most effective way to demonstrate the technique is, perhaps, through an example. This section illustrates the technique on a simple graph coloring model.

The Model We use a rather straight-forward model for the graph coloring problem, that uses an array of decision variables representing the color assigned to each vertex of the graph. The objective is simply to minimize the number of distinct colors used, or, alternatively, the “largest” color used. An auxiliary variable x represents the largest used color. A dis-equality constraint is stated for each edge of the graph to impose a valid coloring (no two adjacent vertices can have the same color). The model is shown below

```
range V      = ...; // the range of vertex identifiers
range C      = ...; // the range of colors
bool  adj[V,V] = ...; // the adjacency matrix

Solver<CP>    m();
var<CP>{int}  c[V](m,C); // the color of each vertex
var<CP>{int}  x(m,C);    // the largest color

minimize<m>
  x
subject to {
  forall (i in V)
    m.post(c[i] <= x);
  forall (i in V, j in V : i < j && adj[i,j])
    m.post(c[i] != c[j]);
}
```

The Search An effective search procedure simply follows a greedy heuristic, that colors first the vertices with the fewest available colors, breaking ties in favor of vertices with the highest degree. The selected vertex is then assigned one of the *used* colors or, as a last resort, a brand new *unused* color.

```
...
using {
  int maxc = max(0, max(v in V : c[v].bound()) c[v].getMin());
  forall (v in V : !c[v].bound()) by (c[v].getSize(), -deg[v]) {
    tryall<m> (k in C : c[v].memberOf(k) && k <= maxc + 1) {
      m.label(c[v], k);
      maxc = max(maxc, k);
    }
    onFailure
      m.diff(c[v], k);
  }
  m.label(x, x.getMin()); // bind the largest color to its lower bound.
}
```

The first line computes `maxc` as the largest assigned color (or 0 if no vertices are assigned yet). The `by` clause of the `forall` defines a lexicographic dynamic variable ordering (smallest domain first, and largest degree second, in case of ties). The `tryall` instruction considers all colors up to `maxc+1`.

Notice how `tryall` uses the `onFailure` clause to remove the failed value from the variable's domain. While not absolutely necessary, this is always safe to state and, in a parallel program, it guarantees, that the threads will never explore the same sub-problem more than once.

The Parallel Version The parallel program simply combines the two fragments discussed above within the template introduced in the beginning of the chapter. The crux of the resulting model is shown in Statement [17.1](#).

```

range V      = ...; // the range of vertex identifiers
range C      = ...; // the range of colors
bool  adj[V,V] = ...; // the adjacency matrix

SearchProblemPool pool(System.getNCPUS());

parall<pool> (p in 1..pool.getSize()) {
  ParallelSolver<CP> m();
  var<CP>{int}      c[V](m,C); // the color of each vertex
  var<CP>{int}      x(m,C);    // the largest color
  minimize<m>
    x
  subject to {
    forall (i in V)
      m.post(c[i] <= x);
    forall (i in V, j in V : i < j && adj[i,j])
      m.post(c[i] != c[j]);
  }
  using {
    int maxc = max(0, max(v in V : c[v].bound()) c[v].getMin());
    forall (v in V : !c[v].bound()) by (c[v].getSize(),-deg[v]) {
      tryall<m> (k in C : c[v].memberOf(k) && k <= maxc + 1) {
        m.label(c[v],k);
        maxc = max(maxc,k);
      }
      onFailure
        m.diff(c[v],k);
    }
    m.label(x,x.getMin()); // bind the largest color to its lower bound.
  }
}

pool.close();

```

Statement 17.1: Parallel COMET Model for Graph Coloring

||| CONSTRAINT-BASED LOCAL SEARCH

Local Search is a very popular heuristic method for solving hard optimization problems. It is not an exact method, in the sense that it does not offer any guarantee of reaching an optimal solution or proving optimality. Nevertheless, it often gives solutions of very high quality in problems where exact methods cannot be applied.

The core idea of local search is iterative improvement: At each point, the solver tries to apply an improving *move* chosen in the *neighborhood* of the current *solution*. However, in most applications of local search the search space contains a big number of local optima, which means that, if one always chooses improving moves, the search will soon get stuck into a local optimum.

In order to circumvent this, a number of hyper-heuristics (also known as *meta-heuristics*) have been developed, that allow the search to escape from local optima, avoid cycling between solutions and explore several parts of the solution landscape. Some of the most widely used heuristics of this kind include *Tabu Search*, *Simulated Annealing* and *Variable Neighborhood Search*.

We highlight these ideas along with showing how to use COMET's Constraint-Based Local Search (CBLS) component, in order to develop local search solutions to different problems. We start with some simple toy-problems, illustrating the main concepts and, after a more in-depth coverage of the related COMET's constructs, we show some more advanced application. Readers further interested in Constraint-Based Local Search can refer to [2] for a more extensive coverage, that goes beyond the scope of this user manual.

Chapter 18

Getting Started with CBLS

The goal of this chapter is to introduce features of COMET that are most useful for Constraint-Based Local Search, through some simple examples. Later chapters provide a more in-depth coverage of the platform's features and more involved applications. Ideally, users should be able to develop their own simple models and applications using the knowledge of this chapter and the reference manual.

18.1 The Queens Problem

In this section, we will see:

- a brief introduction to constraints and constraint systems in COMET's local search
- how to write a generic max-conflict / min-conflict search procedure

The Problem The n -queens problem consists in placing n queens on a $n \times n$ board, so that no two queens lie in the same row, column, or diagonal. The local search algorithm presented here is a min-conflict heuristic starting from an assignment to default values. In each iteration, the algorithm chooses the queen violating the most constraints and moves it to a column minimizing its violations. This is repeated until a valid solution (i.e., one with no violations) is found.

A COMET program for the problem is presented in statement 18.1. The program consists of two easily distinguishable parts: a model (lines 1–13) and a search procedure (lines 15–21).

The Model Since in a feasible solution exactly one queen has to be placed in each column, a natural choice is to associate a queen with each column and to search for an assignment of rows to queens, in which no two queens are placed in the same row or diagonal. As a consequence, the model defines an array of decision variables `queen[c]`, with c in the range $1 \dots n$ of columns and `queen[c]` equal to the row of the queen placed on column c . The problem then consists of searching for an assignment such that the constraints

$$\begin{aligned} \text{queen}[i] &\neq \text{queen}[j] \\ \text{queen}[i] - i &\neq \text{queen}[j] - j \\ \text{queen}[i] + i &\neq \text{queen}[j] + j \end{aligned}$$

hold for all $1 \leq i < j \leq n$. Of course, this is equivalent to three `alldifferent` constraints:

```
alldifferent(queen[1], ..., queen[n])
alldifferent(queen[1]-1, ..., queen[n]-n)
alldifferent(queen[1]+1, ..., queen[n]+n)
```

leading into a linear space model. We now give a more detailed description of the model presented in lines 1–13 of statement 18.1.

```

1 import cotls;
2 int    n = 16;
3 range Size = 1..n;
4 UniformDistribution distr(Size);
5
6 Solver<LS> m();
7 var{int} queen[Size](m,Size) := distr.get();
8 ConstraintSystem<LS> S(m);
9
10 S.post(alldifferent(queen));
11 S.post(alldifferent(all(i in Size)(queen[i] + i)));
12 S.post(alldifferent(all(i in Size)(queen[i] - i)));
13 m.close();
14
15 int it = 0;
16 while (S.violations() > 0 && it < 50 * n) {
17     selectMax (q in Size) (S.violations(queen[q]))
18     selectMin (v in Size) (S.getAssignDelta(queen[q],v))
19     queen[q] := v;
20     it++;
21 }

```

Statement 18.1: CBLS Model for the Queens Problem

The first line imports the local-search module of COMET. Then the model defines a range `Size` equal to `1...n` and a uniform distribution `distr` in this range, with the command

```
UniformDistribution distr(Size);
```

The distribution is used to initialize the queen positions to pseudo-random values. The following instruction defines a local solver `m`.

```
Solver<LS> m();
```

Informally speaking, local solvers are containers that store objects of the model, such as variables and constraints, and make possible an efficient flow of information between components during the search. For example, if a variable changes its value and, as a result, a constraint gets violated, the solver is responsible for updating this information.

Once allocated, the solver defines an array of incremental variables `queen[q]`, as follows:

```
var{int} queen[Size](m,Size) := distr.get();
```

Each variable `queen[c]` represents the row in which the queen of column `c` is located. All these variables are associated with the local server `m`, take their values in the range `Size`, and are initialized with values drawn from the random distribution.

Incremental variables are a central concept in COMET, since they are used in most objects of the language. They are typically associated with a domain representing their set of possible values, and every change in their value typically initiates a propagation step that updates any other object affected.

The next lines are very important and specify the problem constraints in a high-level way. They first declare a local-search constraint system **S** associated with the solver **m**:

```
ConstraintSystem<LS> S(m);
```

and then add three **alldifferent** constraints to **S** with the instructions (lines 10–12):

```
S.post(alldifferent(queen));  
S.post(alldifferent(all(i in Size)(queen[i]+i))));  
S.post(alldifferent(all(i in Size)(queen[i]-i))));
```

specifying that the queens cannot be in the same column or diagonal. Finally, line 13 closes the model:

```
m.close();
```

In practice, the **close** command instructs COMET to build a dependency graph, that joins variables and constraints and allows for more efficient propagation of information during the search phase.

Constraints in Local Search It's useful at this point to give a brief description of constraints in COMET's CBLIS module. Constraints in this module are objects of classes implementing the `Constraint<LS>` interface, a part of which is shown in the following block:

```
interface Constraint<LS> {
    var{int}[] getVariables()
    var{bool} isTrue()
    var{int} violations()
    var{int} violations(var{int})
    int      getAssignDelta(var{int},int)
    int      getSwapDelta(var{int},var{int})
    ...
}
```

The method `getVariables()` gives access to the incremental variables included in the constraint, `isTrue()` indicates whether the constraint is satisfied, `violations()` returns the total number of violations that prevent the constraint from being satisfied, while `violations(x)` gives the number of violations corresponding to incremental variable `x`. The remaining methods are very useful for local search algorithms, as they evaluate the effect of assignments and swaps on the total number of violations. The method `getAssignDelta(x,v)` returns the increase in violations by assigning variable `x` to value `v`, while method `getSwapDelta(x1,x2)` returns the increase in violations, if variables `x1` and `x2` swap values.

A constraint system is a composite type of constraint that is formed as a conjunction of individual constraints, added to it using the `post` method. The number of violations of a constraint system is equal to the sum of violations of the individual constraints posted into it. Since constraint systems are constraints themselves, it suffices to query the `getViolations()` method of the constraint system, in order to get its violations. This provides a transparent way of adding constraints into the model, without having to change the search component.

Going back to the queen problem's model, it's interesting to focus on the constraint

```
S.post(alldifferent(all(i in Size)(queen[i] + i)));
```

The main idea is to use the aggregator `all` to create an array of symbolic expressions

```
[queen[1]+1,...,queen[n]+n]
```

and then post an `alldifferent` constraint on the array of expressions. We'll talk more about COMET expressions in later chapters, but, in essence, they can be seen as a mechanism to increase the

expressivity of the language. In the queens example, without expressions one would have to define a new set of variables $d[1], \dots, d[n]$, add constraints to ensure that $d[i] = \text{queen}[i] + i$, and finally add an `alldifferent(d)` constraint; expressions allow the user to define the model more concisely and with greater clarity.

The Search Procedure The search procedure for the queens problem, shown in lines 15–21, implements a simple max-conflict/min-conflict heuristic, that consists of three main steps:

- choose the queen q with the most violations (line 17)
- choose the row v for queen q that minimizes the total number of violations (line 18)
- assign queen q to row v (line 19)

These steps are iterated until a solution is found or the number of iterations exceeds a given threshold:

```
while (S.violations() > 0 && it < 50 * n)
  selectMax (q in Size)(S.violations(queen[q]))
  selectMin (v in Size) (S.getAssignDelta(queen[q],v))
  queen[q] := v;
```

In the queens problem, the number of violations of an individual queen q is equal to the number of constraints it violates. For examples, if queen q is at conflict with at least one other queen in the same row, this counts as one violation. If, in addition to that, there is at least one queen on the same upward diagonal, this counts as two violations for queen q . In general, individual queens can have between 0 and 3 violations.

The total number of violations for the constraint system is the sum of violations of the three `alldifferent` constraints. For an `alldifferent` constraint, the number of violations is calculated based on the different values taken by its variables. If a value v is taken by $k > 1$ variables, this accounts for $k - 1$ violations. Then, the sum of violations from all different values gives the constraint's total number of violations. Note that, in general, the total number of violations of a constraint is *not* necessarily equal to the total number of violations of its individual variables.

In the search procedure we see an instance of two randomized selectors, `selectMax` to select the most violated queen and `selectMin` to select the row minimizing the violations. In both cases, ties are broken randomly. In both selections, the procedure takes advantage of the fact that the constraint system S is itself a constraint. When querying for the number of violations of `queen[q]` it uses the expression:

```
S.violations(queen[q]);
```

When assessing the impact of moving queen q to row v , it uses the expression:

```
S.getAssignDelta(queen[q],v);
```

Note that the language's architecture allows these queries to be performed in constant time. Finally, the assignment

```
queen[q] := v;
```

initiates a propagation step that automatically updates the violations of all affected constraints. Note that the model and the search are clearly separated in the program's statement, making it easy to modify one without affecting the other.

Generic Max-Min Conflict Search The language offers the flexibility to generalize search procedures, developed for particular applications, into generic search procedures, also applicable to other related problems. An illustration of this is the generic Max-Conflict/Min-Conflict search procedure shown on Statement 18.2. Using this generic procedure, the search component of the queens program can be replaced by a single line:

```
conflictSearch(S,50*n);
```

The generic implementation closely follows the structure of the queens program. The key addition is the line that retrieves the constraint's variables using the method `getVariables`:

```
var{int}[] x = c.getVariables();
```

Once the variables are available, the next line retrieves the range of the variable array:

```
range Size = x.getRange();
```

The only other addition is the command used to retrieve the domain of variable $x[i]$ (in our case this is equal to `Size`).

```
x[i].getDomain()
```

```
function void conflictSearch (Constraint<LS> c, int itLimit) {
    int it = 0;
    var{int}[] x = c.getVariables();
    range Size = x.getRange();
    while (!c.isTrue() && it < itLimit) {
        selectMax (i in Size) (c.violations(x[i]))
        selectMin (v in x[i].getDomain()) (c.getAssignDelta(x[i],v))
        x[i] := v;
        it = it + 1;
    }
}
```

Statement 18.2: Generic Max-Conflict/Min-Conflict Search Procedure

18.2 Magic Squares

This section illustrates:

- an example of a model using numerical constraints
- a generic tabu-search procedure

The Problem A magic square of size n is a placement of the numbers 1 through n^2 in an $n \times n$ square, so that all rows, columns and main diagonals sum to the same number T . Since the sum of all numbers on the square is $n^2(n^2+1)/2$ and each row, column and main diagonal has n numbers, it follows that $T = n(n^2+1)/2$. A COMET program for this simple problem is shown in Statement 18.3, which once again illustrates COMET's modularity: the model component is clearly separated from the search component.

The Model The model first defines the size of the problem, including the value for T , and also declares a random permutation of the numbers in range $1 \dots n^2$, with the line

```
RandomPermutation perm(1..n^2);
```

One can obtain individual values of the permutation by calling `perm.get()` n^2 times.

Then it defines a local search solver `m` and specifies the decision variables as a two-dimensional matrix `magic`. For every i, j , `magic[i,j]` holds the value of the magic square in row i and column j . These values are initialized using the random permutation and their domain is the range $\{1, \dots, n^2\}$:

```
var{int} magic[i in Size,j in Size](m,1..n^2) := perm.get();
```

This ensures that, in the initial state, all positions of the magic square have different values. Since the search procedure used is based on swaps between places of the square, this means that we don't need to explicitly state the condition, that each value in $1, \dots, n^2$ has to be placed in a different position of the square.

```

import cotls;
int n = 20;
range Size = 1..n;
int T = (n * (n^2 + 1))/2;
RandomPermutation perm(1..n^2);

Solver<LS> m();
var{int} magic[i in Size, j in Size](m, 1..n^2) := perm.get();
ConstraintSystem<LS> S(m);

forall (k in Size) {
    S.post(sum(i in Size) magic[k,i] == T);
    S.post(sum(i in Size) magic[i,k] == T);
}
S.post(sum(i in Size) magic[i,i] == T);
S.post(sum(i in Size) magic[i,n-i+1] == T);
m.close();

int tabuLength = 10;
int tabu[Size,Size] = 0;
int it = 0;

while (S.violations() > 0) {
    selectMin (i in Size, j in Size : S.violations(magic[i,j]) > 0 && tabu[i,j] <= it,
               k in Size, l in Size : tabu[k,l] <= it && (k != i || l != j))
        (S.getSwapDelta(magic[i,j],magic[k,l])) {
        magic[i,j] := magic[k,l];
        tabu[i,j] = it + tabuLength;
        tabu[k,l] = it + tabuLength;
    }
    it++;
}

```

Statement 18.3: CBLS Model for the Magic Square Problem

Once the decision variables are defined, the model defines a constraint system, and posts the problems constraints to it. In this problem, we'll see a first instance of numerical constraints. COMET supports both combinatorial and numerical constraints for local search, and constraint systems make it transparent to combine different types of constraints. For the magic square problem, the row and column constraints are stated in the following lines:

```
forall (k in Size) {  
    S.post(sum(i in Size) magic[k,i] == T);  
    S.post(sum(i in Size) magic[i,k] == T);  
}
```

The main diagonal constraints are stated in a similar fashion. Notice how aggregators (**sum** in our case) make it easier to state the problem in a declarative fashion.

```
S.post(sum(i in Size) magic[i,i] == T);  
S.post(sum(i in Size) magic[i,n-i+1] == T);
```

The Search Procedure Before describing the search procedure, it's useful to explain how violations are calculated for numerical constraints. For a constraint of the form:

$$L == R$$

the violations of the constraint is equal to **abs(L-R)**. The violations of a variable in the constraint is equal to the total number of violations of the constraint. We stress again here that the total violations of a constraint is not necessary equal to the sum of violations of the constituent variables.

The core of the search procedure is a greedy local search that minimizes the number of violations by using local moves that swap two tiles in the square. The move performed at every step is determined by the selector

```
selectMin (i in Size, j in Size : S.violations(magic[i,j]) > 0 && tabu[i,j] <= it,
          k in Size, l in Size : tabu[k,l] <= it && (k != i || l != j))
  (S.getSwapDelta(magic[i,j],magic[k,l]))
  magic[i,j] := magic[k,l];
```

which goes through all pairs of positions (i, j) and (k, l) with a violation in position (i, j) , and selects the pair that will produce the highest reduction in violations, when swapping the corresponding decision variables. The main block of the selector performs this swap.

The search uses the method `getSwapDelta(x,y)` of the constraint interface, which gives the change in the number of violations by swapping the values of incremental variables x and y . Note, also, the bidirectional incremental variable assignment operator `:=` which swaps the values of variables `magic[i,j]` and `magic[k,l]` in a single step.

This greedy search is extended into a tabu search procedure by incorporating a tabu list of fixed length `tabuLength`, which limits the options for the selector `selectMin` only to places of the square that have not been swapped in the last `tabuLength` selection steps. The tabu list is implemented using an integer `it` to denote the current iteration and a two-dimensional matrix `tabu` to store the earliest iteration `it` for which `magic[i,j]` will not be tabu.

```
int tabuLength = 10;
int tabu[Size,Size] = 0;
int it = 0;
```

Checking if variable `magic[i,j]` is not tabu is equivalent to checking whether

```
tabu[i,j] <= it
```

When `magic[i,j]` is selected for a swap, the `tabu` matrix is updated, so that the variable will be tabu for the next `tabuLength` steps, using the instruction

```
tabu[i,j] := it + tabuLength;
```

After including the tabu list to the original greedy search, we get the following tabu search selection:

```
selectMin (i in Size, j in Size : S.violations(magic[i,j]) > 0 && tabu[i,j] <= it,
          k in Size, l in Size : tabu[k,l] <= it && (k != i || l != j))
  (S.getSwapDelta(magic[i,j],magic[k,l])) {
  magic[i,j] := magic[k,l];
  tabu[i,j] = it + tabuLength;
  tabu[k,l] = it + tabuLength;
}
```

Generic Tabu Search Once again the modular nature of COMET allows to produce a generic swap-based tabu search procedure, by making only minimal changes to the procedure used for the magic square problem. The only additional methods necessary are (as in the case of the generic max-conflict/min-conflict search) the methods `getVariables()` and `getRange()`. This generic tabu search will work for any neighborhood, in which local moves are swaps, and is shown in statement [18.4](#). Using the generic tabu search, one can simply replace the search component of the magic squares with the single function call:

```
tabuSearch(S,10);
```

```
function void tabuSearch(Constraint<LS> c, int tabuLength) {
    var{int}[] x = c.getVariables();
    range      X = x.getRange();
    int        tabu[X] = 0;
    int        it = 0;

    while (c.violations() > 0) {
        selectMin (i in X : c.violations(x[i]) > 0 && tabu[i] <= it,
                  j in X: tabu[j] <= it && i != j)
            (c.getSwapDelta(x[i],x[j])) {
            x[i] := x[j];
            tabu[i] = it + tabuLength;
            tabu[j] = it + tabuLength;
        }
        it++;
    }
}
```

Statement 18.4: Generic Swap-Based Tabu Search

18.3 Send More Money

This section shows:

- how combinatorial and numerical constraints can easily be combined in CBLS models
- how to write a generic, constraint-driven tabu search procedure
- a simple application of dictionaries

The Problem The crypto-arithmetic “send more money” puzzle entails assigning different decimal digits to the letters in the set $\{S, E, N, D, M, O, R, Y\}$, so that the following addition is correct:

$$\begin{array}{rcccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

The COMET code to solve this problem is shown on Statement 18.5, and uses a generic, constraint-directed local search procedure, which we’ll explain shortly. This solution illustrates how combinatorial and numerical constraints can compose naturally in COMET’s CBLS module.

The Model The numerical constraints associated with each column are modeled using a carry representation, in which $R_i \in \{0, 1\}$ ($i = 1, \dots, 4$) is the carry resulting from the addition of column i . In other words, we use the representation:

$$\begin{array}{rcccc} & R_4 & R_3 & R_2 & R_1 \\ & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

which leads to constraints of the form: $D + E == Y + 10 * R1$.

The program starts by defining an enumerated type `Letters` and a range `Digits`:

```
enum Letters = {S, E, N, D, M, O, R, Y};
range Digits = 0..9;
```

```

import cotls;
Solver<LS> m();
enum Letters = {S, E, N, D, M, O, R, Y};
range Digits = 0..9;
UniformDistribution distr(Digits);

var{int} d[Letters] (m,Digits) := distr.get();
var{int} r[1..4] (m,0..1)      := 1;
ConstraintSystem<LS> Sys(m);
Sys.post(alldifferent(d));
Sys.post(d[S] != 0);
Sys.post(d[M] != 0);
Sys.post(r[4] == d[M]);
Sys.post(r[3] + d[S] + d[M] == d[0] + 10 * r[4]);
Sys.post(r[2] + d[E] + d[0] == d[N] + 10 * r[3]);
Sys.post(r[1] + d[N] + d[R] == d[E] + 10 * r[2]);
Sys.post(d[D] + d[E] == d[Y] + 10 * r[1]);
m.close();

constraintDirectedSearch(Sys,4);

```

Statement 18.5: CBLS Model for the “Send More Money” Puzzle

The decision variables consist of the array **d** for the letter assignments, indexed by **Letters** and initialized by a uniform distribution on **Digits**, and the array **r** for the carries, indexed by **Digits**.

```
var{int} d[Letters](m,Digits) := distr.get();
var{int} r[1..4](m,0..1)      := 1;
```

Then the model creates a constraint system, and posts a combinatorial constraint and two kinds of numerical constraints. The combinatorial constraint is an **alldifferent** on the letter assignment:

```
Sys.post(alldifferent(d));
```

The first kind of numerical constraint is a simple dis-equation, enforcing that the starting digits **S** and **M** are non-zero.

```
Sys.post(d[S] != 0);
Sys.post(d[M] != 0);
```

A dis-equation constraint has 1 violation, if not satisfied, and 0 otherwise. Each variable in the constraint has 1 violation, if the constraint is not satisfied, and 0 otherwise. The second type of numerical constraints is a group of equation constraints ensuring that the overall addition is correct:

```
Sys.post(r[4] == d[M]);
Sys.post(r[3] + d[S] + d[M] == d[0] + 10 * r[4]);
Sys.post(r[2] + d[E] + d[0] == d[N] + 10 * r[3]);
Sys.post(r[1] + d[N] + d[R] == d[E] + 10 * r[2]);
Sys.post(d[D] + d[E] == d[Y] + 10 * r[1]);
```

The different kinds of constraints are combined in the constraint system without any problem. As in previous models, the total violations of the constraint system is equal to the sum of violations of the constraints posted to it. Similarly, for each variable of the constraint system, its violations equal the sum of violations of the variable over all the constraints that it is part of.

Constraint-Directed Search In the queens and the magic square problems, the search procedures focused on the variables, choosing which variable to update based on the variable's violations. Constraint-Directed Search follows an orthogonal approach, by focusing instead on the constraints. Essentially, at every step, the search looks for a violated constraint and tries to update its variables in a way that will reduce the constraint's violations. This is done in an attempt to make some local progress, since at least one constraint is reducing its violations. The search is also using a tabu list on the constraint system's variables, in order to avoid cycling over the same variables.

The complete implementation is presented in Statement 18.6. The procedure first retrieves the constraint system's variables into an array `y`

```
var{int}[] y = S.getVariables();
```

and then defines the tabu list as a dictionary from incremental variables to integer values. The tabu list holds the earliest step, at which the incremental variable will not be tabu. The dictionary entries for all variables are then initialized to 0.

```
dict{var{int} -> int} tabu();  
forall (i in y.getRange())  
    tabu{y[i]} = 0;
```

A dictionary is necessary, rather than an array indexed on the range of variables. This is because the main loop keeps retrieving variables from different individual constraints, and variables are not stored in the same order in all the constraints they appear.

The main body of the search starts by retrieving an array `violation` containing the violations of each constraint in the constraint system.

```
var{int}[] violation = S.getCstrViolations();
```

```

function void constraintDirectedSearch(ConstraintSystem<LS> S, int tabuLength) {
    var{int}[] y = S.getVariables();
    dict{var{int} -> int} tabu();
    forall (i in y.getRange())
        tabu{y[i]} = 0;

    int it = 0;
    var{int}[] violation = S.getCstrViolations();
    while (S.violations() > 0) {
        select (j in violation.getRange() : violation[j] > 0) {
            Constraint<LS> c = S.getConstraint(j);
            var{int}[] x = c.getVariables();
            selectMin (i in x.getRange(), d in x[i].getDomain() :
                tabu{x[i]} <= it && c.getAssignDelta(x[i],d) < 0)
                (S.getAssignDelta(x[i],d)) {
                x[i] := d;
                tabu{x[i]} = it + tabuLength;
            }
        }
        it++;
    }
}

```

Statement 18.6: Generic Constraint-Directed Search

The search keeps iterating, until all constraints are satisfied. Each iteration starts by retrieving a violated constraint `c` and its variable array `x[]`:

```
select (j in violation.getRange() : violation[j] > 0) {
  Constraint<LS> c = S.getConstraint(j);
  var{int}[] x = c.getVariables();
  ...
}
```

It then selects a non-tabu variable `x[i]` of constraint `c` and a value `d` in `x[i]`'s domain that would decrease the constraint's violations, if assigned to `x[i]`. Among all different variable-value pairs, we choose the one that results into the maximum decrease in the constraint system's total violations.

```
selectMin (i in x.getRange(), d in x[i].getDomain() :
  tabu{x[i]} <= it && c.getAssignDelta(x[i],d) < 0)
  (S.getAssignDelta(x[i],d)) {...}
```

The main block of the selector assigns variable `x[i]` to value `d` and updates the tabu list for `x[i]`.

```
x[i] := d;
tabu{x[i]} = it + tabuLength;
```

The whole procedure is completely generic and would work for any kind of constraint system, regardless of the nature of its constraints. One could, for example, solve the magic square problem in this way by replacing the call to the generic tabu search by a call

```
constraintDirectedSearch(S,10);
```

The only consideration here though is that, since constraint-directed search is based on assignment of variables, one has to post an additional `alldifferent` to the magic square model:

```
S.post(alldifferent(all(i in Size,j in Size) magic[i,j]));
```

18.4 Magic Series

In this section:

- we show a simple example of a cardinality combinator constraint in COMET
- we show how to use COMET *events* for increased modularity in local search models

The Problem The magic series problem is a self-referential puzzle that can be defined as follows: An arithmetic series $S_n = (s_0, \dots, s_{n-1})$ is called *magic*, if, for every i , s_i indicates the number of occurrences of i in the series. For example, $S_4 = (2, 0, 2, 0)$ is magic, since $s_0 = 2$ and this equals the number of 0s in the series, $s_2 = 2$, which again indicates the number of 2s, and, finally, $s_1 = s_3 = 0$, which corresponds to the fact that no 1s or 3s appear in the series. A longer magic series is, for instance, $S_8 = (4, 2, 1, 0, 1, 0, 0, 0)$.

The Model The complete COMET code for the problem is shown on Statement 18.7. We first describe the model, which appears in the beginning of the statement (up until the `close` instruction). After declaring the series length and the associated range, the model declares a decision variable array `s`, with `s[i]` representing the term s_i of the series S_n . All variables are initialized to 0.

The most interesting part of the model are the two lines that state the problem constraints using the cardinality constraint combinator `exactly`.

```
forall (v in Size)
  C.post(exactly(s[v],all(i in Size) s[i] == v));
```

In order to have a magic series, it is sufficient that, for every value v with $0 \leq v \leq n$, `s[v]` is equal to the number of occurrences of v in the series. This is equal to the number of constraints of the form `s[i] == v` that are satisfied. Using the `all` aggregator, we first create an array of boolean expressions

$$[s[0] == v, \dots, s[n-1] == v]$$

Then, the cardinality combinator specifies that exactly `s[v]` boolean expressions must be true:

```
exactly(s[v],all(i in Size) s[i] == v)
```

In general, a cardinality combinator `exactly(x,c)`, where `x` is an incremental variable and `c` is an array of boolean expressions, specifies that exactly `x` expressions in `c` must be true. Let k be the value of variable `x` and m the actual number of boolean expressions that are true. Then the number of violations of the constraint is given by $|k - m|$.

```

import cotls;
int n = 8;
range Size = 0..n-1;

Solver<LS> m();
var{int} s[Size](m, Size) := 0;
ConstraintSystem<LS> C(m);

forall (v in Size)
  C.post(exactly(s[v],all(i in Size) s[i] == v));
C.post(sum (i in Size) i*s[i] == n);
m.close();

int      tabuLength = 5;
bool     tabu[Size] = false;
Counter  it(m,0);
int      restartFreq = 10 * n;

whenever it@changes()
  if (it % restartFreq == 0)
    forall (i in Size) {
      s[i] := 0;
      tabu[i] = false;
    }
forall (i in Size)
  whenever s[i]@changes() {
    tabu[i] = true;
    when it@reaches[it + tabuLength]()
      tabu[i] = false;
  }
while (C.violations() > 0) {
  selectMax (i in Size : !tabu[i]) (C.violations(s[i]))
  selectMin (v in Size : s[i] != v) (C.getAssignDelta(s[i],v))
  s[i] := v;
  it++;
}

```

Statement 18.7: CBLS Model for the Magic Series Problem

Violations of individual variables are computed as follows: the number of violations of variable x is equal to the total violations of the constraint; if $k < m$, we assign one violation to each variable appearing in a true expression; if $k > m$, we instead assign one violation to each variable appearing in a false expression. Note that, in an **exactly** constraint, variables can appear in at most one boolean expression of the array. For example, assume that $n = 5$ and $\mathbf{s} = [1, 1, 0, 0, 0]$, and consider the constraint:

```
exactly(s[0], all(i in Size) s[i] == 0)
```

The constraint is then going to have two violations, which is also the number of violations for variable $\mathbf{s}[0]$. This is because three expressions are true and we only wanted one. Variables $\mathbf{s}[2]$, $\mathbf{s}[3]$, $\mathbf{s}[4]$ are going to have one violation each, since they appear in true expressions, whereas variable $\mathbf{s}[1]$ will have no violations, since it appears in a false expression.

The model is enhanced with a redundant numerical constraint, which corresponds to the fact that, for any magic series of length n , it holds that

$$\sum_{i=0}^{n-1} i s_i = n$$

Since this is only a necessary condition, it does not change the search space, but it helps guide the local search more efficiently. In summary, it makes it possible to break some of the symmetry of the problem, and it also compensates for the tendency to assign small values to variables $\mathbf{s}[i]$, $i > 0$, which reduce the violations of the cardinality constraints. The redundant constraint is specified as follows:

```
C.post(sum (i in Size) i*s[i] == n);
```

Once again, it is completely straightforward to combine in the same model cardinality combinators and numerical constraints.

Summary of Search The remaining part of Statement 18.7 contains the code to perform the search. The main search heuristic is a max-conflict/min-conflict search, combined with a tabu-search meta-heuristic. The most interesting aspect of this code is the use of *events* to implement the tabu search and to add a restart component. Events allow to do that in a modular fashion, leading to more readable code. This example also illustrates the use of **Counter** objects in the CBLS module. The remainder of this section will describe all this in more detail.

Basic Search Heuristic The basic search procedure is almost the same with the one used in the queens problem, the only change being the addition of a tabu-search component.

```
while (C.violations() > 0) {
  selectMax (i in Size : !tabu[i]) (C.violations(s[i]))
  selectMin (v in Size : s[i] != v) (C.getAssignDelta(s[i],v))
  s[i] := v;
  it++;
}
```

In every iteration, the procedure selects the non-tabu variable `s[i]` with the most violations, and then assigns it to the value that minimizes the total number of violations. The tabu list is defined as a boolean array `tabu` over the range of variables, so that `tabu[i]` is true when variable `s[i]` is tabu.

Counters Note the use of a counter `it` to keep track of the number of iterations. More precisely, it is defined as a `Counter` object, linked to the local solver `m` and initialized to 0, with the instruction:

```
Counter it(m,0);
```

Counters constitute a special case of monotonically increasing incremental variables and they are always defined with respect to a local solver. They are a very useful tool in building local search solutions. For example, they can be used in events that control the flow of the search procedure of the meta-heuristic employed.

Events As we also saw in Chapter 7, *events* constitute a fundamental control mechanism of COMET. An event consists of a condition and a block of code, and can appear textually separated from other blocks of code. Informally, an event acts like an alarm: once it has been posted, and when its condition is satisfied, an event-handler executes its corresponding block of code. Events can be nested, increasing their functionality. Different COMET objects support different types of events and, in addition, users can define their own events. In the magic series search example, we use events for writing modular code for a restarting component, on top of the main search, and for managing the tabu list.

Restarting Using Events In order to escape from poor-quality local optima, a commonly used technique in local search is to restart the search from scratch, if the solution found after a given number of iterations is not satisfactory. In the example of the magic series, a restart is equivalent to re-initializing the decision variables and the tabu list. A straightforward way to do this is by adding an outer loop around the main search loop, to manage restarts.

A more compositional approach is to write a separate block of code, which makes use of an event. This greatly increases code readability and re-usability. These are the lines that implement restarts:

```
whenever it@changes()
  if (it % restartFreq == 0)
    forall (i in Size) {
      s[i] := 0;
      tabu[i] = false;
    }
```

The effect of these lines of code is that, whenever counter `it` increases, we check whether `restartFreq` iterations have taken place since the last restart. If this is the case, each `s[i]` is reset to 0 and the tabu list is reset to false. The event utilized here is the `changes()` event, which is supported for counter objects. The instruction **whenever** specifies that the event's code block will be executed *every time* the counter `it` changes its value.

Tabu List Management Using Events As mentioned earlier, events can be nested naturally. This is used for managing the tabu list, as shown in the following excerpt of the code:

```
forall (i in Size)
  whenever s[i]@changes() {
    tabu[i] = true;
    when it@reaches[it + tabuLength]()
      tabu[i] = false;
  }
```

These lines first declare a different **changes** event for each decision variable **s[i]**:

```
forall (i in Size)
  whenever s[i]@changes() {...}
```

Whenever a variable **s[i]** changes its value, the code block of its corresponding event is executed. Each such code block first makes the variable **tabu** and then declare a nested event, **reaches**, for the counter **it**:

```
tabu[i] = true;
when it@reaches[it + tabuLength]()
  tabu[i] = false;
```

This is an example of a *key-event*, which executes its code block, once the object associated with the event reaches the value specified by the key in the square brackets (in our case, **it + tabuLength**). In our example, the effect of the event is that **tabu[i]** is reset to false after **tabuLength** iterations, so that the variable **s[i]** stops being tabu. The **reaches** event is supported by counters, but not by standard incremental variables.

Note that the instruction **when** specifies that the event's code will be executed *only once*, at the next occurrence of the event, and the event is not going to be active after that. This is not the case for the instruction **whenever**, which specifies that the code will be executed at every subsequent occurrence of the event, i.e., every time its condition is satisfied.

Another instruction of this kind is the instruction **foreveron**, which has the effect of executing the event's code both at the time that the event is declared, and at every subsequent occurrence of the event. For instance, if one wanted to track the current state of the variables and their violations not only after every iteration, but also at their initial state, a way to do that is by inserting the following block of code, that declares a **foreveron** event:

```
foreveron it@changes() {
  cout << "Iteration: " << it << endl;
  cout << s << "\t";
  cout << all(i in Size) C.violations(s[i]) << endl;
}
```

This would not be possible with **whenever**, since it would skip the initial state.

18.5 All-Interval Series

In this section:

- we introduce objective functions in COMET's CBLIS module
- we show a simple two-stage heuristic
- we give a more detailed description of first-order expressions

The Problem The All-Interval Series problem consists in finding a permutation of the first n non-negative integers such that the absolute differences between consecutive numbers are all different. A trivial solution to this problem is the sequence

$$(0, n-1, 1, n-2, 2, n-3, \dots)$$

Of course, in general, one is interested in finding a variety of non-trivial solutions. Statement 18.8 shows a COMET model and search for the all-interval series problem.

The Model The key of this model is to restate the problem as an optimization problem. An equivalent formulation of the all-interval problem is to maximize the number of different distances

$$|v_2 - v_1|, \dots, |v_n - v_{n-1}|$$

Clearly, this corresponds to a feasible solution of the original all-interval series formulation, if the number of different distances is equal to $n-1$.

After allocating a local solver, the model defines the size of the problem and the ranges used: **Size** is the range of the series, **Domain** is the set of possible values in the series, and **SD** is a truncated version of **Size** used in stating the objective function.

```
int    n      = 30;
range Size    = 1..n;
range Domain  = 0..n-1;
range SD      = 1..n-1;
```

```

import cotls;
Solver<LS> m();

int    n      = 30;
range Size    = 1..n;
range Domain  = 0..n-1;
range SD      = 1..n-1;

RandomPermutation perm(Domain);
var{int} v[Size](m,Domain) := perm.get();
Function<LS> 0 = maxNbDistinct(all(k in SD) abs(v[k+1]-v[k]));
m.close();

var{int} evaluation = 0.evaluation();
int nbSearches = 1;
int maxSearches = 100;
int it = 0;

while ((nbSearches <= maxSearches) && (evaluation != n-1)) {
  selectMax (i in Size) (0.increase(v[i]))
    selectMax (j in Size : j != i) (0.getSwapDelta(v[i],v[j]))
      v[i] := v[j];
  it++;
  if ((evaluation != n-1) && (it == 10000 * n)) {
    RandomPermutation perm(Domain);
    forall (i in Size)
      v[i] := perm.get();
    nbSearches++;
    it = 0;
  }
}

```

Statement 18.8: CBLS Model Using Functions for the All-Interval Series

Then, the decision variables are declared and initialized from a random permutation on `Domain`.

```
RandomPermutation perm(Domain);
var{int} v[Size](m,Domain) := perm.get();
```

What is new in this model, is the use of an objective function, `maxNbDistinct`, in order to keep track of the number of different distances:

```
Function<LS> O = maxNbDistinct(all(k in SD) abs(v[k+1]-v[k]));
```

In this definition, the problem's objective function, `O`, counts the number of distinct values in the array of absolute differences

$$[\text{abs}(v[2]-v[1]), \dots, \text{abs}(v[k+1]-v[k])]$$

created using the `all` aggregator:

$$\text{all}(k \text{ in } SD) \text{ abs}(v[k+1]-v[k])$$

Objective Functions in Local Search Later chapters of this tutorial will provide a more in-depth coverage of objective functions for local search, but here is some basic information. All objective functions in COMET implement the interface `Function<LS>`. A partial statement of this interface is the following:

```
interface Function<LS> {
    var{int} evaluation()
    var{int} increase(var{int})
    var{int} decrease(var{int})
    int      getAssignDelta(var{int},int)
    int      getSwapDelta(var{int},var{int})
    ...
}
```

In this interface, function `evaluation()` maintains in an incremental fashion the value of the objective function. Functions `increase(x)` and `decrease(x)` specify how much a given variable `x` can increase or decrease the function's evaluation, by changing its value to any value in its domain. Function `getAssignDelta(x,v)` returns the increase in the evaluation by assigning value `v` to variable `x`, and, finally, function `getSwapDelta(x,y)` shows the increase of the evaluation by swapping variables `x` and `y`.

Back to the all-interval series model, the objective function `O` appearing in Statement 18.8 is an instance of the class `MaxNbDistinct<LS>`, which implements the interface `Function<LS>` of COMET's CBLS module. There are several classes of objective functions already implemented in COMET. The user should refer to the HTML documentation of Module LS for a more complete list. In addition, users can also implement their own functions, as shown in some examples in later chapters.

A Two-Stage Greedy Heuristic The search procedure features a two-stage greedy heuristic with a restarting component. It starts by defining an incremental variable `evaluation` which is simply an alias to the current value of the objective functions evaluation:

```
var{int} evaluation = O.evaluation();
```

It then defines some auxiliary integer variables: `nbSearches` represents the current search, `maxSearches` is the maximum allowed number of searches, and `it` corresponds to the current step of the search.

The main loop of the search keeps running, as long as no solution has been found (`evaluation ≠ n-1`) and the number of searches is below the maximum limit.

```
while ((nbSearches <= maxSearches) && (evaluation != n-1)) {...}
```

The core of the search is this two-stage greedy heuristic:

```
selectMax (i in Size) (O.increase(v[i]))
  selectMax (j in Size : j != i) (O.getSwapDelta(v[i],v[j]))
    v[i] := v[j];
```

First, the outer selector picks the variable $v[i]$ that has the most potential to increase the value of the function. Then, this variable is swapped with another variable $v[j]$, chosen by the inner selector so that the swap maximizes the increase in the objective function. If no solution is found after a maximum number of steps, the search performs a restart: it re-initializes the decision variables to a different random permutation and resets the step counter:

```
if ((evaluation != n-1) && (it == 10000 * n)) {
    RandomPermutation perm(Domain);
    forall (i in Size)
        v[i] := perm.get();
    nbSearches++;
    it = 0;
}
```

First-Class Expressions The all-interval series illustrates another example, after the queens problem, of expressions simplifying the modeling. For this reason, this section concludes with a more detailed description of COMET's first-class expressions.

First-class expressions can be viewed as a generalized version of incremental variables. They are constructed from incremental variables and constants, using arithmetic, logical and relational operators. What makes them appealing is the fact that constraints and objective functions can also be defined over expressions, not only over incremental variables. This greatly simplifies the process of defining models involving complex expressions.

Let's review now the examples of using expressions in constraints and objective functions. In the queens problem, the model included the following alldifferent constraint:

```
S.post(alldifferent(all(i in Size)(queen[i] + i)));
```

This code can be seen as a shortcut for

```
expr{int} d[i in Size] = queen[i] + i;
S.post(alldifferent(d));
```

A declaration of the form **expr{int}** defines a first-class integer expression, that can be queried to evaluate the effect of assignments and swaps on its variables. The interface for these queries is similar to the interface for constraints or objective functions. For example, `d[i].getAssignDelta(queen[i],3)` returns the increase in the value of expression `d[i]`, if `queen[i]` is assigned to 5. This ability to query deltas in expressions is automatically incorporated by COMET, when querying the `alldifferent` constraint through `getAssignDelta` and `getSwapDelta`.

Expressions can also be defined over boolean variables, as in the following constraint of the magic series example:

```
S.post(exactly(s[v],all(i in Size) s[i] == v));
```

Again, this can be rewritten using boolean expressions:

```
expr{boolean} b[i in Size] = (s[i] == v);
S.post(exactly(s[v],b));
```

Things are similar with objective functions. In the all-interval series, the objective function 0:

```
Function<LS> 0 = maxNbDistinct(all(k in SD) abs(v[k+1]-v[k]));
```

can be viewed as a shortcut for the code:

```
expr{int} e[k in SD] = abs(v[k+1] - v[k]);
Function<LS> 0 = maxNbDistinct(e);
```

In this example, computing the value of a simple call `0.getSwapDelta(v[i],v[j])` can be quite tedious: up to four different expressions of the form `abs(v[k+1]-v[k])` may contain one of the variables `v[i]` and `v[j]`, and all these expressions have to be considered, in order to compute the effect of the swap into the objective function. Through the seamless composition of the `swapDelta` method for expressions with the `swapDelta` method for objective functions, this tedious process is automatically taken care of by COMET.

Chapter 19

Local Search Structures

This chapter gives more extensive information on the most common structures for constraint-based local search using `COMET`. This includes incremental variables, invariants, constraints and objective functions, as long as constructs to facilitate managing solutions and neighborhoods. All concepts are illustrated through examples and applications.

19.1 Incremental Variables

A key idea of COMET's local search module is the use of differentiable objects. Objects of this kind can be queried, to find out about the effect of a value assignment or a swap between variables on the object's evaluation. Incremental variables are the underlying basis of this differentiable-oriented architecture.

Incremental variables can be seen as a generalized version of typed variables with extra functionality. Incremental variables are defined in conjunction with a local solver and they are the building blocks for forming all kinds of differentiable objects within the local solver: expressions, invariants, constraints and objective functions. Typically, the local solver provides the mechanism for managing dependencies between variables that belong to the same differential object, and for updating the evaluation of differential objects after changes in the incremental variables' values.

Each incremental variable is assigned a domain of values, either automatically or explicitly by the user. There are four types of incremental variables: integer, floating point, boolean, and set over integers. Some examples of incremental variables are contained in the following code block:

```
import cotls
Solver<LS> m();
range Domain = 1..n;

var{int}      a(m,Domain);
var{float}    f(m) := 3.14;
var{bool}     b(m) := true;
var{set{int}} s(m) := {2,3,5,7,11};
```

Notice how assignment to incremental variables is done using the `:=` operator. Following a similar syntax, a very useful COMET operator is the operator `:=:` that can be used to swap two incremental variables, as we have seen, for instance, in the all-interval series problem.

Incremental Variable Interface All integer incremental variables are objects of class `var{int}`. The interface for this class is the following:

```
class var{int} {  
    void      exclude();  
    void      includ();  
    int       getId();  
    int       getSnapshot(Solution);  
    set{int}  getDomain();  
    Solver<LS> getLocalSolver();  
    void      setDomain(set{int});  
}
```

The supported methods can be summarized as follows:

- `exclude()`: excludes the variable from any solution obtained by the local solver
- `includ()`: includes the variable in any solution obtained by the local solver (default)
- `getId()`: returns the variable's id, in range $0..N$ (N : number of variables in the model)
- `getSnapshot(Solution)`: looks up the value of the variable in the given Solution object
- `getDomain()`: retrieves the variable's domain
- `setDomain(setint)`: specifies a domain for the values the variable can take
- `getLocalSolver()`: returns the local solver the variable belongs to (one solver per variable)

The API for float incremental variables, `var{float}`, is basically a restricted version of the `var{int}` class, and its supported methods work in similar fashion.

```
class var{float} {  
    void      exclude();  
    void      includ();  
    int       getId();  
    float     getSnapshot(Solution);  
    Solver<LS> getLocalSolver();  
}
```

The situation is similar for boolean incremental variables, **var{bool}**:

```
class var{bool} {
    void      exclude();
    void      includ();
    int       getId();
    bool      getSnapshot(Solution);
    Solver<LS> getLocalSolver();
}
```

The most advanced type of incremental variable described here is the incremental set over integers, **var{set{int}}**, that has an API similar to other types of incremental variables:

```
class var{set{int}} {
    void      exclude();
    void      includ();
    int       getId();
    void      insert();
    void      delete();
    int       getSize();
    Solver<LS> getLocalSolver();
}
```

The set-specific methods, **insert()**, **delete()**, **getSize()**, have the obvious functionality. Of course, behind the scenes, adding or removing elements from an incremental set can lead to highly non-trivial propagation.

Events on Incremental Variables Incremental variables support different kinds of events, although some events may not be supported by all types of incremental variables. There are also some exceptions on the use of the keywords **when**, **whenever** and **foreveron**. For example, **foreveron** cannot be used with the event **@changes(float old, float new)** in the case of floating point incremental variables. For enhanced functionality, incremental sets support two additional events: **@insert(int i)** and **@remove(int i)**. As a simple example, one can write:

```
whenever s@insert(int v)
    cout << v << " was added to " << s << endl;
```

Note that these two events can only be used with **when** and **whenever**.

19.2 Invariants

In this section:

- we introduce invariants in COMET, a key structure for constraint-based local search
- we use invariants to speed up the queens problem

Invariants are one-way constraints, expressed in terms of incremental variables and expressions. They specify a relation, that must be maintained under assignments of new values to the participating variables. Note that invariants have a completely declarative nature: they only specify the relation to be maintained incrementally; *not* how to update it. More specifically, an invariant is an instruction of the form:

$$v \leftarrow \text{exp}$$

where v is an incremental variable and exp is an expression.

Comet guarantees that, at any time during the computation, the value of variable v will be equal to the value of the expression exp . For example, the following COMET code ensures that, at all times, variable y will equal the square of variable x .

```
var{int} x(m);  
var{int} y(m) <- x^2;
```

An important thing to point out is that invariant declarations are not allowed to form cycles. Expressions on the right hand side can be arbitrarily complex. For example, one can use aggregators to declare the following invariant over an array a :

```
var{int} s(m) <- sum(i in 1..10) a[i];
```

This specifies that s is always the sum of $a[1]$ through $a[10]$. Every time a new value is assigned to position i of the array, the value of s is updated accordingly (in constant time).

19.2.1 Numerical Invariants

Numerical invariants can be formed using incremental variables, constants and standard operators supported by COMET:

- unary: `abs`, `floor`, `ceil`, `-`
- binary: `+`, `-`, `*`, `/`, `%`, `^`, `min`, `max`, `&&`, `||`
- relational: `>`, `<`, `<=`, `>=`, `==`, `!=`

One can also use the aggregate counterparts of binary operators, defined over sets or ranges:

- `sum(k in S)`
- `prod(k in S)`
- `min(k in S)`
- `max(k in S)`
- `and(k in S)`
- `or(k in S)`

19.2.2 Combinatorial Invariants

A more interesting family of COMET invariants are the combinatorial invariants. These invariants maintain relationships of a combinatorial nature between input and output variables and are an important structure in many applications. We now give a summary of the most common combinatorial invariants. In later sections of the tutorial we show some example that use these invariants.

Element Invariant This is an ubiquitous invariant in many applications. An element invariant is an expression in which an array is indexed by an incremental variable. For example, this is the case in the following statement:

```
var{int} m[i in R] <- f[g[i]]
```

which indexes the array `f` of incremental variables with the incremental variable `g[i]`. Semantically the meaning of this statement is straightforward, although things are non-trivial behind the scenes: in the above statement, array `m` has to be updated every time either `g[i]` or `f[v]` changes.

Count Invariant Consider an array of incremental variables **a**, defined over the range **R**, where all incremental variable have the same domain **D**. The **count** invariant maintains incrementally an array of range **D**, in which the i^{th} entry, for each $i \in D$, is the number of variables that are equal to i . Semantically, a statement of the form:

```
var{int}    a[R];
var{int}[] f = count(a);
```

is equivalent to:

```
var{int} a[R];
var{int} f[d in D] <- sum(i in R) (a[i] == d);
```

However, the actual implementation of **count** is much more time- and space-efficient.

Distribute Invariant A generalization of the **count** invariant that can prove very useful in some applications is the **distribute** invariant. Extending the **count** example, the **distribute** invariant incrementally maintains an array, in which the i^{th} entry is the set of indices of variables that are equal to i . For instance, the statement

```
var{int}    a[R];
var{set{int}}[] indices = distribute(a);
```

is semantically equivalent to:

```
var{int}    a[R];
var{set{int}} indices[d in D] <- filter(i in R) (a[i] == d);
```

Again, using the built-in **distribute** invariant provides higher efficiency.

19.2.3 Set Invariants

COMET also supports invariants over set expressions, that specify sets of integers, sets of floating-point numbers or sets of tuples. Standard set operators supported by COMET such as union, intersection and set difference, can be used to form such expressions, along with filtering operators, such as **filter** and **collect**, presented in earlier chapters of the tutorial. Furthermore, one can use operators such as **card** and **member** to maintain set cardinality and membership information as invariants. We are showing two illustrations of set invariants: an implementation of Heap Sort and a way to speed up the Queens Problem local search approach.

Heap Sort A simple illustration of set invariants is in implementing a heap sort algorithm, in which one can maintain the smallest element in the heap as an invariant. The COMET code for heap sort over incremental integer variables is the following:

```
function void heapSort(var{int}[] t) {
  Solver<LS> m = t.getLocalSolver();
  var{set{int}} Heap(m);
  forall (i in t.getRange())
    Heap.insert(t[i]);
  var{int} minHeap(m) <- min(i in Heap) i;
  m.close();
  forall (i in t.getRange()) {
    t[i] := minHeap;
    Heap.delete(minHeap);
  }
}
```

The input of this function is an array of incremental variables, **t**. In the first line, the function retrieves the local solver that contains the incremental variables in the array:

```
Solver<LS> m = t.getLocalSolver();
```

The next three lines define the heap, which is simply an incremental variable **Heap** of type **var{set{int}}**, and insert into it all the elements of array **t**.

```
var{set{int}} Heap(m);  
forall (i in t.getRange())  
    Heap.insert(t[i]);
```

The most important part of the code is the line that follows immediately after, which defines an invariant that ensures that the incremental integer variable **minHeap** will always be equal to the smallest integer in the heap.

```
var{int} minHeap(m) <- min(i in Heap) i;
```

In practice, this means that, every time the minimum element is removed from the set, **minHeap** will automatically update its value to the new minimum element. This is exploited by the main loop of the implementation, that keeps extracting the smallest value **minHeap** from **Heap**, until it the heap becomes empty.

```
forall (i in t.getRange()) {  
    t[i] := minHeap;  
    Heap.delete(minHeap);  
}
```

Speeding up Queens A more interesting application of invariants is speeding up the solution for the queens problem presented earlier. The core of the search procedure for the queens problem is repeated in this code block:

```
selectMax (q in Size) (S.violations(queen[q]))
selectMin (v in Size) (S.getAssignDelta(queen[q],v))
  queen[q] := v;
```

In Local Search, it is not unusual neighboring solutions to have very similar neighborhoods, and this also happens to be the case for the queens problem. Therefore, it is useful to be able to maintain some parts of the neighborhood incrementally, rather than recomputing them from scratch, after every transition to a neighboring solution.

Set invariants offer a neat way to do this. For this particular case, one can define a set of integers `MostViolatedQueens`, which, at every point, contains the set of queens that have the largest number of violations:

```
var{set{int}} MostViolatedQueens(m)
  <- argmax(all(i in Size) S.violations(queen[i]));
```

Note the use of set aggregator `argmax` for computing the set of most violated queen indices. The only other modification needed to be made to the original queens code is replacing the outer selector, that goes through all queens and recomputes the violations of each queen:

```
selectMax (q in Size) (S.violations(queen[q]))
```

with a selector that utilizes the dynamically updated set `MostViolatedQueens`:

```
select (q in MostViolatedQueens)
```

Because of the fact that there are only small updates to that set after each iteration, this can significantly speed up the code. The complete implementation of the queens approach using invariants is presented in Statement [19.1](#)

```

import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
var{set{int}} MostViolatedQueens(m)
    <- argmax(all(i in Size) S.violations(queen[i]));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
    select (q in MostViolatedQueens)
        selectMin (v in Size) (S.getAssignDelta(queen[q],v))
            queen[q] := v;
    it++;
}

```

Statement 19.1: CBLS Model for the Queens Problem Using Invariants

19.3 Constraints

Constraints in COMET are built on top of invariants, and are central in the architecture of the CBLS module. We've already seen different kinds of constraints in the introductory examples. In this section:

- we show an extended version of the `Constraint<LS>` interface
- we present a variety of useful constraints
- we show how to combine simple constraints into more complex ones

19.3.1 The Constraint Interface

This is an extended description of the `Constraint<LS>` API:

```
interface Constraint<LS> {
    Solver<LS>    getLocalSolver()
    var{int}[]    getVariables()

    var{boolean}  isTrue()
    var{int}      violations()
    var{int}      violations(var{int})
    var{int}      decrease(var{int})

    int           getAssignDelta(var{int},int)
    int           getAssignDelta(var{int},int,var{int},int)
    int           getAssignDelta(var{int}[],int[])
    int           getSwapDelta(var{int},var{int})
    int           getSwapDelta(var{int},var{int},var{int},var{int})

    void          post()
}
```

Some methods of the interface were already used in previous examples, so the focus, in this section, is on the rest of the methods. The first group of methods offers functionality for retrieving the local solver the constraint belongs to (`getLocalSolver`) and the incremental variables present in the constraint (`getVariables`)

The second group of methods returns information about satisfiability of the constraint. What is new here is the function `decrease(x)`, which has the same semantics as the `violations(x)` method, and returns the maximum decrease in the constraint's violations by assigning the incremental variable `x` to any other value in its domain. The total number of violations is given by the `violations()` method and method `isTrue()` returns true if the number of violations is 0.

The next group of methods is extensively used in local search applications. Supported are three variants of the `getAssignDelta` method and two variants of the `getSwapDelta` method:

- `getAssignDelta(x,v)` returns the difference in violations by assigning variable `x` to value `v`
- `getAssignDelta(x,v,y,w)` returns the difference in violations by assigning variable `x` to value `v` and variable `y` to value `w`
- `getAssignDelta(x[],v[])` returns the difference in violations by simultaneously assigning the variables of the array `x` to the values of the array `v`

In a similar fashion,

- `getSwapDelta(x,y)` returns the difference in violations by swapping variables `x` and `y`
- `getSwapDelta(x1,y1,x2,y2)` returns the difference in violations by swapping variable `x1` with `y1` and variable `x2` with `y2`

Finally, the method `post` is used for adding a constraint to a constraint system. In some sense, this method acts as “glue” for combining different constraints into constraints of more complex structure.

19.3.2 Violations

Constraints are characterized by their number of violations, which is a measure of how far from satisfaction the constraint is. In most cases, it also makes sense to assign a number of violations to individual variables of the constraint, in order to provide more guidance to the local search algorithm employed. There are three common ways to assign violations to a constraint:

- Decomposition-Based Violations
- Variable-Based Violations
- Value-Based Violations

Decomposition-Based Violations The most intuitive of the three is the decomposition-based approach. This is typically the case for constraint systems, that are formed as a conjunction of a number of constraints. In this case, it makes sense the violations of the constraint system to be equal to the sum of violations of the individual constraints that comprise the constraint system. Unfortunately, not all types of constraints are decomposable into conjunctions of simpler constraints

Variable-Based Violations Intuitively, this is the minimum number of variables that need to change, in order for the constraint to be satisfied. In this case, the number of violations is a direct measure of distance from feasibility, since it equals the number of simple local moves (in particular, assignments of values to variables) necessary, in order to reach a feasible solution. Unfortunately, this measure of violations is difficult to compute in most practical applications, and, in addition, it may not be well-defined, if the constraint is not satisfiable.

Value-Based Violations For a big family of combinatorial constraints, the most applicable way of assigning violations is value-based. More specifically, in constraints that specify lower or upper bounds on the number of occurrences of particular values in the variable assignment, one can base the number of violations on how well these bounds hold for different values in the domain. For example, if a particular value v appears three times and the constraint requires it to appear at most once, then we could assign two violations to that value. Then the total constraint violations will be equal to the sum of violations over all values in the domain of reference.

The best way to describe value-based violations is through the example of the `alldifferent` constraint on an array \mathbf{x} , which holds, if all variables of the array have distinct values, or, equivalently, when each value of the domain is assigned to at most one variable. Then, if a value is assigned $n > 1$ times, we assign $n - 1$ violations to it. See, for example, the graph of Figure 19.1, which depicts an assignment of variables (left ellipse) to values (right ellipse). In this assignment, two variables are assigned to the same value (the third value in the right ellipse, marked in red), whereas all other variables are assigned to different values. Following the value-based approach, this corresponds to one violation for the red value and no violations for the rest of the values. Therefore, the total number of violations of the constraint is equal to 1.

At this point, we also give an example of how differentiation work, using the same constraint. Figure 19.2, illustrates the situation in which variable \mathbf{x}_i is assigned value \mathbf{b} instead of \mathbf{a} . By doing that, the number of variables assigned to value \mathbf{a} is now going to become one, which eliminates the violations for that value. In fact, by doing this re-assignment, no value is assigned to more than one variable, so the total number of violations for the constraint will now become 0. Therefore, the function call `getAssignDelta(xi,b)` will return -1.

With all this in mind, we go on to present a number of different constraints of interest. We start with the simpler, numerical constraints, and then describe some of the combinatorial constraints implemented in `COMET`. For every type of constraint, we explain how violations are computed, both for the whole constraint and for individual variables.

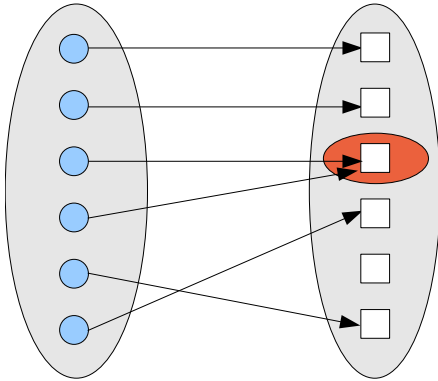


Figure 19.1: Assignment for an Alldifferent Constraint

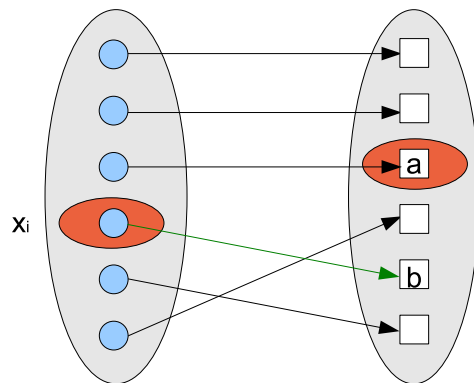


Figure 19.2: Effect of getAssignDelta for an Alldifferent Constraint

19.3.3 Numerical Constraints

Numerical Constraints are the simplest constraints in `COMET`. Similarly to numerical invariants, they are formed by combining incremental variables, constants and unary, binary and relational operators (`abs`, `floor`, `ceil`, `+`, `-`, `*`, `/`, `%`, `^`, `min`, `max`, `&&`, `||`, `>`, `<`, `<=`, `>=`, `==`, `!=`), or their aggregate counterparts.

We've seen several examples of numerical constraints in the introductory examples. Usually, one needs to define numerical constraints of the form: `L rel-op R`, where `L`, `R` are arithmetic expressions and `rel-op` is any binary relational operator. Depending on which operator is used, the violations of a constraint that is **not** satisfied are computed as follows:

- `L == R` : The number of violations is equal to `abs(L-R)`
- `L != R` : The number of violations is equal to 1
- `L <= R` : The number of violations is equal to `max(L-R,0)`
- `L < R` : The number of violations is equal to `max(L-R+1,0)`

The situation is symmetric for operators `>` and `>=`.

In all of the above cases, the number of violations assigned to each individual variable in the constraint is equal to the total number of violations of the constraint.

19.3.4 Combinatorial Constraints

Of course, the most important component of constraint-based local search are combinatorial constraints. This tutorial covers only a few of the combinatorial constraints available in `COMET`, in particular those that appear in the accompanying applications. You should consult the HTML reference for a complete list.

Alldifferent We've already given a detailed account of the `alldifferent` constraint, so this just a summary. Violations here are value-based. Every time a value is assigned to multiple variables in the constraint, each additional appearance of the value contributes to one violation of the constraint. For each individual variable, one violation is assigned, if a re-assignment of the variable to another value would decrease the total number of violations, otherwise no violation is assigned to the variable.

Before discussing other constraints, we make a brief comment on how to post constraints. As mentioned before, each constraint corresponds to a class implementing the interface `Constraint<LS>`, but the user does not need to directly call the constructor, in order to use a constraint: `COMET` offers a short-hand functionality for doing that.

For example, when adding an `alldifferent` constraint over array `a` to the constraint system `S`, with the following line of code:

```
S.post(alldifferent(a));
```

this is equivalent to the code:

```
AllDifferent<LS> c = new AllDifferent<LS>(a);
S.post(c);
```

In other words, the function `alldifferent` calls the constructor of class `AllDifferent<LS>`. This contributes to more compact and better readable code. In general, the convention is that functions of this kind are lower-case versions of the corresponding constraint class.

Atmost The `atmost` constraint can be seen as some kind of generalization of `alldifferent`. The `alldifferent` constraint limits the number of appearances of every value in the domain to at most one. With the `atmost` constraint we can choose the upper limit for each value in the domain. An `atmost` constraint is defined using an array of incremental variables and an integer array whose range is the variables' domain. The syntax is

```
atmost(int[] limit, var{int}[] x)
```

and here is a simple example:

```
range Size = 1..5;
range Domain = 2..4;
int lim[Domain] = [2, 1, 3];
var{int} a[Size](m,Domain);
Atmost<LS> c = atmost(lim,a);
```

This simply states that at most two variables can be equal to 2, at most one can be equal to 3, and at most three can be equal to 4.

The number of violations in the `atmost` constraint is value-based. Let $n(v)$ be the number of variables equal to v , and let $N(v)$ the upper limit set by the constraint. Then, the violations corresponding to value v is equal to $\max(n(v) - N(v), 0)$. The total number of violations for the

constraint is equal to the sum of the violations over all values in the domain. For individual variables, we assign a number of violations equal to the violations corresponding to its current value. In the previous example, if at some point $\mathbf{a} = [3, 3, 4, 3, 4]$, then there are 2 violations for the constraint (corresponding to value 3), and variables $\mathbf{a}[1]$, $\mathbf{a}[2]$ and $\mathbf{a}[4]$ have 2 violations each.

Knapsack Generalizing the **atmost** constraint one step further can lead to the **knapsack** constraint, which is defined on an array of incremental variables using two integer arrays: one specifying the “weight” of each variable in the array and one specifying the “capacity” of each value in the domain. The constraint is satisfied, if, for every value v in the domain, the sum of weights of the variables equal to v is at most equal to the capacity for that value. The **atmost** constraint is a special case of the **knapsack** constraint, in which all weights are equal to 1 and the capacity simply specifies the maximum number of times each value can appear. The syntax of the **knapsack** constraint is:

```
knapsack(var{int} x, int[] weight, int[] capacity);
```

Slightly modifying the **atmost** example given earlier, we get the following example for the **knapsack** constraint:

```
range Size = 1..5;
range Domain = 2..4;
var{int} a[Size] (m,Domain);
int weight[Size] = [5,1,3,1,7];
int cap[Domain] = [2,5,3];
Knapsack<LS> c = knapsack(a, weight, cap);
```

Once again, violations are value-based. If $w(v)$ is the total weight of variables equal to v , and $C(v)$ the capacity of that value, then the violations corresponding to value v is equal to $\max(w(v) - C(v), 0)$. The total number of violations is the sum of violations over all values in the domain. As in the **atmost** constraint, each variable is assigned the violations of its value.

In the above example, assuming that $\mathbf{a}=[3, 3, 2, 3, 4]$, there is one violation for value 2 (total weight is 3 and capacity is 2), two violations for value 3 (total weight 7 and capacity 5), and four violations for value 4 (total weight 7 for capacity 3). This leads to a total of seven violations. Also, each of the variables $\mathbf{a}[1]$, $\mathbf{a}[2]$ and $\mathbf{a}[4]$ has two violations, variable $\mathbf{a}[3]$ has one violation, and variable $\mathbf{a}[5]$ has four violations.

Sequence Finally, we describe the **sequence** constraint, which is a very useful modeling tool for many scheduling and sequencing applications. Different variants of this constraint are implemented in COMET, but the version presented here can be used in modeling car sequencing problems. The syntax of the constraint is the following

```
sequence(var{int}[] s, set{int} val, int ub, int window)
```

Given a sequence **s** of integer incremental variables and a set of integer values, **val**, the constraint holds, if, in every subsequence **s[i], ..., s[i+window-1]** of the array **s**, there are at most **ub** values from the set **val**.

For instance, in a setting of a car assembly line, this can be useful to specify constraints of the form: “at most four blue or green cars in every sequence of ten cars produced”. If **s** is the sequence of colors of the cars produced in the assembly line, this can be stated as:

```
enum Colors = {RED,BLUE,GREEN,BLACK};
SequenceConstraint<LS> D = sequence(s, {BLUE,GREEN}, 4, 10);
```

Note that the domain of the incremental variables appearing in the constraint should not be too large, to avoid overflows during the preprocessing involved in posting the constraint.

The assignment of violations to this constraint is done through constraint decomposition. The constraint can be seen as a conjunction of multiple **atmost** constraints, one per subsequence. Then the total number of violations is the sum of violations over all individual **atmost** constraints. The computation of violations for individual variables is done in a similar way: it is the sum of violations assigned to the variable, over all **atmost** constraints in which it participates.

19.3.5 Compositionality of Constraints

A fundamental feature of COMET is that it allows to build complex constraints by combining simpler constraints using different kinds of constraint-combinators. The resulting constraints still conform to the `Constraint<LS>` interface, often allowing for generic solution approaches. Some of these combinators, including constraint systems and cardinality combinators, are presented next.

Logical Combinators A common way of building complex constraints is through the use of logical connectives, conjunction and disjunction. Consider first a constraint c , which is the conjunction of two other constraints c_1 and c_2 . The constraint will hold, if both c_1 and c_2 are true. If c is not satisfied, then the number of violations is simply the sum of violations of its constituent constraints. Similarly, for each variable x in the constraint, we assign a number of violations, which is equal to the sum of the violations assigned to x because of constraint c_1 and the violations assigned to it because of constraint c_2 . For example, if $a=5$, $b=3$, and $c=3$, then the conjunctive constraint

```
S.post(a < b && b != c);
```

posted into the constraint system S will have a total of four violations, three because of $a < b$ and one from $b \neq c$. Since variable b participates in both constraints, it will also have four violations, while variable a will have three violations and variable c will have one violation. Notice how a decomposition-based approach can be applied for computing violations in this case.

Alternatively, one can define a disjunction of constraints. Assume, for example, that $c = c_1 \vee c_2$. Then, the violations of c is the minimum between the violations of c_1 and the violations of c_2 :

```
c.violations() = min(c1.violations(),c2.violations())
```

The violations for an individual variable c is equal to the maximum decrease to the total violations that can take place by assigning x to another value in its domain. Because the computation of the total violations contains a `min` operator, the formula for the violations of x is more complicated, compared to the conjunction of constraints.

```
c.violations() - min(c1.violations() - c1.violations(x),
                    c2.violations() - c2.violations(x))
```

Constraint Systems We've seen many instances of constraint systems, which are essentially conjunctions of constraints, so they are equivalent to conjunction logical combinators. This means that their violations are determined exactly the same way as in conjunction combinators. Their total violations is the sum of violations of their constituent constraints, and the same holds for the participating variables.

Constraint Systems implement the `Constraint<LS>` interface, enhancing it with some very useful methods, that allow individual access to its constraints.

- `getRange()` returns the range of constraints
- `getConstraint(i)` returns the constraint of index `i` within the constraint system
- `getCstrViolations()` returns an array with the violations of all constraints it contains

For local search applications, the most important aspect of constraint systems is that they allow to combine all kinds of heterogeneous constraints in a transparent fashion.

Cardinality Combinators In the magic series problems, we saw an example of a cardinality combinator, specifying that *exactly* k boolean expressions must be true. As mentioned there, the total number of violations is equal to the absolute difference between k and the actual number of true expressions. If fewer than k boolean expressions are true, we assign one violation to each variable contained in a false expression. Symmetrically, if more than k constraints hold, we assign a violation to each variable within a true expression.

A simple generalization of `exactly` is the constraint combinator `atmost`, which specifies that at most k boolean expressions must hold. When this is not the case, the number of violations is equal to the number of true expressions minus k . For each individual variable, a violation is assigned, if the variable is contained in one of the true expressions. Here is a simple example:

```
range Size = 1..5;
var{int} a[Size](m);
Constraint<LS> d = atmost(2, all(i in Size) a[i] == i + 1);
```

If we assume that `a = [2, 2, 4, 5, 4]`, then the number of violations of constraint `d` is equal to 1, since three boolean expressions are true and the upper limit is set to 2. One violation is assigned to each of the variables `a[1]`, `a[3]` and `a[4]`, which take part in expressions that evaluate to `true`.

Weighted Constraints Finally, in cases where the user wants to assign more weight to a particular constraint, for example, in models where that constraint is very important, COMET offers the possibility of a weighted constraint. The code

```
S.post(3 * alldifferent(x));
```

will add an `alldifferent` constraint of triple weight to the constraint system `S`. This code is a short-cut version of the following lines:

```
Constraint<LS> c = 3 * alldifferent(x);  
S.post(c);
```

The idea here is that the operator `*` is used to generate a new constraint, which will hold every time the underlying `alldifferent` constraint holds. The total violations of the new constraint will be the product of an integer weight (3 in this example) times the total violations of the original constraint. Similarly, for every variable contained in the constraint, the violations will be the product of this weight times the violations of the variable in the original constraint.

Remark: From the user's point of view, many of the constraints in the Local Search module have the same functionality and/or syntax with constraints in the Constraint Programming module. However, behind the scenes, there is a big difference in the way they are implemented: CP constraints are based on a constraint-store architecture, whereas LS constraints are built on top of incremental variables and invariants.

19.4 A Time Tabling Problem

We illustrate many of the concepts discussed so far by presenting a complete solution to a non-trivial application: an interesting time-tabling problem, in which we are given a number of *events* that need to be scheduled for time slots and rooms. Rooms are equipped with features and have capacities specifying the maximum number of students that each can accommodate. A number of students has registered to each event, and each event requires a set of features. Events need to be scheduled under the following restrictions:

- at most one event can be scheduled per room in any given time slot
- each student can attend at most one event per time slot
- each event must be scheduled for a room equipped with the required features
- each event must be scheduled for a room with enough capacity for its registered students

Note that, after some preprocessing, the last two constraints can be replaced by a constraint specifying a set of possible rooms for every event. Once these sets are determined, one does not have to consider capacities and features

The problem assumes that there are enough slots and rooms for scheduling the events. In order to simplify the problem's formulation, we can add some “dummy” events with no features and no students registered (so that they can be scheduled for any room or slot), in order to make the number of events equal to the number of room-slot pairs. In other words, to ensure that:

$$\#Events = \#Rooms \times \#Slots$$

Then the problem can be formulated as a one-to-one assignment between events and (room,slot) pairs, in which each event is assigned to a room within its set of possible rooms and no student attends more than one event per time slot. Arranged as a table, in which rows correspond to rooms and columns correspond to time-slots, the assignment may look like this, assuming that $\{E_1, E_2, \dots, E_n\}$ is the set of events.

| | | Time Slots | | | | | | | |
|-------|-----|------------|-----|--|--|--|--|--|--|
| Rooms | E1 | E16 | ... | | | | | | |
| | E20 | E4 | ... | | | | | | |
| | E2 | E30 | ... | | | | | | |
| | ... | ... | ... | | | | | | |

In a feasible solution, there is no intersection between the sets of students registered for events E1, E20 and E2, and likewise for the corresponding sets for events E16, E4, E30.

The complete COMET code for this problem is shown on Statements 19.2, 19.3 and 19.4. Statement 19.2 contains definitions for the variables that hold the problem's data. Statement 19.3 describes how to process the input data, in order to generate the main structures used in modeling the problem, and shows the actual local search model. Finally, Statement 19.4 contains the code for finding an initial (infeasible) time-table and for performing the search.

The complete code can also be found in file `timetableLS.co`, which loads data from file `timetable.data`. The full code also includes an animated visualization of the search. Visualization in COMET is discussed extensively in Part V of this tutorial. Section 25.2 describes how to add visualization elements to this particular time-tabling problem.

Reading the Data The code for reading the data, on Statement 19.2, starts by storing the input file name (given as the second argument argument in the command line, after the source code file) into string variable `file` and by creating an input stream `data` from the data input file.

```
string file = System.getArgs()[2];
ifstream data(file);
```

The first line of the input file contains the number of events, rooms, features and students. These are read into the variables `nbEvents`, `nbRooms`, `nbFeatures` and `nbStudents`, using the `getInt()` function. The number of slots, `nbSlots` is fixed to 45 (9 slots per day, for 5 days).

Then the code defines a number of ranges used in managing the problem data: `Slots`, `Events`, `Rooms`, `Features` and `Students`. An extra range `EventsExtended` is defined to index both actual and dummy events:

```
range EventsExtended = 1..(nbRooms*nbSlots);
```

Finally, the code retrieves data about room capacities, student registration to events and room and event features. These are stored in the following matrices:

- `roomCapa[r]`: capacity of room `r`
- `studentEvent[s,e]`: equals 1 if student `s` is registered for event `e`, or 0 otherwise
- `roomFeature[r,f]`: equals 1 if room `r` is equipped with feature `f`
- `eventFeature[e,f]`: equals 1 if event `e` requires feature `f`

```

string file = System.getArgs()[2];
ifstream data(file);

int    nbDays          = 5;
int    nbSlotsPerDay   = 9;
int    nbSlots         = nbDays * nbSlotsPerDay;
int    nbEvents        = data.getInt();
int    nbRooms         = data.getInt();
int    nbFeatures      = data.getInt();
int    nbStudents      = data.getInt();

range Slots            = 1..nbSlots;
range Events           = 1..nbEvents;
range EventsExtended   = 1..(nbRooms*nbSlots); // includes dummy events
range Rooms            = 1..nbRooms;
range Features         = 1..nbFeatures;
range Students         = 1..nbStudents;

int roomCapa[Rooms] = data.getInt();

int studentEvent[Students,EventsExtended] = 0;
forall (s in Students,e in Events)
    studentEvent[s,e] = data.getInt();

int roomFeature[Rooms,Features] = data.getInt();

int eventFeature[EventsExtended,Features] = 0;
forall (e in Events, f in Features)
    eventFeature[e,f] = data.getInt();

```

Statement 19.2: CBLS Model for Time-Tabling: I. Data Collection

Note that matrices `studentEvents` and `eventFeature` are defined for dummy events also. Since the input data is restricted to actual (non-dummy) events, storing data into these matrices cannot be done in a single line, as in the case of `roomFeature`, for example: this would store information pertaining to actual events into dummy events. This is why we need a **forall** loop (that only goes through the range `Events`) to store values into these two matrices

```
int studentEvent[Students,EventsExtended] = 0;
forall (s in Students, e in Events)
    studentEvent[s,e] = data.getInt();
```

Preprocessing and Model One could attempt to define a model for the problem without any further processing of the data. However, this would be inefficient, since it is possible to abstract the problem a little further. This is done in Statement 19.3. It first computes a number of set arrays based on the problem's data:

- `eventFeatureSet[e]`: set of features required by event `e`
- `roomFeatureSet[r]`: set of features in room `r`
- `studentsInEvent[e]`: set of students registered for event `e`

This is done using the **filter** operator for sets. For example, the code for the `eventFeatureSet` is the following:

```
set{int} eventFeatureSet[e in EventsExtended] =
    filter (f in Features) (eventFeature[e,f] == 1);
```

For every event `e`, the filter operator goes through all features `f` and returns the ones that are required by the event, i.e., the ones for which `eventFeature[e,f]` equals 1. The computation is similar for the other two sets. We also compute an array containing the number of students registered for every event:

```
int nbStudentsInEvent[e in EventsExtended] =
    studentInEvents[e].getSize();
```

```

// PREPROCESSING
set{int} eventFeatureSet[e in EventsExtended] =
    filter (f in Features) (eventFeature[e,f] == 1);
set{int} roomFeatureSet[r in Rooms] =
    filter (f in Features) (roomFeature[r,f] == 1);
set{int} studentsInEvent[e in EventsExtended] =
    filter (s in Students) (studentEvent[s,e] == 1);
int      nbStudentsInEvent[e in EventsExtended] =
    studentInEvents[e].getSize();

// Compute the set of events that can be scheduled in each room
set{int} possEventsInRoom[r in Rooms] =
    filter (e in EventsExtended) (roomCapa[r] >= nbStudentsInEvent[e]
    && (and(f in eventFeatureSet[e]) (roomFeatureSet[r].contains(f))));

// Compute the set of rooms that can accept each event
set{int} possRoomsOfEvent[e in EventsExtended] =
    filter (r in Rooms) (roomCapa[r] >= nbStudentsInEvent[e]
    && (and(f in eventFeatureSet[e]) (roomFeatureSet[r].contains(f))));

// Conflict matrix gives how many students attend both events in a pair
int conflictMatrix[EventsExtended,EventsExtended] = 0;
forall (e1 in EventsExtended,e2 in EventsExtended: e1 != e2) {
    set{int} attendBoth = (studentsInEvent[e1] inter studentsInEvent[e2]);
    conflictMatrix[e1,e2] = attendBoth.getSize();
}

// MODEL
import cotls;
Solver<LS> ls = new Solver<LS>();
ConstraintSystem<LS> S(ls);

var{int} X[Rooms,Slots](ls,EventsExtended);
forall (s in Slots, r1 in Rooms, r2 in Rooms : r1 < r2)
    S.post(conflictMatrix[X[r1,s],X[r2,s]] == 0);

S.close();
ls.close();

```

Statement 19.3: CBLS Model for Time-Tabling: II. Preprocessing and Model

Preprocessing concludes with computing a concise representation of the input data, summarizing all the information needed for the model and the search in the following structures:

- `possEventsInRoom[r]`: set of events that can be scheduled in room `r`
- `possRoomsOfEvent[e]`: set of rooms that can host event `e`
- `conflictMatrix[e1,e2]`: number of students attending both events `e1` and `e2`

The sets of possible rooms and events are computed on the basis of meeting capacity and feature requirements. In computing `possEventsInRoom[r]` a **filter** operator selects the events `e`, whose number of students fits into the room's capacity and whose set of features is a subset of the room's set of features:

```
set{int} possEventsInRoom[r in Rooms] =
    filter (e in EventsExtended) (roomCapa[r] >= nbStudentsInEvent[e]
    && (and(f in eventFeatureSet[e]) (roomFeatureSet[r].contains(f))));
```

Note the use of the aggregate version **and** of the `&&` operator, used for ensuring that all features `f` in `eventFeatureSet[e]` are contained in `roomFeatureSet[r]`.

The code for computing `possRoomsOfEvent` is almost identical. The only difference lies in the fact that the **filter** operator now selects rooms instead of events, but the condition specifying capacity and feature requirements remains the same.

Finally, computing the conflict matrix involves a **forall** selector, that goes through all pairs of events `e1` and `e2`, `e1 ≠ e2`, including dummy events, and calculates the intersection of their sets of students. Then it stores the intersection's cardinality into the corresponding position of the conflict matrix.

```
forall (e1 in EventsExtended, e2 in EventsExtended: e1 != e2) {
    set{int} attendBoth = (studentsInEvent[e1] inter studentsInEvent[e2]);
    conflictMatrix[e1,e2] = attendBoth.getSize();
}
```

The decision variables of the problem's local solver `ls` is simply a matrix `X` indexed on rooms and slots, so that `X[r,s]` is the event assigned to room `r` for time slot `s`.

```
var{int} X[rooms,slots](ls,eventsExtended);
```

During the search, the capacity and feature requirements are maintained as hard constraints. The search is then trying to minimize the number of students that attend more one event in the same time slot. This is reflected in the model through numerical constraints stating that the conflict matrix be equal to 0, for all pairs of events assigned to the same time slot. Note how the language allows to index the conflict matrix by two incremental variables.

```
forall (s in Slots, r1 in Rooms, r2 in Rooms : r1 < r2)
  S.post(conflictMatrix[X[r1,s],X[r2,s]] == 0);
```

Search: Initialization The search procedure has two basic components: an initialization phase and a hill-climbing search phase. We first give a more detailed description of the initialization. The initial time-table is found using a greedy procedure, that satisfies all hard constraints ignoring student conflicts. The procedure first defines a set of available (not scheduled) events, initialized from the range of all events:

```
set{int} availableEvents = collect(e in EventsExtended) e;
```

Then it goes through the rooms by decreasing number of possible events:

```
forall (r in Rooms) by (possEventsInRoom[r].getSize()) {...}
```

and, for each room, it attempts to assign an event to each time-slot.

```

// INITIALIZATION
set{int} availableEvents = collect(e in EventsExtended) e;
forall (r in Rooms) by (possEventsInRoom[r].getSize())
  forall (s in Slots)
    selectMin (e in availableEvents inter possEventsInRoom[r])
      (possRoomsOfEvent[e].getSize()) {
      X[r,s] := e;
      availableEvents.delete(e);
    }
if (availableEvents.getSize() != 0)
  System.terminate();

// SEARCH
while (S.violations() > 0)
  select (r1 in Rooms, s1 in Slots : S.violations(X[r1,s1]) > 0)
    selectMin (r2 in Rooms : possEventsInRoom[r2].contains(X[r1,s1]),
      s2 in Slots : possEventsInRoom[r1].contains(X[r2,s2]))
      (S.getSwapDelta(X[r1,s1],X[r2,s2]))
      X[r1,s1] := X[r2,s2];

```

Statement 19.4: CBLS Model for Time-Tabling: III. Initialization and Search

Given a room r and a slot s , it selects the event e with the smallest number of room options (smallest possible `possRoomsOfEvent(e)`), among the available events that can be scheduled for room r . This is done with the inner-most selector:

```
selectMin (e in availableEvents inter possEventsInRoom[r])
  (possRoomsOfEvent[e].getSize())
```

Once an event has been selected, it is scheduled for room r and time slot s and removed from the set of available events:

```
X[r,s] := e;
availableEvents.delete(e);
```

If there are still unscheduled events after the initialization process, the program terminates with a failure, although for relatively simple instances, this doesn't happen so often.

Search: Hill Climbing The main search procedure described on Statement 19.4 is a hill-climbing procedure, that keeps swapping events randomly, satisfying room requirements, until it reaches a feasible time-table. In every iteration, the search first selects a room-slot pair, $(r1,s1)$, for which the scheduled event, $e1=X[r1,s1]$, has violations:

```
select (r1 in Rooms, s1 in Slots : S.violations(X[r1,s1]) > 0) {...}
```

Then, the inner selector is choosing another pair, $(r2,s2)$, so that event $e1$ can be scheduled for room $r2$. The second pair is selected so that the event $e2=X[r2,s2]$ scheduled there can also be scheduled for room $r1$:

```
selectMin (r2 in Rooms : possEventsInRoom[r2].contains(X[r1,s1]),
          s2 in Slots : possEventsInRoom[r1].contains(X[r2,s2]))
  (S.getSwapDelta(X[r1,s1],X[r2,s2]))
X[r1,s1] := X[r2,s2];
```

Among all possible selections for the second pair, the selector chooses the one leading to the smallest total number of violations, after swapping the two events. The main body of the selector simply performs the swap between the events scheduled for the selected room-slot pairs. As mentioned earlier, we describe an animated visualization for this problem in Section 25.2.

19.5 Solutions - Neighbors

In this section we introduce *Solutions* and *Neighbors*, two local search structures that greatly facilitate the development of local search algorithms. Their usefulness is illustrated in solving the Progressive Party Problem.

19.5.1 The Progressive Party Problem

The Progressive Party problem is an abstraction of a yachting party, during which different crews (the *guests*) visit different boats (the *hosts*) for a number of time periods. The goal is to find an assignment of guests to boats for each time period, under the following constraints:

- No boat can host more persons than its capacity
- A guest cannot visit the same boat twice
- No two guests can meet more than once

The simpler version of this problem is a satisfaction problem: we are looking for an assignment of guests to boats for a given number of periods. The optimization version of the problem, where one is asked to find an assignment with the largest possible number of periods, is equivalent to solving a sequence of satisfaction problems, for increasing number of periods. This section focuses on the satisfaction version of the problem, and describes a tabu-search approach with several advanced features, such as: variable-length tabu list, aspiration criteria, intensification, restarts, and multiple neighborhoods. We start by showing how to formulate a model for the problem.

The Model The model used for the problem is shown in Statement 19.5. For clarity of presentation, the data initialization is not shown here. The complete code, including data collection, can be found in the COMET codes accompanying this tutorial. The problem's data can be fully described by three ranges for **Hosts**, **Guests**, and **Periods**, and two arrays specifying the space requirements:

- `cap[h in Hosts]`: Capacity of host boat `h`
- `crew[g in Guests]`: Size of guest crew `g`

The decision variables used are in the form of a matrix

`boat[g in Guests,p in Periods]`

representing the boat visited by guest `g` during period `p`.

```

// THE DATA
range Hosts;
range Guests;
range Periods;
int    cap[Hosts];      // capacity of hosts
int    crew[Guests];    // size of crews

// THE MODEL
import cotls;
Solver<LS> m();
UniformDistribution distr(Hosts);
var{int} boat[Guests,Periods](m,Hosts) := distr.get();
ConstraintSystem<LS> S(m);

forall (g in Guests)
    S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
forall (p in Periods)
    S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
forall (i in Guests,j in Guests : j > i)
    S.post(atmost(1,all(p in Periods) boat[i,p] == boat[j,p]));
var{int} violations = S.violations();
m.close();

```

Statement 19.5: Progressive Party in CBLS: The Model

The `boat` matrix is initialized to random values, using a uniform distribution `distr` over the values in the `Hosts` range:

```
UniformDistribution distr(Hosts);
var{int} boat[Guests,Periods](m,Hosts) := distr.get();
```

In the constraint system defined in the model, there are three sets of constraints: The first one ensures that each guest `g` visits each boat at most once, by specifying an `alldifferent` constraint on the boats hosting `g` over all periods.

```
forall (g in Guests)
  S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
```

The second set of constraints, specifies that boat capacities cannot be exceeded.

```
forall (p in Periods)
  S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
```

The knapsack constraint gets handy in this case: For any given period `p`, the different boats can be thought of as an array of containers into which items (crews) of different sizes are placed, respecting the container (boat) capacities. This array is formed by applying aggregator `all` to matrix `boat[g,p]`:

```
all (g in Guests) boat[g,p]
```

Both sets of constraints are weighed by a factor of 2, in order to increase their impact on the model's violations. In practice, this means that violations of these constraints are multiplied by 2.

The third set of constraints, defined over all unordered pairs of guests `{i,j}`, states that any two guests can meet at most once during the party. The cardinality combinator `atmost`, described in earlier sections, is used to ensure that at most one constraint of the form: `boat[i,p] == boat[j,p]` is satisfied, for each pair `{i,j}`.

```
forall (i in Guests,j in Guests : j > i)
  S.post(atmost(1,all(p in Periods) boat[i,p] == boat[j,p]));
```

The Search Once the model has been formulated, the search can be added on top of it without any concerns about the variety of combinatorial constraints used in the model. In fact, any kind of search based on assignments and swaps over the entries of matrix `boat[g,p]` would be compatible with the above defined model.

The core of the tabu search procedure presented here is a min-conflict heuristic, which, during each iteration, chooses a period and a guest and moves it to a different boat. This heuristic is gradually extended, to include aspiration criteria, intensification, restarts, and, eventually, an extra neighborhood based on swaps. The complete code is shown on Statements 19.10 and 19.11, and we describe it in gradual refinements.

Basic Tabu Search Scheme The basic tabu search procedure, which is based on a dynamic tabu list length, is shown on Statement 19.6. The tabu list of the algorithm is based on tuples of the form `<g,p,h>` indicating that the following assignment is tabu:

```
boat[g,p] := h
```

The core of the procedure is the following max/min conflict selection:

```
selectMax (g in Guests, p in Periods) (S.violations(boat[g,p]))
  selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
    tabu[g,p,h] <= it) (delta) {...}
```

The outer selector selects the guest-period pair `(g, p)` for which the most violations appear. Then, the inner selector chooses a host `h` that leads to the smallest number of violations (the minimal `delta`), if we move the pair `(g,p)` to it. Of course, only non-tabu assignments are considered. Note the way that expressions can be factorized in selectors. By setting:

```
delta = S.getAssignDelta(boat[g,p],h)
```

we can reuse the value of `delta` anywhere in the selector, without having to recompute it.

```

int it = 1;
int tabu[Guests,Periods,Hosts] = -1;
int tbl = 3;
int tblMin = 2;
int tblMax = 10;

while (violations > 0) {
    int old = violations;
    selectMax (g in Guests, p in Periods) (S.violations(boat[g,p]))
    selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
        tabu[g,p,h] <= it) (delta) {
        tabu[g,p,boat[g,p]] = it + tbl;
        boat[g,p] := h;
        if (violations < old && tbl > tblMin)
            tbl--;
        if (violations >= old && tbl < tblMax)
            tbl++;
    }
    it++;
}

```

Statement 19.6: Progressive Party in CBLS: Dynamic Length Tabu Search

The main body of the selector simply makes the assignment of the pair to the new boat, and updates the tabu status by making the old boat assignment tabu. The tabu list has a dynamically updated length, whose parameters are defined using the following variables:

```
int tbl = 3;  
int tblMin = 2;  
int tblMax = 10;
```

Variable `tbl` denotes the current length of the tabu list, `tblMin` the minimum length the algorithm allows for and `tblMax` the maximum length. The update is very simple and performed as follows:

```
if (violations < old && tbl > tblMin)  
    tbl--;  
if (violations >= old && tbl < tblMax)  
    tbl++;
```

Every time an assignment reduces the number of violation in the constraint system, the length is reduced by one. If, on the other hand, the assignment increases the violations, then the length is increased by one. Increases or decreases to the tabu length are restricted within the range defined by `tblMin`, `tblMax`.

Aspiration criteria The efficiency of the basic tabu procedure can be enhanced by using an aspiration criterion. The idea is to override the tabu status of an assignment, whenever it improves the best solution found so far. The motivation is rather obvious: since a new best solution has been found, we temporarily ignore the tabu status, in order not to miss the chance of reaching it.

Implementing this is really straightforward and only takes two changes into the original tabu search code: keeping track of the smallest number of violations found and enhancing the selector condition, as shown in Statement [19.7](#).

```

int it = 1;
int tabu[Guests,Periods,Hosts] = -1;
int tbl = 3;
int tblMin = 2;
int tblMax = 10;
int best = violations;

while (violations > 0) {
    int old = violations;
    selectMax (g in Guests, p in Periods) (S.violations(boat[g,p]))
    selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
        tabu[g,p,h] <= it || delta + violations < best) (delta) {
        tabu[g,p,boat[g,p]] = it + tbl;
        boat[g,p] := h;
        if (violations < old && tbl > tblMin)
            tbl--;
        if (violations >= old && tbl < tblMax)
            tbl++;
    }
    if (violations < best)
        best = violations;
    it++;
}

```

Statement 19.7: Progressive Party in CBLS: Aspiration Criteria

We keep track of the violations of the best solution found so far by defining a variable `best` and updating it every time after each new assignment:

```
int best = violations;
...
if (violations < best)
    best = violations;
```

Then, we only need to change the selection condition of the inner selector of the tabu search:

```
selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
           tabu[g,p,h] <= it || delta + violations < best) (delta) {...}
```

A new boat to host the pair (g,p) can be selected, if it is not tabu, or if the new number of violations, equal to `delta + violations`, is smaller than the best value found so far, given by variable `best`.

19.5.2 Using Solutions

Before describing the rest of the features used in the tabu search for the progressive party, we summarize the COMET concept of **Solution**, a very useful feature for managing solutions found during a search. Given a local solver, they can be used to store the current assignment of all incremental variables, *excluding* those defined through invariants. Solutions are first-class objects, which means that they can be passed as parameters, stored in data structures, or returned as results of function calls. Typically, one can define a Solution object with a call

```
Solution s(m);
```

At any point of execution of an algorithm, one can restore a solution **s** using the **restore** method.

```
s.restore();
```

This will restore all variables to their state at the point when solution **s** was created. The solver will then automatically update any other object that depends on the incremental variables of the solution. In addition to restoring a whole solution **s**, one can also query it to determine the values of specific variables. For instance,

```
x.getSnapshot(s);
```

returns the value of incremental variable **x** in solution **s**.

We need to emphasize, once again, that solutions *do not store* the assignment of variables that are defined through invariants, i.e., variables that appear on the left-hand side of an invariant. This means that, if one attempts to call **getSnapshot()** on a variable that is defined through an invariant, this will lead to an error. For example, the following block would generate an error:

```
var{int} x;  
var{int} y <- x + 1;  
y.getSnapshot(s);    // this produces an error
```

Solutions can greatly facilitate coding meta-heuristics that reuse already found solutions. This is illustrated in adding an intensification component to the tabu search for the progressive party problem.

Intensification Intensification is one of the ways to exploit more out of already found good solutions. The basic idea is to return to the best solution found so far, if no improvement has taken place for a number of iterations. Intuitively, The goal is to explore “good” neighborhoods more extensively and avoid too long walks in “bad” neighborhoods. Of course, the neighbor selection must have some degree of randomization, so that a different search path is followed after each return to the best solution found so far.

Adding intensification leads to the search procedure presented in Statement 19.8. The code is using three additional variables:

```
Solution solution(m);  
int      nonImprovingSteps = 0;  
int      maxNonImproving = 100;
```

which define a **Solution** object **solution**, an integer variable **nonImprovingSteps**, that counts the number of iterations in which no improvement has taken place to the best found solution, and a variable **maxNonImproving**, that stores the maximum number of non-improving iterations, before an intensification is performed. The actual intensification component of the implementation runs after each new assignment, and covers three cases:

If the new assignment improves the best solution found, then the new assignment is stored into the **Solution** object **solution**, variable **best** is updated to reflect the new best number of violations, and the number of non-improving iterations is reset to 0:

```
if (violations < best) {  
    best = violations;  
    solution = new Solution(m);  
    nonImprovingSteps = 0;  
}
```

```

int it = 1;
int tabu[Guests,Periods,Hosts] = -1;
int tbl = 3;
int tblMin = 2;
int tblMax = 10;
int best = violations;

Solution solution(m);
int      nonImprovingSteps = 0;
int      maxNonImproving = 100;

while (violations > 0) {
    int old = violations;
    selectMax (g in Guests, p in Periods) (S.violations(boat[g,p]))
    selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
        tabu[g,p,h] <= it || delta + violations < best) (delta) {
        tabu[g,p,boat[g,p]] = it + tbl;
        boat[g,p] := h;
        if (violations < old && tbl > tblMin)
            tbl--;
        if (violations >= old && tbl < tblMax)
            tbl++;
    }
    if (violations < best) {
        best = violations;
        solution = new Solution(m);
        nonImprovingSteps = 0;
    }
    else if (nonImprovingSteps == maxNonImproving) {
        solution.restore();
        nonImprovingSteps = 0;
    }
    else
        nonImprovingSteps++;
    it++;
}

```

Statement 19.8: Progressive Party in CBLS: Intensification

If this is not the case, we check whether the maximum limit for non-improving iterations has been reached, in which case the currently best solution, stored in variable `solution` is restored and the number of non-improving iterations is reset to 0:

```
else if (nonImprovingSteps == maxNonImproving) {  
    solution.restore();  
    nonImprovingSteps = 0;  
}
```

Finally, if no improvement occurred but we are still below the limit for non-improving iterations, the counter of non-improving is increased:

```
nonImprovingSteps++;
```

Restarts A local search heuristic may often get trapped in a region with no “good” solutions or where “good” solutions are hard to reach (they may require a long series of transitions). Although, intensification can help in escaping such “bad” regions, it may not be sufficient, since the best found solution could be isolated from solutions with fewer violations (requiring more iterations to reach than the maximum limit for non-improving steps).

A common way to get around this situation is through restarts: if no feasible solution is found after a number of iterations, the search is restarted from a new random assignment. We can include restarts to the search with a single block of code, as shown on Statement 19.9.

The extra lines are in the **if** block right before increasing the iteration counter `it`. At the end of each iteration, we check, whether a number of iterations have gone by since the last restart, without any feasible solution found (`best > 0`). The number of iterations between restarts is given by the variable `restartFreq` defined in the beginning of the search. Thus, the condition for a restart is

```
if ((it % restartFreq == 0) && (best > 0))
```

```

int tabu[Guests,Periods,Hosts] = -1;
int it = 1; int tbl = 3; int tblMin = 2; int tblMax = 10; int best = violations;
Solution solution(m);
int nonImprovingSteps = 0; int maxNonImproving = 100; int restartFreq = 1000;

while (violations > 0) {
    int old = violations;
    selectMax (g in Guests, p in Periods) (S.violations(boat[g,p]))
    selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
        tabu[g,p,h] <= it || delta + violations < best) (delta) {
        tabu[g,p,boat[g,p]] = it + tbl;
        boat[g,p] := h;
        if (violations < old && tbl > tblMin)
            tbl--;
        if (violations >= old && tbl < tblMax)
            tbl++;
    }
    if (violations < best) {
        best = violations;
        solution = new Solution(m);
        nonImprovingSteps = 0;
    }
    else if (nonImprovingSteps == maxNonImproving) {
        solution.restore();
        nonImprovingSteps = 0;
    }
    else
        nonImprovingSteps++;

    if ((it % restartFreq == 0) && (best > 0)) {
        with delay(m)
            forall (g in Guests, p in Periods)
                boat[g,p] := distr.get();
        best = violations;
        solution = new Solution(m);
    }
    it++;
}

```

Statement 19.9: Progressive Party in CBLS: Restarts

In case of a restart, first all variables are re-assigned to random values by querying the distribution `distr`, declared when stating the model:

```
with delay(m)
  forall (g in Guests, p in Periods)
    boat[g,p] := distr.get();
```

The instruction

```
with delay(m)
```

specifies that no assignment or propagation is performed, before all new random values have been queried from the random distribution. Without this instruction, a propagation step would take place after each individual assignment, which is less efficient than doing all the necessary propagation after all new values have been determined.

After the new random solution has been generated, variables `best` and `solution` are re-initialized accordingly:

```
best = violations;
solution = new Solution(m);
```

19.5.3 Using Neighbors

Very often in local search algorithms the neighborhood is composed of a number of different, usually heterogeneous, sub-neighborhoods. As a result, it is not possible to select moves using a single selector. In cases like that, one is forced to go through multiple selection phases: in a first phase, scan each sub-neighborhood separately, in order to evaluate their moves; and in a later phase, select both a sub-neighborhood and a move from that sub-neighborhood. This may lead to code of reduced readability.

COMET supports a **neighbor** construct, based on closures, that can address issues related to heterogeneous neighborhoods. Informally, a neighbor is defined through a pair (δ, M) , where δ is a move's evaluation and M contains the code for the move. Neighbor objects can be stored in special container objects called *neighbor selectors*, which can store moves together with their evaluations. After declaring a neighbor selector N , the user can add a neighbor to it with an instruction of the form:

```
neighbor( $\delta, N$ )  $M$ ;
```

In this statement, M is the code for the move, in the form of a closure. Closures are covered in more detail elsewhere in this tutorial, but one can use neighbors even without a full understanding of the closure mechanism. From an operational point of view, defining a neighbor is equivalent to recording the move it performs, without executing it.

There are several classes of neighbor selectors supported by COMET. The complete list of neighbor selectors supported can be found on COMET's reference. Here are the most useful ones:

MinNeighborSelector: greedy neighbor selector containing the best move submitted so far, along with its evaluation

AllNeighborSelector: collects all moves submitted, in increasing order of evaluation

KMinNeighborSelector: semi-greedy selector choosing randomly among the k best neighbors

Of course, users can always define their own selectors, as long as they conform to the same interface.

A neighbor selector can be queried through the method `hasMove`, to check if it contains any move. The method `getMove` for neighbor selectors returns the closure corresponding to the select move's code. One can then perform the move by invoking the `call` method of closure objects, which runs the closure's code. Therefore, once moves have been stored into a neighbor selector, one can select and perform one of the moves with only two extra lines of code:

```
if (N.hasMove())
  call (N.getMove());
```

This will become more clear, when explaining how to extend the neighborhood for the progressive party problem.

Extending the Neighborhood The neighborhood we have seen so far consists only of assignments to new values. The search can become much more efficient, if we also include in the neighborhood swaps between boat assignments. In particular, given a guest g and a period p , the new move will select another guest, g' , and swap the boat assignments of g and g' in period p . Guest g' is selected so that the number of violations after the swap is minimal. The only technical difficulty is that we need a different selector than the one used in the other move of the neighborhood (which chooses a new boat for a given guest in a given period).

To overcome this difficulty, we define moves using neighbors. The complete code appears on Statement 19.10. Since we are interested in selecting the best move at each point, the code first defines a `MinNeighborSelector N` in line 8:

```
MinNeighborSelector N();
```

Move definitions take place in lines 12–28 of Statement 19.10, while the code for move selection and execution appears right after that, in lines 30–36. The outer selection in line 12 remains the same and is used for defining both types of moves:

```
selectMax (g in Guests, p in Periods) (S.violations(boat[g,p])) {...}
```

```

1 int tabu[Guests,Periods,Hosts] = -1;
2 int it = 1; int tbl = 3; int tblMin = 2; int tblMax = 10; int best = violations;
3
4 Solution solution(m);
5 int      nonImprovingSteps = 0;
6 int      maxNonImproving = 100;
7 int      restartFreq = 1000;
8 MinNeighborSelector N();
9
10 while (violations > 0) {
11     int old = violations;
12     selectMax (g in Guests, p in Periods) (S.violations(boat[g,p])) {
13         selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
14             tabu[g,p,h] <= it || delta + violations < best) (delta)
15         neighbor(delta,N) {
16             tabu[g,p,boat[g,p]] = it + tbl;
17             boat[g,p] := h;
18         }
19
20         selectMin (g1 in Guests, delta = S.getSwapDelta(boat[g,p],boat[g1,p]):
21             (tabu[g,p,boat[g1,p]] <= it && tabu[g1,p,boat[g,p]] <= it)
22             || delta + violations < best) (delta)
23         neighbor(delta,N) {
24             tabu[g,p,boat[g,p]] = it + tbl;
25             tabu[g1,p,boat[g1,p]] = it + tbl;
26             boat[g,p] := boat[g1,p];
27         }
28     }
29
30     if (N.hasMove()) {
31         call (N.getMove());
32         if (violations < old && tbl > tblMin)
33             tbl--;
34         if (violations >= old && tbl < tblMax)
35             tbl++;
36     }

```

Statement 19.10: CBLS Model for the Progressive Party Problem (Part 1/2)

The assignment move we have already seen is defined in lines 12–18.

```
selectMin (h in Hosts, delta = S.getAssignDelta(boat[g,p],h):
    tabu[g,p,h] <= it || delta + violations < best) (delta)
neighbor(delta,N) {
    tabu[g,p,boat[g,p]] = it + tbl;
    boat[g,p] := h;
}
```

The same selector is used as in previous versions of the code. The only difference is that now the main body is enclosed in a neighbor and inserted to neighbor selector N. The move's code is just recorded without being executed; it will only be executed, if the move is actually chosen by the selector N. Swap moves can be seamlessly added to the neighbor selector in a similar fashion (lines 20–27)

```
selectMin (g1 in Guests, delta = S.getSwapDelta(boat[g,p],boat[g1,p]):
    (tabu[g,p,boat[g1,p]] <= it && tabu[g1,p,boat[g,p]] <= it)
    || delta + violations < best) (delta)
neighbor(delta,N) {
    tabu[g,p,boat[g,p]] = it + tbl;
    tabu[g1,p,boat[g1,p]] = it + tbl;
    boat[g,p] := boat[g1,p];
}
```

In defining the move, the first three lines simply select a guest *g1* to swap boat assignments with guest *g* in the selected period. The selection is made so that the number of violations is minimized and the new assignment of both guests is not tabu:

```
tabu[g,p,boat[g1,p]] <= it
tabu[g1,p,boat[g,p]] <= it
```

The aspiration criterion is also included in the selection. The main body, which is stored in the neighbor selector, is then straightforward. It first updates the tabu status of the current assignment, and then performs the actual swap:

```
tabu[g,p,boat[g,p]] = it + tbl;  
tabu[g1,p,boat[g1,p]] = it + tbl;  
boat[g,p] := boat[g1,p];
```

Once the moves have been updated into the neighbor selector, it suffices to select the one that leads to the smallest number of violations and perform it. This is done in lines 30–36. The first two lines query the selector to retrieve the move with the fewest violations and execute the corresponding closure with the `call` instruction:

```
if (N.hasMove())  
    call (N.getMove());
```

Note that, if one decided to extend the neighborhood even further, this would simply require including the code for the new moves and adding it to the same neighbor selector through a neighbor construct. After the move takes place, the next four lines update the tabu list length, in exactly the same way as in previous version of the search:

```
if (violations < old && tbl > tblMin)  
    tbl--;  
if (violations >= old && tbl < tblMax)  
    tbl++;
```

The remainder of the complete version of the search procedure is identical to the version that used restarts, and is shown on Statement [19.11](#) for the sake of completeness.

```
1  if (violations < best) {
2    best = violations;
3    solution = new Solution(m);
4    nonImprovingSteps = 0;
5  }
6  else if (nonImprovingSteps == maxNonImproving) {
7    solution.restore();
8    nonImprovingSteps = 0;
9  }
10 else
11   nonImprovingSteps++;
12
13 if ((it % restartFreq == 0) && (best > 0)) {
14   with delay(m)
15     forall (g in Guests, p in Periods)
16       boat[g,p] := distr.get();
17   best = violations;
18   solution = new Solution(m);
19 }
20 it++;
21 }
```

Statement 19.11: CBLS Model for the Progressive Party Problem (Part 2/2)

Chapter 20

CBLS User Extensions

This chapter explains how to extend some of the basic modeling structures of COMET, in order to deal with particular needs of different applications. COMET offers the possibility for user-defined invariants, constraints and objective functions. The most important aspects of this functionality are provided in this chapter. More advanced examples can be found in Chapter [21](#), which contains interesting local search applications.

20.1 User-Defined Constraints

In this section:

- We describe how to write user-defined constraints
- We give an example in the context of the queens problem

20.1.1 User-Defined AllDistinct Constraint

There are various reasons for resorting to user-defined constraints. For instance, one might want to define a constraint to simplify the description of a model, that would be more complicated to state using already defined constraints. Or, one might want to slightly modify the semantics, to better guide the search. We now present an example motivated by the second reason.

The queen problem solution presented earlier in the tutorial used the `AllDifferent<LS>` constraint. According to the semantics of this constraint, each individual variable has a single violation, if there is at least one other variable with the same value, and no violations otherwise. However, for the queens problem, it makes better sense to have value-based violations: the number of violations of each variable is equal to the number of other variables with the same value. This would make the search focus more on queens with many conflicts. Statement 20.1 shows a value-based version of the `AllDifferent<LS>` constraint, the `AllDistinct` constraint.

User-defined constraints should extend the `UserConstraint<LS>` class, which implements the `Constraint<LS>` interface, as reflected in the first line of the `AllDistinct` class definition:

```
class AllDistinct extends UserConstraint<LS> {...}
```

```

class AllDistinct extends UserConstraint<LS> {
  Solver<LS> m; var{int}[] a; range R; range V;
  dict{var{int} -> int} map;
  bool posted;
  var{int}[] valueCount;
  var{int}[] valueViolations;
  var{int}[] variableViolations;
  var{int} totalViolations;
  var{bool} isConstraintTrue;

  AllDistinct(var{int}[] _a) : UserConstraint<LS>(_a.getLocalSolver()) {
    m = _a.getLocalSolver(); a = _a; R = _a.getRange(); posted = false;
    post();
  }

  void post() {
    if (!posted) {
      map = new dict{var{int} -> int}();
      forall (i in R)
        map{a[i]} = i;
      valueCount = count(a);
      V = valueCount.getRange();
      valueViolations = new var{int}[i in V](m) <- max(0,valueCount[i]-1);
      variableViolations = new var{int}[i in R](m) <- valueViolations[a[i]];
      totalViolations = new var{int}(m) <- sum(i in V) valueViolations[i];
      isConstraintTrue = new var{bool}(m) <- (totalViolations == 0);
      posted = true;
    }
  }

  Solver<LS> getLocalSolver() { return m; }
  var{int}[] getVariables() { return a; }
  var{bool} isTrue() { return isConstraintTrue; }
  var{int} violations() { return totalViolations; }
  var{int} violations(var{int} x) { return variableViolations[map{x}]; }
  int getSwapDelta(var{int} x,var{int} y) { return 0; }
  int getAssignDelta(var{int} x,int d) {
    return (x == d) ? 0 : (valueCount[d] >= 1) - (valueCount[x]>= 2); }
}

```

Statement 20.1: User-Defined AllDistinct Constraint

Before describing the class methods, we first give a brief description of the class members:

- **m**: the constraint's local solver
- **a**: the array of incremental variables
- **R**: the range of array **a**
- **V**: the domain of the variables in **a**
- **map{x}**: position of variable **x** in the array **a**
- **posted**: indicates if the constraint has been posted
- **valueCount[v]**: number of variables equal to **v**
- **valueViolations[v]**: violations for value **v**
- **variableViolations[i]**: violations of variable **a[i]**
- **totalViolations**: total number of constraint violations
- **isConstraintTrue**: indicates whether the constraint is satisfied

The constructor of `AllDistinct` extends the constructor of class `UserConstraint<LS>` and only requires a single argument: an array of incremental variables.

```
AllDistinct(var{int}[] _a) : UserConstraint<LS>(_a.getLocalSolver()) {
    m = _a.getLocalSolver();
    a = _a;
    R = _a.getRange();
    posted = false;
    post();
}
```

It first initializes the local solver, the array of incremental variables and the range of variables, from the input array `_a`. Then it sets the `posted` variable to `false` and invokes the `post` method, which initializes the core variables of the class using invariants. Initialization takes place only if variable `posted` is false and, after that, `posted` is set to `true`, to ensure that initialization only takes place once.

We now describe the main functions of the class, starting with the `post` method. Note that one does not have to extend all the functions supported by the `UserConstraint<LS>` class: it suffices

to implement the parts that are relevant to the particular model.

The `post` method, shown on Statement 20.1, is called every time a new constraint object is created. Its main body is only executed if `posted` is false. It starts by initializing the dictionary `map` and associating each incremental variable with its position in the array:

```
map = new dict{var{int} -> int}();
forall (i in R)
  map{a[i]} = i;
```

This map will get handy in the implementation of methods dealing with variable violations. It will basically allow to access the violations of any variable `a[i]`, without having to know its position in the array `a` inside the constraint object.

Then, the combinatorial invariant `count` is used to incrementally maintain, in an array `valueCount`, the number of occurrences of each value in the array of incremental variables. The range of `valueCount` determines the domain of values `V` of the variables forming the `AllDistinct` constraint:

```
valueCount = count(a);
V = valueCount.getRange();
```

The violation-related variables of the class are maintained as invariants defined on top of `valueCount`. The array `valueViolations` stores the number of violations for the values in `V`. For each value `v`, the number of violations is 0, if the value appears at most once. Otherwise, it is equal to the number of extra occurrences:

```
valueViolations = new var{int}[i in V](m) <- max(0,valueCount[i] - 1);
```

In order to have value-based violations, the number of violations of each individual variable is maintained incrementally, as the value violations of its current assignment:

```
variableViolations = new var{int}[i in R](m) <- valueViolations[a[i]];
```

The total number of violations for the constraint is simply the sum of value violations over all values in the domain V ; these are maintained incrementally with the following numerical invariant:

```
totalViolations = new var{int}(m) <- sum(i in V) valueViolations[i];
```

Finally, the constraint is satisfied, if the total violations is 0, and this is maintained with the boolean variable `isConstraintTrue`

```
isConstraintTrue = new var{bool}(m) <- (totalViolations == 0);
```

Most of the remaining methods are rather straightforward, except for `violations(var)` and `getAssignDelta`. Also, note that the `getSwapDelta` method returns 0, since swaps have no effect on alldifferent constraints. This is the implementation of method `violations(var)`:

```
var{int} violations(var{int} x) { return variableViolations[map{x}]; }
```

Given an incremental variable x , the method uses the dictionary `map`, to find the position `map{x}` of the variable in the internal array of variables; it then returns the variable violations of the variable in that position. We conclude this description of the `AllDistinct` constraint with the method that provides differentiation, `getAssignDelta(x,d)`:

```
int getAssignDelta(var{int} x,int d) {
  return (x == d) ? 0 : (valueCount[d] >= 1) - (valueCount[x]>= 2); }
```

The method computes the difference in violations by assigning incremental variable x to value d in the domain V . If d is the current value of x , the result is 0. Otherwise, it is determined whether this assignment will generate an additional violation for value d (if `valueCount[d] >= 1`) and whether it will reduce the violations for the current value of x by 1 (if `valueCount[x]>= 2`).

20.1.2 The Queens Problem using the AllDistinct constraint

Statement [20.2](#) shows how to use the user-defined `AllDistinct` constraint for the Queens Problem. Assume that the definition is contained in the file `alldistinct.co`. Then, in order to use the constraint, it suffices to add a single instruction telling `COMET` to include that file:

```
include "alldistinct";
```

The `AllDistinct` is the first constraint posted to the constraint system `S`:

```
S.post(AllDistinct(queen));
```

```

import cotls;
include "alldistinct";

Solver<LS> m();
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);
var{int} queen[i in Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);
S.post(AllDistinct(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
    selectMax (q in Size) (S.violations(queen[q]))
        selectMin (v in Size) (S.getAssignDelta(queen[q],v))
            queen[q] := v;
    it++;
}

```

Statement 20.2: CBLS Model for the Queens Problem Using a User-Defined Constraint

20.2 User-Defined Functions

In this section:

- we model and solve a steel mill slab problem
- we show how to use COMET invariants to maintain feasible moves
- we show how to implement a user-defined objective function and its differentiation

20.2.1 The Steel Mill Slab Problem

We illustrate user-defined objective functions through an example: we present a CBLS model for a problem inspired by the steel industry: the Steel Mill Slab Problem. We have already seen a CP model for this problem in Section 12.8, but the presentation of the local search approach does not depend on that section. To gain some intuition on the problem, let's give a simplified paradigm of steel mill production.

Steel is produced by casting molten iron into slabs. There is only a limited number of possible capacities for the slabs produced in the mill. The steel mill has to process a number of submitted orders, each characterized by its color and its size. The color of an order corresponds to the route it has to follow through the mill, in order to be processed.

Given a set of input orders, the problem is then to assign them to slabs, so that the orders are processed using the minimum amount of steel. The amount of steel used is proportional to the sum of capacities of all slabs produced. The assignment of orders to slabs has to satisfy two conditions:

- The total size of the orders assigned to a slab cannot exceed the largest slab capacity
- Each slab can contain at most two colors (Color Constraints)

The rationale behind color constraints is the fact that it is expensive to cut up slabs, so that different colored orders are sent to different production areas of the mill.

A solution is formed by assigning to each slab a set of orders grouped together, so that the constraints are satisfied and all orders are assigned to slabs. Once a group of orders has been assigned to a slab, the slab's capacity is chosen among the available capacity options that are large enough to accommodate the whole group. Note that, if the total size of the orders assigned to a slab is smaller than the slab's capacity, the unused capacity corresponds to a steel loss. Therefore, the problem is equivalent to minimizing the total steel loss over all slabs produced.

A small instance of this problem is given next. The set of possible slab capacities is $\{7, 10, 12\}$. There are five orders with colors and sizes given in the following table.

| | | | | | |
|-------|---|---|---|---|---|
| Order | 1 | 2 | 3 | 4 | 5 |
| Size | 5 | 8 | 3 | 4 | 6 |
| Color | 1 | 3 | 2 | 1 | 2 |

Here is a solution to this instance, using three slabs:

| | | | |
|---------------|-------|----|---|
| Slab Capacity | 12 | 10 | 7 |
| Orders | 1,3,4 | 2 | 5 |
| Load | 12 | 8 | 6 |
| Loss | 0 | 2 | 1 |

In this solution, the total loss is 3 because there is a loss of 2 in the second slab and a loss of 1 in the third slab. Statement 20.3 gives the COMET code for a Constraint-Based Local Search solution to the problem. The algorithm consists of two phases:

- Modeling and Initialization: Read and preprocess the data, and assign one order to each slab satisfying capacity and color constraints
- Search: In each iteration, move an order from a slab with a loss into another slab, so that the loss decrease is maximized, while maintaining feasibility. Repeat, until total loss reaches zero.

Model and Initialization The implementation uses invariants to maintain the set of feasible moves and the set of slabs for which there is steel loss. The best move is chosen by differentiating a dedicated user-defined function, specifically designed for this problem.

The first lines of the code import the local search model and include the file `steelobjective.co`, that contains the definition of the user-defined objective function `SteelObjective`. The following lines define the problem's data. Ranges `Slabs`, `Orders` and `Colors` are self-explanatory. The array `capacity` stores the available slab capacities. The arrays `color` and `size` store the color and the size of each order. For simplicity, the process of reading the input data is omitted, and summarized with the single line:

```
// readData();
```

An important observation is that, since the goal is to minimize the total loss, the capacity chosen for each slab has to be the minimum available capacity that can accommodate its load. Since this only depends on the value of the load, and since the largest available capacity cannot be exceeded, it makes sense to precompute the loss corresponding to each possible value for a slab's load.

```

import cotls;
include "steelobjective";
Solver<LS> m();
range Slabs; range Orders; range Colors; range Capacities;
int[] capacity;
int[] color;
int[] size;
// readData();

int maxCapacity = max(i in Capacities)(capacity[i]);
int loss[i in 0..maxCapacity] =
    min(c in Capacities: capacity[c] >= i) (capacity[c] - i);
loss[0] = 0;

RandomPermutation perm(Orders);
var{int} slabOfOrder[o in Orders](m,Orders) := perm.get();

var{set{int}} ordersInSlab[s in Slabs](m)
    <- filter(o in Orders) (slabOfOrder[o] == s);
var{int} load[s in Slabs](m)
    <- sum(o in ordersInSlab[s]) size[o];
var{int} slabLoss[s in Slabs](m)
    <- loss[load[s]];
var{set{int}} colorsInSlab[s in Slabs](m)
    <- collect(o in ordersInSlab[s]) (color[o]);
var{int} nbColorsInSlab[s in Slabs](m)
    <- card(colorsInSlab[s]);
var{set{int}} possibleSlabs[o in Orders](m)
    <- filter(s in Slabs) (s != slabOfOrder[o]
        && load[s] + size[o] <= maxCapacity
        && (nbColorsInSlab[s] < 2
            || member(color[o],colorsInSlab[s])));

Function<LS> SteelObj = SteelObjective(m,slabOfOrder,size,load,loss);
m.close();

while(SteelObj.evaluation() > 0)
    selectMax[2](s in Slabs)(slabLoss[s])
    selectMin(o in ordersInSlab[s], sNew in possibleSlabs[o])
        (SteelObj.getAssignDelta(slabOfOrder[o],sNew))
    slabOfOrder[o] := sNew;

```

Statement 20.3: CBLS Model for the Steel Mill Slab Problem

This is performed in the following block of code, which generates an array `loss` that gives the steel loss corresponding to each possible load value.

```
int maxCapacity = max(i in Capacities)(capacity[i]);
int loss[i in 0..maxCapacity] =
    min(c in Capacities: capacity[c] >= i) (capacity[c] - i);
loss[0] = 0;
```

More precisely, if `s` is the total size of the orders assigned to a slab, then `loss[s]` is the loss incurred by choosing the minimum slab capacity that can accommodate these orders. Of course, `loss[0]` is 0, since it corresponds to an empty order. For example, if there are three possible capacities, {5, 8, 10}, the computed array will be:

```
loss = [0, 4, 3, 2, 1, 0, 2, 1, 0, 1, 0]
```

After the precomputation phase, the model declares the basic decision variables used in the search, `slabOfOrder[o]`, that represent the slab where each order `o` is placed. Initialization is performed through a random permutation of the orders.

```
RandomPermutation perm(Orders);
var{int} slabOfOrder[o in Orders](m,Orders) := perm.get();
```

All other variables used depend on these decision variables and are maintained through invariants. This is a summary of their semantics:

- `ordersInSlab[s]`: the set of orders placed into slab `s`
- `load[s]`: the total size of items placed in slab `s`
- `slabLoss[s]`: the loss of slab `s`
- `colorsInSlab[s]`: the set of colors present in slab `s`
- `nbColorsInSlab[s]`: the number of different colors present in slab `s`
- `possibleSlabs[o]`: the set of slabs, where `o` can be placed, still maintaining feasibility

We now give a more detailed description, also explaining the different types of invariants used. The set of orders placed in each slab s is maintained incrementally using a **filter** invariant, that selects the orders o , for which `slabOfOrder[o]` is equal to s :

```
var{set{int}} ordersInSlab[s in Slabs](m)
    <- filter (o in Orders) (slabOfOrder[o] == s);
```

In order to maintain the load of each slab, it is sufficient to use a **sum** invariant over the set of orders placed in the slab:

```
var{int} load[s in Slabs](m) <- sum(o in ordersInSlab[s]) size[o];
```

Once this is available, an element invariant can be used to maintain the loss of each slab s . We simply need to index the precomputed array `loss` with the incremental variable `load[s]`:

```
var{int} slabLoss[s in Slabs](m) <- loss[load[s]];
```

The set of colors present in slab s can be maintained with a **collect** invariant, that collects the colors of the orders assigned to the slab:

```
var{set{int}} colorsInSlab[s in Slabs](m)
    <- collect(o in ordersInSlab[s]) (color[o]);
```

The number of colors present in each slab is maintained incrementally through a simple cardinality invariant over the above set of colors.

```
var{int} nbColorsInSlab[s in Slabs](m) <- card(colorsInSlab[s]);
```

The most interesting invariant is `possibleSlabs[o]`, which maintains the feasible moves associated with an order `o`. It uses a **filter** invariant to select the slabs into which order `o` can be moved to (excluding its current slab), without violating the capacity and color constraints of the new slab. In other words, after adding `o` to the new slab, the total weight will not exceed the maximum slab capacity and the number of different colors on the slab will still be at most 2:

```
var{set{int}} possibleSlabs[o in Orders](m) <- filter(s in Slabs)
    (s != slabOfOrder[o] && load[s] + size[o] <= maxCapacity
    && (nbColorsInSlab[s] < 2 || member(color[o], colorsInSlab[s])));
```

The color constraint is satisfied, if either the current number of colors is smaller than two, or the color of order `o` is already present in the new slab. Finally, the code declares the objective function, `SteelObj`, used in the model and closes the model, to ensure that the invariants are correctly maintained. The objective function is an object of the class `SteelObjective`, a user-defined function that represents the total steel loss, and whose implementation we describe shortly.

```
Function<LS> SteelObj = SteelObjective(m, slabOfOrder, size, load, loss);
m.close();
```

Search Once the model and the objective are in place, the search can be stated very concisely. The search keeps running until the evaluation returned by the objective function, i.e., the total steel loss, is zero. The search is directed by the following two nested selectors:

```
selectMax[2](s in Slabs)(slabLoss[s])
    selectMin(o in ordersInSlab[s], sNew in possibleSlabs[o])
        (SteelObj.getAssignDelta(slabOfOrder[o], sNew))
```

The outer selector randomly selects a slab `s` among the two slabs with the highest loss. Then, the inner selector chooses an order `o` assigned to `s` and a new slab `sNew`, among the set of possible moves `possibleSlabs[o]`, so that moving `o` to `sNew` leads to the largest reduction to the objective function. The main body of the selector simply makes the assignment of order `o` into slab `sNew`.

```
slabOfOrder[o] := sNew;
```

20.2.2 User-Defined Function for Steel Mill Slab

This section concludes with a description of the user-defined objective function defined for the Steel Mill Slab problem. User-defined functions can be defined, by simply extending the class `UserFunction<LS>` and implementing the necessary methods. For the purposes of our implementation, we only need to implement the constructor and the methods `evaluation` and `getAssignDelta`. Statement 20.4 contains the complete implementation of the user-defined function `SteelObjective`.

Most members of the `SteelObjective` class are identical to the corresponding variables in the main code to solve the Steel Mill problem, and take their values through the arguments passed to the constructor. This is the case for class members `m`, `Slabs`, `Orders`, `slabOfOrder`, `size`, `load`, `loss` and `slabLoss`. Note though that, since variables may be stored in a different order within the `SteelObjective` class, compared to the main code, the class has its own version of the invariant maintaining `slabLoss`. This is taken care of by the constructor, which also initializes the other two members of the class, `totalLoss` and `orderSize`.

The first lines of the constructor are straightforward: they simply copy the input data to the corresponding class members and use the input arrays `_load` and `_slabOfOrder` to determine the ranges `Slabs` and `Orders`, respectively:

```
m          = _m;
Slabs      = _load.getRange();
Orders     = _slabOfOrder.getRange();
slabOfOrder = _slabOfOrder;
size       = _size;
load       = _load;
loss       = _loss;
```

The following line initializes the local copy of the array `slabLoss`, that represents the loss in each slab. This is done through an element invariant combining the local array `load`, that represents slab load, and the precomputed array `loss`, given as input to the constructor:

```
slabLoss = new var{int}[s in Slabs](m) <- loss[load[s]];
```

```

class SteelObjective extends UserFunction<LS> {
  Solver<LS>      m;
  range          Slabs;
  range          Orders;
  var{int}[]     slabOfOrder;
  int []         size;
  var{int}[]     load;
  int []         loss;
  var{int}[]     slabLoss;
  var{int}       totalLoss;
  dict{var{int}->int} orderSize;

  SteelObjective(Solver<LS> _m,var{int}[] _slabOfOrder,int[] _size,
    var{int}[] _load,int[] _loss): UserFunction<LS>(_m){
    m      = _m;
    Slabs  = _load.getRange();
    Orders = _slabOfOrder.getRange();
    slabOfOrder = _slabOfOrder;
    size     = _size;
    load     = _load;
    loss     = _loss;
    slabLoss = new var{int}[s in Slabs](m) <- loss[load[s]];
    totalLoss = new var{int}(m) <- sum(i in Slabs) slabLoss[i];
    orderSize = new dict{var{int}->int}();
    forall (o in Orders)
      orderSize{slabOfOrder[o]} = size[o];
  }

  var{int}[] getVariables() { return slabOfOrder; }
  var{int}   evaluation()   { return totalLoss; }

  int getAssignDelta(var{int} x,int s) {
    if (x == s)
      return 0;
    int w = orderSize{x};
    if (load[s] + w > loss.getRange().getUp())
      return System.getMAXINT();
    return loss[load[x]-w] + loss[load[s]+w] - slabLoss[x] - slabLoss[s];
  }
}

```

Statement 20.4: User-Defined SteelObjective Function for the Steel Mill Slab Problem

The incremental variable `totalLoss` represents the value of the objective function. It is equal to the sum of the loss over all slabs, and is maintained through a **sum** invariant over the array `slabLoss`:

```
totalLoss = new var{int}(m) <- sum(i in Slabs) slabLoss[i];
```

The most interesting variable of the class is the dictionary `orderSize`, which maps every incremental variable `slabOfOrder[o]` to the size of the order this incremental variable corresponds to. The use of this dictionary will become clear in the implementation of method `getAssignDelta` and its initialization is performed as follows:

```
orderSize = new dict{var{int}->int}();
forall (o in Orders)
  orderSize{slabOfOrder[o]} = size[o];
```

The class implements three methods, aside from the constructor. Method `getVariables` returns the array `slabOfOrder` of incremental variables used in the objective function:

```
var{int}[] getVariables() { return slabOfOrder; }
```

Method `evaluation` returns the incremental variable `totalLoss` corresponding to the total steel loss in the slabs.

```
var{int} evaluation() { return totalLoss; }
```

The main method of the class is `getAssignDelta(x,s)`, which returns the difference to the objective function's value, if incremental variable `x` takes value `s`. The method returns 0, if the new slab `s` is equal to the current slab. Otherwise, it first computes the size `w` of the order corresponding to incremental variable `x`, using the dictionary `orderSize`:

```
int w = orderSize{x};
```

Note that `x` corresponds to the slab assignment of some order `o` (`x = slabOfOrder[o]`), which is not given directly as an argument to the method; this explains the need for a dictionary.

The next two lines check whether the load of the new slab `s` would exceed the maximum capacity, if an order of size `w` were added to it. Since the precomputed array `loss` is indexed by load value, and precomputation is only performed for values at most equal to the maximum slab capacity, an attempt to compute the loss for a value exceeding the maximum capacity would cause an out-of-bounds error. If this is the case, the method returns the system's maximum integer to indicate that the move would lead to an infeasible assignment.

```
if (load[s] + w > loss.getRange().getUp())
    return System.getMAXINT();
```

Of course, this cannot be the case in the current implementation, since the invariant maintaining `possibleSlabs` (defined in Statement 20.3), only contains feasible moves, which don't allow to exceed the maximum slab capacity. Note how the instructions `getRange` and `getUp` are used to compute the maximum value that can index the array `loss`.

If no special case is encountered, the increase to the objective function's evaluation, if variable `x` is assigned to slab `s`, is computed in constant time, as follows:

```
return loss[load[x]-w] + loss[load[s]+w] - slabLoss[x] - slabLoss[s];
```

This line first computes the new value for the steel loss of the two slabs `x` and `s` involved in the move, after removing an order of size `w` from slab `x` and adding this size to the load of slab `s`. It then subtracts the current steel loss of these two slabs, which is computed by adding `slabLoss[x]` and `slabLoss[s]`.

Chapter 21

CBLS Applications

This chapter contains some more advanced applications than those shown in previous chapters. The goal here is to combine different ideas presented so far in the tutorial, and to present examples that are more challenging, both in terms of modeling and in terms of search.

21.1 The Warehouse Location Problem

This section presents an iterated tabu search solution to the uncapacitated Warehouse Location Problem. The solution takes advantage of some of the advanced objective functions supported by COMET and also uses set invariants, in order to incrementally maintain the neighborhood.

In this problem, we are given a set of m stores, S , and a set of n warehouses, W , that can serve the stores. Initially, all warehouses are closed, and we are asked to open a subset of the warehouses, so that all stores are served and the total costs involved are minimized. Opening a warehouse w entails a fixed cost of f_w . Each store must be served by exactly one open warehouse. This involves a transportation cost from the warehouse to the store. More precisely, if store s is served by warehouse w , the corresponding cost is $c_{w,s}$.

We only consider the uncapacitated version of the problem, in which warehouses can serve an unlimited number of stores. The goal is to compute a subset of warehouses to open and an assignment of stores to open warehouses, so that the total fixed and transportation costs are minimized.

21.1.1 Modeling Warehouse Location

Recall that there is no limit in the number of stores each warehouse can serve. This means that, once the set of open warehouse is determined, the optimal assignment of warehouses to stores is to serve each store by its closest warehouse. Therefore, each solution can be represented by a set of (open) warehouses:

$$W_o = \{w_1, w_2, \dots, w_t\}, \quad w_i \in W, \quad t \leq n$$

This leads to a rather simple model: an array **open**, indexed by the range of warehouses and indicating whether a warehouse is open, suffices to model a solution. Because of this, the search procedure is based on “flip” operations, that change the status of a warehouse from closed to open, and vice versa.

The complexity then lies in incrementally maintaining the value of the objective function over changes to the set of open warehouses. This is not trivial, given that a change to the set of open warehouses may lead to a different optimal assignment of stores to warehouses. COMET takes care of this complexity by implementing the appropriate objective functions.

WarehouseLocation Class Statement 21.1 shows the statement of the class used in solving the problem, class `WarehouseLocation`, which contains all the information necessary for the modeling and the search involved in this section's approach. The complete implementation of the class will be presented after an overview of the class members. The class first declares the variables used to store the data that define an instance of the problem:

- **Warehouses:** the range of warehouses
- **Stores:** the range of stores
- `fcost[w in Warehouses]`: the fixed cost for warehouse `w`
- `tcost[w in Warehouses,s in Stores]`: the transportation cost from warehouse `w` to store `s`

The following group of variables declare a local solver `m`, a uniform distribution `odistr` and a boolean array `open` to indicate open warehouses.

```
Solver<LS> m;
UniformDistribution odistr;
var{bool}[] open;
```

The most important part of the model are the objective functions declared next, which incrementally maintain the costs corresponding to any subset of open warehouses: the total transportation cost, `travelCost`, the total fixed cost, `fixedCost`, and the sum of these two, `cost`:

```
FloatFunctionOverBool<LS> travelCost;
FloatFunctionOverBool<LS> fixedCost;
FloatFunctionOverBool<LS> cost;
```

```

class WarehouseLocation {
    range      Warehouses;
    range      Stores;
    float []    fcost;
    float [,]   tcost;

    Solver<LS>   m;
    UniformDistribution odistr;
    var{bool}[] open;

    FloatFunctionOverBool<LS> travelCost;
    FloatFunctionOverBool<LS> fixedCost;
    FloatFunctionOverBool<LS> cost;

    var{bool}[] tabu;
    Counter     it;
    int         nbDiversifications;
    int         nbIterations;

    var{float}[] gain;
    var{set{int}} nonTabuWH;
    var{set{int}} bestFlip;
    var{set{int}} openSet;
    var{float}   obj;
    Solution     bestSolution;
    float        bestCost;

    WarehouseLocation(range wr, range sr, float[] fc,float[,] tc,int nbd,int nbi);
    void search();
    void localSearch();
    void updateBestSolution();
    void reset();
}

```

Statement 21.1: CBLS Model for the Warehouse Location Problem: I. The Class Statement

Apart from the class constructor, the remainder of the class consists of variables, invariants and functions for controlling the search and managing solutions. The variables used to control the search include an array `tabu`, indexed on `Warehouses` and used for the tabu list, a counter `it`, and two integer parameters for the tabu heuristic: `nbDiversifications` and `nbIterations`

```
var{bool}[] tabu;  
Counter it;  
int nbDiversifications;  
int nbIterations;
```

The search uses a number of useful invariants defined next:

- `gain[w]`: cost increase by flipping warehouse `w` (< 0 for improving flips)
- `nonTabuWH`: the set of non-tabu warehouses
- `bestFlip`: the set of warehouses, whose flipping produces the best improvement
- `openSet`: the set of open warehouses
- `obj`: the current cost

The class also declares a `Solution` object, `bestSolution`, to store the best found solution, and a float variable `bestCost` for the corresponding score. Finally, it declares the methods `search()`, `localSearch()`, `updateBestSolution()` and `reset()` that perform the search procedure, whose implementation we'll describe after reviewing the constructor.

21.1.2 WarehouseLocation Class Constructor

The complete code for the constructor is shown on Statement 21.2. Due to space considerations, we only give some highlights of the constructor skipping more straightforward parts. (The code as shown on the tutorial does not handle a few special cases arising in the search, such as ensuring that at least one warehouse is open in any solution. These cases are covered in the COMET code accompanying the tutorial.)

After reading the data and allocating a local solver `m`, the constructor initializes the boolean array `open` using a uniform distribution `odistr` in the range 0..1, so that about half of the warehouses are open in the initial solution:

```
open = new var{bool}[Warehouses](m) := (odistr.get() == 1);
```

The lines that follow define the objective functions modeling the problem:

```
travelCost = new FloatSumMinCostFunctionOverBool<LS>(Warehouses,Stores,open,tcost)
fixedCost  = new FloatBoolSumObj<LS>(fcost,open);
cost       = fixedCost + travelCost;
```

Since these function are essential to this model, we explain them in a little more depth. As you can see in the class statement, all three objective functions conform to the Interface `FloatFunctionOverBool<LS>`, which means that they can be queried using the same set of methods. However, each of them is an object of a different implementation of the interface.

The objective function `travelCost` is an object of the class `FloatSumMinCostFunctionOverBool<LS>`, whose semantics are described for the warehouse location example, but it can be of more general use. Given a boolean array `open`, defined in the range `Warehouses`, and a float matrix `tcost[Warehouses,Stores]`, the function:

```
FloatSumMinCostFunctionOverBool<LS>(Warehouses,Stores,open,tcost);
```

incrementally maintains a sum of floating point values, one for every entry in the range `Stores`. For a given `s` in that range, this sum is equal to the smallest value of `tcost[w,s]`, over all `w` with `open[w] = true`.

```

WarehouseLocation::WarehouseLocation(range wr,range sr,
                                     float [] fc,float[,] tc,int nbd,int nbi) {

    Warehouses = wr;
    Stores     = sr;
    fcost      = fc;
    tcost      = tc;

    m          = new Solver<LS>();
    odistr     = new UniformDistribution(0..1);
    open       = new var{bool}[Warehouses](m) := (odistr.get() == 1);

    travelCost = new FloatSumMinCostFunctionOverBool<LS>(Warehouses,Stores,open,tcost);
    fixedCost  = new FloatBoolSumObj<LS>(fcost,open);
    cost       = fixedCost + travelCost;

    tabu       = new var{bool}[Warehouses](m) := false;
    it         = new Counter(m,0);
    nbDiversifications = nbd;
    nbIterations   = nbi;

    gain       = new var{float}[w in Warehouses] = cost.flipDelta(open[w]);
    nonTabuWH  = new var{set{int}}(m) <- filter(w in Warehouses) (!tabu[w]);
    bestFlip   = new var{set{int}}(m) <- argMin(w in nonTabuWH) gain[w];
    openSet    = new var{set{int}}(m) <- filter(w in Warehouses) (open[w] == true);
    obj        = cost.value();

    bestSolution = new Solution(m);
    bestCost     = obj;

    m.close();
}

```

Statement 21.2: CBLS Model for the Warehouse Location Problem: II. Constructor and Model

The objective function `fixedCost` is an object of the class `FloatBoolSumObj<LS>`, which incrementally maintains the fixed cost of open warehouses. Again, it is simpler to describe the semantics in terms of the particular example. Given a boolean array `open` and a float array `fcost`, both defined on the same range, the function

```
FloatBoolSumObj<LS>(fcost,open);
```

incrementally maintains the sum of `fcost[w]`, over all `w`, for which `open[w] = true`.

Finally, the objective function `cost`, corresponding to the total cost is simply defined as the sum of the previous two objectives. This can be done without any problem, since both objective functions conform to the same interface.

After setting the initial tabu state to **false** for all warehouses and initializing the counter `it` to 0, the constructor initializes the invariants used in the search, which are now explained in more detail. The first invariant is an array indexed on `Warehouses` which, for every warehouse `w` in that range, holds the gain that can occur by flipping the corresponding decision variable's value. It uses the `flipDelta` method of objective functions defined over booleans, for example, functions implementing the interface `FloatFunctionOverBool`.

```
gain = new var{float}[w in Warehouses] = cost.flipDelta(open[w]);
```

The second invariant incrementally maintains the set of warehouses that are not tabu at any given step, and therefore can be used in the moves of that step. It is a simple **filter** set-invariant:

```
nonTabuWH = new var{set{int}}(m) <- filter(w in Warehouses) (!tabu[w]);
```

Using these two invariants, it is easy to maintain the set of warehouses that are not tabu and lead to the best (minimum) gain. This is another instance of the **argMin** invariant, that returns the set of warehouses `w` in `nonTabuWH` that have the smallest `gain[w]` (remember that negative gain corresponds to an improvement to the objective function). Since there may be multiple warehouses with the same gain, this is a set rather than a single value. Here is the definition of this invariant:

```
bestFlip = new var{set{int}}(m) <- argMin(w in nonTabuWH) gain[w];
```

The `openSet` invariant simply stores the set of open warehouses, through a **filter** operator:

```
openSet = new var{set{int}}(m) <- filter(w in Warehouses) (open[w] == true);
```

Before closing the model, the constructor initializes an invariant `obj` to hold the objective function value at any given point (using the `value` method of float objective functions), allocates a solution object for the currently best solution (which at this point is equal to the random initial set of open warehouses), and, finally, initializes variable `bestCost` to the cost of this initial random assignment.

```
obj = cost.value();  
bestSolution = new Solution(m);  
bestCost = obj;
```

21.1.3 Basic Tabu Search

In the rest of this section, we describe the local search algorithm for warehouse location, starting with the basic tabu search component and then showing how to incorporate it into an iterated tabu search scheme. The basic tabu search scheme is implemented by method `localSearch()`, as shown in Statement 21.3. The iterative scheme presented later simply runs this basic algorithm for a number of iterations, i.e., each iteration corresponds to a single run of method `localSearch()`.

The local search neighborhood consists of flipping moves, that switch the status of a warehouse from open to closed, and vice versa. This is performed by simply changing the value of the corresponding entry of the `open` array. The heuristic uses a fixed-length tabu list, to avoid flipping the same warehouses over and over again. At every step, it only considers flips that result into the highest decrease of the objective value, breaking ties randomly. If all flips increase the resulting cost, the algorithm performs a diversification, by randomly closing a number of open warehouses. Let us now get into the details of method `localSearch()`.

```

void WarehouseLocation::localSearch() {
    int      nonImprovingSteps = 0;
    float     bestCostIter = obj;
    Solution bestSolutionIter = new Solution(m);
    int      tabuLength = 3;

    while (nonImprovingSteps < 500) {
        select(w in bestFlip)
        if (gain[w] <= 0) {
            open[w] := !open[w];
            tabu[w] := true;
            when it@reaches[it + tabuLength]()
                tabu[w] := false;
        }
        else
            forall (i in 1..nbDiversifications)
                if (card(openSet) > 1)
                    select (w in openSet)
                        open[w] := false;
        it++;
        if (obj < bestCostIter && bestCostIter - obj > 0.00001) {
            bestCostIter = obj;
            bestSolutionIter = new Solution(m);
            nonImprovingSteps = 0;
        }
        else
            nonImprovingSteps++;
    }
    bestSolutionIter.restore();
}

```

Statement 21.3: CBLS Model for the Warehouse Location Problem: III. Tabu Search

The method assumes that a feasible assignment of variables has already been computed and uses it as a starting point for exploring the neighborhood. The method keeps track of the best solution found during its execution and its cost with the `Solution` variable `bestSolutionIter` and the float variable `bestCostIter`. It also keeps track of the number of consecutive steps without any improvement to the best solution, using the integer variable `nonImprovingSteps`. These variables, along with the tabu length are defined and initialized as follows:

```
int      nonImprovingSteps = 0;
float    bestCostIter = obj;
Solution bestSolutionIter = new Solution(m);
int      tabuLength = 3;
```

The main loop of the `localSearch` method keeps running, until the number of non improving steps reaches a fixed limit of 500. It consists of two parts: performing a move and updating the best solution found, if necessary. The first part starts by selecting a warehouse `w` that leads to the highest cost reduction. This is chosen from the set `bestFlip`, which is maintained as an invariant:

```
select (w in bestFlip) {...}
```

Then it takes two cases: If this has a negative gain, meaning that the cost is reduced, it flips the status of warehouse `w` and makes it tabu. Then, it installs a simple event that will remove its tabu status after `tabuLength` steps.

```
if (gain[w] <= 0) {
    open[w] := !open[w];
    tabu[w] := true;
    when it@reaches[it + tabuLength]()
        tabu[w] := false;
}
```

If the best gain is positive, which means that all solutions in the neighborhood are deteriorating, then a diversification step is performed, in which a number `nbDiversifications` of open warehouses are closed, selected from the incrementally maintained set `openSet`.

Before closing any warehouse, there is an additional check to ensure that there is at least one warehouse left open after doing so; this is because feasible solutions need to have at least one open warehouse.

```
forall (i in 1..nbDiversifications)
  if (card(openSet) > 1)
    select(w in openSet)
      open[w] := false;
```

Once a move or diversification is complete, we increase the counter by one and check if the new solution is improving the best solution found in the iteration. The improvement has to be by at least 0.00001, to account for numerical instabilities.

```
if (obj < bestCostIter && bestCostIter - obj > 0.00001)
```

In case of an improvement, `bestSolutionIter` and `bestCostIter` are updated and the counter of non improving steps is reset to 0:

```
bestCostIter = obj;
bestSolutionIter = new Solution(m);
nonImprovingSteps = 0;
```

This allows for an extra 500 steps in search of a new improvement. Otherwise, `nonImprovingSteps` is increased by one. When `nonImprovingSteps` exceeds 500, the method restores the assignment of open warehouses according to the best solution found so far in the iteration and returns.

```
bestSolutionIter.restore();
```

21.1.4 Iterated Tabu Search

The `search()` method implements the iterated tabu search algorithm, using the helper functions `updateBestSolution()` and `reset()`. All three methods are shown on Statement 21.4. Starting from the initial random solution, method `search()` performs a number of iterations given by `nbIterations`. Each iteration first calls method `localSearch()`; if there is an improvement, it calls `updateBestSolution()` to update the best solution; finally, method `reset()` generates a new random solution. Before returning, `search()` restores the best found solution after all iterations.

The `updateBestSolution()` method is almost identical to the part of the `localSearch()` that updates the best solution found during its run. Similarly, the main code for method `reset()` resembles some lines of the constructor. It simply goes through all warehouses, sets their open status randomly, using the distribution `odistr`, and resets the tabu status to `false` and the counter to 0. Note that the code for randomly setting warehouse status is enclosed within an `atomic` block, which postpones any propagation, until `open[w]` has been determined for all warehouses `w`.

```
with atomic(m)
  forall (w in Warehouses) {
    open[w] := (odistr.get() == 1);
    tabu[w] := false;
  }
```

Example of Use We conclude this section with a simple example of how to use the `WarehouseLocation` class. The commented line `readData()` summarizes the code to read instance data. The code solves a warehouse location instance using ten iterations and two diversifications (two warehouses closed) in each diversification step.

```
import cotls;
range wr;
range sr;
float fc[wr];
float tc[wr,sr];
//readData();
WarehouseLocation wh(wr,sr,fc,tc,2,10);
wh.search();
```

```

void WarehouseLocation::search() {
    forall (i in 1..nbIterations) {
        localSearch();
        updateBestSolution();
        reset();
    }
    bestSolution.restore();
}

void WarehouseLocation::updateBestSolution() {
    if (obj < bestCost && bestCost - obj > 0.00001) {
        bestCost = obj;
        bestSolution = new Solution(m);
    }
}

void WarehouseLocation::reset() {
    with atomic(m)
        forall (w in Warehouses) {
            open[w] := (odistr.get() == 1);
            tabu[w] := false;
        }
    it := 0;
}

```

Statement 21.4: CBLS Model for the Warehouse Location Problem: IV. Iterated Tabu Search

21.2 The Social Golfers Problem

This section illustrates user-defined constraints and invariants in COMET, through a local search approach to the Social Golfers Problem. This is a challenging optimization problem, and using efficient user-defined structures greatly simplifies modeling and searching.

The problem is described for a specific instance, although generalization is straightforward. It corresponds to the following situation faced by the coordinator of a local golfing club. In his club, there are 18 golfers, each of which plays golf once a week, and always in groups of three. This means that during each week, six groups of three golfers are formed. Since the club's golfers enjoy socializing, the coordinator would like to come up with an eight-week schedule for their games, so that any given pair of golfers plays in the same group at most once. In other words, each golfer is grouped with as many other golfers as possible.

Note that, for a given group size and a given number of groups, the difficulty of finding such a schedule increases with the number of weeks, since it becomes more difficult to satisfy the constraint that any two golfers can meet at most once.

21.2.1 Modeling Social Golfers

The model of Social Golfer Problem is given in Statement [21.5](#). The data of an instance is described by three ranges:

- **Weeks:** the range of weeks in the schedule
- **Groups:** the range of groups during each week
- **Slots:** the range of golfers within each group

The number of persons participating in the tournament is equal to the number of groups times the number of slots per group. This is computed with the aid of the `getUp()` method for ranges; a range `Golfers` of this size is created to identify the participating golfers:

```
range Golfers = 1..(Groups.getUp() * Slots.getUp());
```

The model also defines a tuple type `Position`, to better handle triples of the form `(week,group,slot)`:

```
tuple Position {int week; int group; int slot;}
```

```

import cotls;
Solver<LS> m();

range Weeks    = 1..8;
range Groups   = 1..6;
range Slots    = 1..3; // slots per group
range Golfers  = 1..(Groups.getUp() * Slots.getUp());

tuple Position {int week; int group; int slot;}

var{int} golfer[Weeks,Groups,Slots](m,Golfers);
init(Weeks,Golfers,Groups,Slots,golfer);
SocialTournament tourn(m,Weeks,Groups,Slots,Golfers,golfer);
m.post(tourn);

var{int} conflict[w in Weeks,g in Groups,s in Slots] = tourn.violations(golfer[w,g,s]);
var{int} violations = tourn.violations();
m.close();

function void init(range Weeks, range Golfers, range Groups,
                   range Slots, var{int}[,,] golfer) {
  forall (w in Weeks) {
    RandomPermutation golferPerm(Golfers);
    forall (g in Groups,s in Slots)
      golfer[w,g,s] := golferPerm.get();
  }
}

```

Statement 21.5: CBLS Model for the Social Golfer Problem: I. The Model

Any solution to a problem's instance needs to assign a golfer to every such triple defined in the instance. An incremental variable `golfer[w,g,s]` is introduced, to represent the golfer in the range `Golfers` that will play in week `w`, group `g` and slot `s` of group `g`, in other words, in the position characterized by the triple (w,g,s) .

```
var{int} golfer[Weeks,Groups,Slots](m,Golfers);
```

Note that, for any given week and group, the model introduces an arbitrary ordering of the slots, in order to avoid using set variables. Since the group size is fixed and known in advance, set variables would unnecessarily increase the model's complexity.

The constraint that each golfer plays exactly once per week is maintained as a hard constraint during the search. That is, the golfers playing during any given week form a permutation of the participants. The function `init` is called to randomly initialize the variables, so that, within each week, groups are formed using a different random permutation of the participants.

```
init(Weeks,Golfers,Groups,Slots,golfer);
```

This function is also used during the search, every time a restart is performed, and its COMET code appears in the last lines of Statement [21.5](#):

```
function void init(range Weeks, range Golfers, range Groups,
                  range Slots, var{int}[,,] golfer) {
  forall (w in Weeks) {
    RandomPermutation golferPerm(Golfers);
    forall (g in Groups,s in Slots)
      golfer[w,g,s] := golferPerm.get();
  }
}
```

Furthermore, any local move performed during the search has to maintain the following hard constraint: a move cannot lead to a schedule in which some golfer plays more than once in the same week.

The main constraint of this problem is that any pair of golfers cannot be grouped together more than once. This is managed through the user-defined constraint `SocialTournament` which extends class `UserConstraint<LS>`. The details of this user-defined constraint are explained later; for now, let us describe how violations are assigned by the constraint, in the context of the Social Golfers Problem.

Consider a decision variable `golfer[w,g,s]`, and let G the golfer placed in position (w,g,s) and $m(G)$ the set of golfers placed in the other slots of group g during week w . Then, the number of variable violations assigned to `golfer[w,g,s]` is equal to the number of golfers from $m(G)$ that golfer G meets in weeks other than w . The total number of constraint violations is given by the sum, over all pairs of golfers, of the number of times the pair is placed in the same group, after the first time this happens. For example, if two golfers g, h appear in the same group three times, this counts as two violations; if they are grouped together only once (or not at all), this corresponds to no violations for the pair.

An instance `tourn` of the `SocialTournament` constraint is declared and posted to the local solver with the following lines:

```
SocialTournament tourn(m,Weeks,Groups,Slots,Golfers,golfer);
m.post(tourn);
```

The violation assignment is accessible through the two forms of `violations()` methods available for constraint objects. Once the constraint is posted, the model uses these methods to define two more incremental variables to facilitate the search: An array `conflict[w,g,s]`, that stores the violations of the corresponding decision variables `golfer[w,g,s]`, and a variable `violations`, which is equal to the total violations of the `tourn` constraint.

21.2.2 The Search

The search procedure used is a standard tabu search with randomized tabu length and aspiration criteria, enhanced with a restarting component, that takes place, if no improvement occurs for a number of iterations. The neighborhood is very simple: its set of moves consists of swaps between two variables `golfer[w,g,s]` within the same week, from two different groups. The main challenge is to incrementally maintain violations after each swap, but this is taken care of by the user-defined constraint.

The complete implementation of the search is shown in Statement 21.6. Most of the code should look familiar, since similar versions of it have already been used in previous examples of the tutorial. The main loop of the search consists of a selector, that first selects the golfers to be swapped and then performs the swap and updates tabu status and restart information.

```

int tabu[Weeks,Golfers,Golfers] = -1;
UniformDistribution tabuDistr(4..20);
int best = violations;
int it = 0;
int nonImprovingSteps = 0;
int maxNonImproving = 15000;

while (violations > 0)
  selectMin(w in Weeks,
    g1 in Groups, s1 in Slots: conflict[w,g1,s1] > 0,
    g2 in Groups: g2 != g1, s2 in Slots,
    delta = tourn.getSwapDelta(golfer[w,g1,s1],golfer[w,g2,s2]) :
      tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] < it
      || violations + delta < best)
    (delta) {
      golfer[w,g1,s1] := golfer[w,g2,s2];
      int tabuLength = tabuDistr.get();
      tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] = it + tabuLength;
      tabu[w,golfer[w,g2,s2],golfer[w,g1,s1]] = it + tabuLength;
      if (violations < best) {
        best = violations;
        nonImprovingSteps = 0;
      }
      else {
        nonImprovingSteps++;
        if (nonImprovingSteps > maxNonImproving) {
          init(Weeks,Golfers,Groups,Slots,golfer);
          best = violations;
          nonImprovingSteps = 0;
        }
      }
      it++;
    }
}

```

Statement 21.6: CBLS Model for the Social Golfer Problem: II. Tabu Search with Aspiration Criteria and Restarts

The randomized decision for the tabu tenure is taken care by the following lines, which define a uniform distribution and query it to set the tabu length:

```
UniformDistribution tabuDistr(4..20);
...
int tabuLength = tabuDistr.get();
```

The remaining parts of the main block of the selector are similar to the code for the Progressive Party Problem, so we now focus on the selection part:

```
selectMin(w in Weeks,
    g1 in Groups, s1 in Slots: conflict[w,g1,s1] > 0,
    g2 in Groups: g2 != g1, s2 in Slots,
    delta = tourn.getSwapDelta(golfer[w,g1,s1],golfer[w,g2,s2]) :
        tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] < it
    || violations + delta < best)
(delta) {...}
```

The search selects the best swap which is not tabu between two variables `golfer[w,g1,s1]` and `golfer[w,g2,s2]` corresponding to two different groups, `g1` and `g2`, in the same week `w`. The first golfer is selected, so that it has at least one violation in the constraint. This is the case if `conflict[w,g1,s1] > 0`. The tabu list is defined over triples `(w,p1,p2)` corresponding to the swap of golfers `p1` and `p2` in week `w`. The move is tabu, if the following condition holds:

```
tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] < it
```

Once again, note the use of factorization. In the selector, `delta` is used to store the difference to the number of violations by performing the swap, which is given by the `getSwapDelta` method of constraint `tourn`. Then, this `delta` is used both for checking the aspiration criterion and as an argument of the selector. Note that the variable `best` stores the violations of the best solution found since the last restart.

21.2.3 The User-Defined Constraint `SocialTournament`

We now get into the most important aspects of the implementation. Namely, the implementation of the user-defined constraint `SocialTournament` and the user-defined invariant `Meet`, the main building block of the constraint.

The semantics of the constraint violations were mentioned when describing the model, and are summarized here. Let G the golfer placed in position (w, g, s) and $m(G)$ the set of golfers placed in the other slots of group g during week w . Then,

- the number of violations of variable `golfer[w,g,s]` is equal to the number of golfers from $m(G)$ that golfer G meets in weeks other than w
- the total number of constraint violations is given by the sum, over all pairs of golfers, of the number of times exceeding 1 that the pair is grouped together

The interface and constructor of the user-defined constraint `SocialTournament` is given in Statement 21.7. This is a summary of its class members:

- `Weeks, Groups, Slots, Golfers, golfer`: same description as in the model
- `meetings[g in Golfers, h in Golfers]`: number of times g and h play in the same group
- `position{x}`: Position (w, g, s) corresponding to incremental variable x
- `varViolations[w,g,s]`: violations of decision variable `golfer[w,g,s]`
- `violationDegree`: total number of violations

Implementation of `SocialTournament` Constraint The constructor of `SocialTournament` simply instantiates some of the class's variables from its parameters. The remaining class members are initialized by the `post` method. Apart from the constructor, the only methods necessary to implement are the `post`, `violations` (total and per variable) and `getSwapDelta` methods of the `UserConstraint<LS>` class. Since only swap moves are used in the search, there is no need to implement other methods, such as `getAssignDelta`. The implementation of these methods appears on Statement 21.8, with the exception of `violations()` which is defined inline on Statement 21.7:

```
var{int} violations() { return violationDegree; }
```

```

class SocialTournament extends UserConstraint<LS> {
    Solver<LS>          m;
    range              Weeks;
    range              Groups;
    range              Slots;
    range              Golfers;

    var{int}[,,]       golfer;
    var{int}[,,]       meetings;
    dict{var{int}->Position} position;
    var{int}[,,]       varViolations;
    var{int}            violationDegree;

    SocialTournament(Solver<LS> _m, range _Weeks, range _Groups, range _Slots,
                     range _Golfers, var{int}[,,] _golfer) : UserConstraint<LS>(_m) {
        m          = _m;
        Weeks      = _Weeks;
        Groups     = _Groups;
        Slots      = _Slots;
        Golfers    = _Golfers;
        golfer     = _golfer;
    }

    void          post();
    var{int}      violations(var{int} x);
    var{int}      violations() { return violationDegree; }
    int           getSwapDelta(var{int} x, var{int} y);
}

```

Statement 21.7: User-Defined Constraint SocialTournament for Social Golfers: I. Statement and Constructor

The most useful class members are the two-dimensional matrix `meetings`, which incrementally maintains the number of times any pair of golfers appears together in a group, and the dictionary `position`, which maps every incremental variable `golfers[w,g,s]` to the triple `(w,g,s)`.

The `post` method states and initializes the class members that determine the semantics of the constraint, using these two structures, which are defined and maintained by an instance `meetInvariant` of the user-defined invariant `Meet`, which is described in a later section. The invariant is posted in the first two lines of the method:

```
Meet meetInvariant(m,Weeks,Groups,Slots,Golfers,golfer);
m.post(meetInvariant);
```

the invariant matrix `meetings` and the dictionary `position` are then retrieved through the corresponding methods of class `Meet`:

```
meetings = meetInvariant.getMeetings();
position = meetInvariant.getPosition();
```

The violations for individual variables and the total number of violations are maintained through invariants. The variable violations of each `golfer[w,g,s]` are maintained with the invariant `varViolations`:

```
varViolations = new var{int}[w in Weeks,g in Groups,s in Slots](m) <-
    sum(t in Slots: t != s) (meetings[golfer[w,g,s],golfer[w,g,t]] >= 2);
```

For a given `golfer[w,g,s]`, the invariant considers golfers in other slots of the same group and week. For each such golfer, `golfer[w,g,t]`, it checks the condition

$$\text{meetings}[\text{golfer}[w,g,s],\text{golfer}[w,g,t]] \geq 2$$

to determine whether the two golfers are grouped together more than once in the schedule. Note that the condition is reified to 1, if it is satisfied and to 0 otherwise. Then the invariant takes the sum over all such conditions, to determine the violations for `golfer[w,g,s]`.

```

void SocialTournament::post() {
    Meet meetInvariant(m,Weeks,Groups,Slots,Golfers,golfer);
    m.post(meetInvariant);
    meetings = meetInvariant.getMeetings();
    position = meetInvariant.getPosition();

    varViolations    = new var{int}[w in Weeks,g in Groups,s in Slots](m) <-
        sum(t in Slots : t != s) (meetings[golfer[w,g,s],golfer[w,g,t]] >= 2);
    violationDegree = new var{int}(m) <-
        sum(g in Golfers, h in Golfers: g < h) max(0,meetings[g,h]-1);
}

var{int} SocialTournament::violations(var{int} x) {
    Position p = position{x};
    return varViolations[p.week,p.group,p.slot];
}

int SocialTournament::getSwapDelta(var{int} x,var{int} y) {
    Position xp = position{x};
    Position yp = position{y};
    assert(xp.week == yp.week);
    assert(xp.group != yp.group);

    int delta = 0;
    forall (s in Slots : s != yp.slot)
        delta += (meetings[x,golfer[yp.week,yp.group,s]] >= 1)
            - (meetings[y,golfer[yp.week,yp.group,s]] >= 2);
    forall (s in Slots : s != xp.slot)
        delta += (meetings[y,golfer[xp.week,xp.group,s]] >= 1)
            - (meetings[x,golfer[xp.week,xp.group,s]] >= 2);
    return delta;
}

```

Statement 21.8: User-Defined Constraint SocialTournament for Social Golfers: II. Implementation

The total number of violations is incrementally maintained through the invariant `violationDegree`:

```
violationDegree = new var{int}(m) <-
  sum(g in Golfers, h in Golfers: g < h) max(0,meetings[g,h]-1);
```

Each pair of golfers, (g,h) , that is grouped together $x > 1$ times, adds $x - 1$ violations to the total number, otherwise it causes no violation. Thus, the contribution of pair (g,h) to the violations is given by the formula

$$\max(0, \text{meetings}[g,h]-1);$$

The total number of violations is computed by summing the contribution of all pairs of golfers.

The implementation of `violations(x)` is very simple. Given an incremental variable `x`, it uses the dictionary `position` to retrieve the triple `p`, whose components, `p.week`, `p.group` and `p.slot`, give the week, group and slot corresponding to `x`. Then the function returns the variable violations corresponding to `p`'s components:

```
Position p = position{x};
return varViolations[p.week,p.group,p.slot];
```

Statement 21.8 concludes with the implementation of the method `getSwapDelta`, which computes the effect of swapping two golfers `x` and `y` in two different groups of the same week. The method first retrieves the week, group and slot of each input variable, in the triples `xp` and `yp`:

```
Position xp = position{x};
Position yp = position{y};
```


Then it adds two assertions, that check the components of these triple, to make sure that the golfers to be swapped are in the same week and in different groups:

```
assert(xp.week == yp.week);
assert(xp.group != yp.group);
```

The following lines compute the difference to the total violations by replacing golfer *y* with golfer *x*, in slot *yp.slot* of group *yp.group*.

```
forall (s in Slots : s != yp.slot)
  delta += (meetings[x,golfer[yp.week,yp.group,s]] >= 1)
           - (meetings[y,golfer[yp.week,yp.group,s]] >= 2);
```

The **forall** loop considers the golfers that are assigned to different slots than *y* in *y*'s group. For every such slot *s*, let *z* is the golfer placed there, i.e.,

```
z = golfer[yp.week,yp.group,s]
```

If *meetings*[*x*,*z*] >= 1, this means that golfers *x* and *z* already play together at least once, so placing golfer *x* in the same group will increase the total number of violations by 1. Similarly, if *meetings*[*y*,*z*] >= 2, the pair of golfers *y* and *z* already contributes at least one violation to the total. Therefore, replacing *y* with *x* will reduce the number of times *y* and *z* play together, and consequently the total number of violations, by 1. Since the effect in both cases is plus/minus 1, both conditions are reified to compute the effect on the increase *delta*. Computing the delta induced by replacing *x* with *y* in *xp.group* is symmetric:

```
forall (s in Slots : s != xp.slot)
  delta += (meetings[y,golfer[xp.week,xp.group,s]] >= 1)
           - (meetings[x,golfer[xp.week,xp.group,s]] >= 2);
```

21.2.4 User-Defined Invariants

The mechanism to add user-defined constraint to COMET is provided through the Interface `Invariant<LS>`, which is summarized in the following block:

```
interface Invariant<LS> {
    Solver<LS> getLocalSolver()
    void      post(InvariantPlanner<LS>)
    void      initPropagation()
    void      propagateInt(boolean, var{int})
    void      propagateFloat(boolean, var{float})
    void      propagateInsertIntSet(boolean, var{set{int}}, int)
    void      propagateRemoveIntSet(boolean, var{set{int}}, int)
}
```

To create a user-defined invariant, it suffices to write a class that implements the `Invariant<LS>` interface. Once a user invariant class is defined, instances can be created and posted to the local solver like any other invariant, using a `post` method. An example of user-defined invariant is the `Meet` invariant defined for the social golfers problem. Before getting into that, here is a brief review of the main methods of the `Invariant<LS>` interface.

The `getLocalSolver` method is a simple accessor that returns the `Solver<LS>` that this invariant is defined on. Typically, it is a one-liner in the class constructor.

The `post(ip)` method is used for posting the invariant inside a constraint system. Its only parameter is an `InvariantPlanner<LS>` variable `ip`; this is a planner that should be used during posting to state any dependencies between the input variables, the invariant, and the output variables. Class `InvariantPlanner<LS>` provides a method `addSource(v)`, that can be used to report that the invariant depends on input incremental variable `v`. Similarly, `addTarget(v)` is a method used to report that the incremental variable `v`'s value depends on the current invariant.

The `initPropagation()` method is automatically called by COMET during planification and allows the invariant to properly initialize its internal state. Note that `initPropagation` is called by COMET, when all source variables, defined by calls to `addSource` during posting, are correctly initialized.

Finally, `propagateInt(b,x)` is the central method triggered by COMET during incremental updates. It is automatically called by COMET, every time there is a change to the value of an integer source variable. The boolean variable `b` indicates whether this is the last invocation of the method. Incremental variable `x` indicates which variable triggered the propagation of the invariant. Method `propagateFloat(b,x)` is the equivalent for floating point variables. The remaining methods are not described here, but have to be included in any class implementing the interface.

The User-Defined Invariant Meet The user-defined invariant **Meet** is used internally in the implementation of the user-defined constraint **SocialTournament** presented earlier. Its role is to incrementally maintain the two-dimensional array **meetings**, that gives the number of times each pair of golfers is grouped together in a tournament. The statement and constructor of invariant **Meet** is contained in Statement 21.9, along with the simplest class methods, which are defined inline. The implementation of the remaining class methods is shown on Statement 21.10.

All class members are the same with the corresponding class members of user-defined constraint **SocialTournament** defined in Statement 21.7, and follow the same semantics. The **Meet** invariant is responsible for correctly initializing and maintaining the array **meetings** and the dictionary **position** used in determining the constraint's violations. These two structures and the invariant's local solver are accessible through the following one-liner functions:

```
Solver<LS>          getLocalSolver() { return m; }
var{int}[, ]        getMeetings()    { return meetings; }
dict{var{int}->Position} getPosition() { return position; }
```

The class constructor basically initializes the instance variables from the constructor's arguments. The only non-trivial part of the constructor lies in the last four lines, which initialize the **meetings** array to 0, initialize the dictionary **position**, and map each incremental variable **golfer[w,g,s]** to its corresponding week-group-slot triple:

```
meetings = new var{int} [Golfers,Golfers](m) := 0;
position = new dict{var{int}->Position}();
forall (w in Weeks, g in Groups, s in Slots)
    position{golfer[w,g,s]} = new Position(w,g,s);
```

The class statement also includes void implementations of the remaining methods of the **Invariant<LS>** interface, in order to avoid a compiler error:

```
void propagateFloat(bool b,var{float} v) {}
void propagateInsertIntSet(bool b,var{set{int}} s,int value) {}
void propagateRemoveIntSet(bool b,var{set{int}} s,int value) {}
```

```

class Meet implements Invariant<LS> {
    Solver<LS>          m;
    range              Weeks;
    range              Groups;
    range              Slots;
    range              Golfers;
    var{int}{,,}       golfer;
    var{int}{,,}       meetings;
    dict{var{int}->Position} position;

    Solver<LS>          getLocalSolver() { return m; }
    var{int}{,,}       getMeetings()    { return meetings; }
    dict{var{int}->Position} getPosition() { return position; }

    Meet(Solver<LS> _m, range _Weeks,range _Groups,range _Slots,
         range _Golfers, var{int}{,,} _golfer) {
        m          = _m;
        Weeks      = _Weeks;
        Groups      = _Groups;
        Slots       = _Slots;
        Golfers     = _Golfers;
        golfer      = _golfer;
        meetings    = new var{int}[Golfers,Golfers](m) := 0;
        position    = new dict{var{int}->Position}();
        forall (w in Weeks, g in Groups, s in Slots)
            position{golfer[w,g,s]} = new Position(w,g,s);
    }

    void post(InvariantPlanner<LS> planner);
    void initPropagation();
    void propagateInt(bool b,var{int} v);

    void propagateFloat(bool b,var{float} v) {}
    void propagateInsertIntSet(bool b,var{set{int}} s,int value) {}
    void propagateRemoveIntSet(bool b,var{set{int}} s,int value) {}
}

```

Statement 21.9: User-Defined Invariant Meet: I. Statement and Constructor

It remains to review the main class methods, implemented in Statement 21.10, to conclude this CBLs approach to the social golfers problem. The `post` method posts the invariant into a constraint system. It uses the planner's `addSource()` method to report that the invariant depends on the variables `golfer[w,g,s]`, for all possible values of week, group and slot. In other words, to report that variables `golfer[w,g,s]` are the source variables:

```
forall (w in Weeks, g in Groups, s in Slots)
    planner.addSource(golfer[w,g,s]);
```

Similarly, using the planner's `addTarget()` method, variables `meetings[p,q]` are added as target variables, which means that their value is automatically updated, every time a source variable `golfer[w,g,s]` takes a new value:

```
forall (p in Golfers, q in Golfers)
    planner.addTarget(meetings[p,q]);
```

The `initPropagation()` method computes the initial values of target variables `meetings`. It consists of two nested selectors. The outer selector goes through all weeks and groups. For each week `w` and group `g`, the code runs through all different pairs of slots within the group, and increases by 1 the number of times that the golfers placed there are grouped together. Keep in mind that the `meetings` array is defined for ordered pairs of golfers, so both assignments are necessary.

```
forall (s in Slots, t in Slots : s < t) {
    meetings[golfer[w,g,s],golfer[w,g,t]]++;
    meetings[golfer[w,g,t],golfer[w,g,s]]++;
}
```

```

void Meet::post(InvariantPlanner<LS> planner) {
    forall (w in Weeks, g in Groups, s in Slots)
        planner.addSource(golfer[w,g,s]);
    forall (p in Golfers, q in Golfers)
        planner.addTarget(meetings[p,q]);
}

void Meet::initPropagation() {
    forall (w in Weeks, g in Groups)
        forall (s in Slots, t in Slots : s < t) {
            meetings[golfer[w,g,s],golfer[w,g,t]]++;
            meetings[golfer[w,g,t],golfer[w,g,s]]++;
        }
}

void Meet::propagateInt(bool b, var{int} v) {
    Position p = position{v};
    int oldGolfer = v.getOld();
    int newGolfer = v;
    forall (s in Slots : s != p.slot) {
        int o = golfer[p.week,p.group,s];
        meetings[oldGolfer,o]--;
        meetings[o,oldGolfer]--;
        meetings[newGolfer,o]++;
        meetings[o,newGolfer]++;
    }
}

```

Statement 21.10: User-Defined Invariant Meet: II. Implementation

Finally, incremental updates of the target variables `meetings` are taken care of by the `propagateInt` method, which is called every time a source variable `v` takes a new value. The method first retrieves the (week,group,slot) triple `p` corresponding to the variable `v`, through the `position` dictionary, and stores the old and new value of the variable to integers `oldGolfer` and `newGolfer`. Note the use of the `getOld()` method to retrieve the previous value of an incremental variable.

```
Position p      = position{v};  
int oldGolfer   = v.getOld();  
int newGolfer   = v;
```

Then, the method goes through all the other slots in the week and group corresponding to variable `v`. These are given as components of the triple `p`: `p.week` and `p.group`. For each such slot, `s`, it first determines the golfer `o` assigned to `s`:

```
int o = golfer[p.week,p.group,s];
```

Since `oldGolfer` is not placed in that group anymore, we decrease the number of times `o` is grouped together with `oldGolfer` by one:

```
meetings[oldGolfer,o]--;  
meetings[o,oldGolfer]--;
```

Similarly, since `newGolfer` is now present in the same group with golfer `o`, we need to increase the number of times `o` is grouped together with `newGolfer` by one:

```
meetings[newGolfer,o]++;  
meetings[o,newGolfer]++;
```

IV MATHEMATICAL PROGRAMMING

Chapter 22

Linear Programming

This chapter gives a review of the Mathematical Programming module of COMET. This module basically allows to write Linear and Integer Programming models; it also offer facilities for more advanced linear programming techniques, such as adding cuts, column generation and hybridization with MIP and CP. The chapter starts with a simple Linear Programming (LP) model, and then shows how to apply more advanced techniques in the context of non-trivial examples.

22.1 Production Planning

Statement 22.1 depicts the COMET model for a simple production planning problem. A number of 12 products can be produced, each of which has a set of features, such as volume, weight, etc. There is a capacity constraint on the total amount that can be produced from each feature; for example, an upper limit of the total weight of the produced products. Also, each product generates a profit per unit produced. The objective is to maximize the total profit, while satisfying capacity constraints.

The library to import, in order to use a linear programming solver is `cotln`. After defining the problem's data, the model creates a solver and a float linear programming variable `var<LP>{float}` each product: `x[p]` represents the quantity produced of product `p`.

```
import cotln;
Solver<LP> lp();
var<LP>{float} x[Products] (lp);
```

The objective function is stated in exactly the same way as in Constraint Programming, with the restriction that it must be a linear expression:

```
maximize<lp>
  sum(p in Products) c[p] * x[p]
```

Finally the capacity constraints are posted in the **subject to** block. The constraints of a linear programming model must also be linear.

```
forall (d in Dimensions)
  lp.post(sum(p in Products) coef[d,p] * x[p] <= b[d]);
```

```

int    nbDimensions = 7;
int    nbProducts   = 12;
range Dimensions    = 1..nbDimensions;
range Products      = 1..nbProducts;

int    b[Dimensions] = [18209, 7692, 1333, 924, 26638, 61188, 13360];
float  c[Products]   = [96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81];

float coef[Dimensions,Products] = [
  [ 19,  1, 10,  1,  1, 14, 152, 11,  1,  1, 1, 1],
  [  0,  4, 53,  0,  0, 80,  0,  4,  5,  0, 0, 0],
  [  4, 660,  3,  0, 30,  0,  3,  0,  4, 90, 0, 0],
  [  7,  0, 18,  6, 770, 330,  7,  0,  0,  6, 0, 0],
  [  0, 20,  0,  4, 52,  3,  0,  0,  0,  5, 4, 0],
  [  0,  0, 40, 70,  4, 63,  0,  0, 60,  0, 4, 0],
  [  0, 32,  0,  0,  0,  5,  0,  3,  0, 660, 0, 9]];

import cotln;
Solver<LP> lp();
var<LP>{float} x[Products](lp);
maximize<lp>
  sum(p in Products) c[p] * x[p]
subject to
  forall (d in Dimensions)
    lp.post(sum(p in Products) coef[d,p] * x[p] <= b[d]);

```

Statement 22.1: Linear Programming Model for Production Planning

22.2 Warehouse Location

We have already seen a CBLIS approach to the Warehouse Location Problem. In this problem, a number of warehouse locations are candidates to serve stores. Each store can be served by a single warehouse, and warehouses can serve unlimited number of stores. The problem is to decide which warehouses to open and which warehouse is assigned to each store. There is a fixed cost for opening a warehouse and also a transportation cost for each possible (store,warehouse) link, and the goal is to minimize the total cost. This section revisits the problem, to present a Mixed Integer Programming approach. The implementation is basically adding a search component to an LP model.

Model The problem data consists of the ranges of warehouses and stores. The value `fcost[w]` is the fixed cost of opening a warehouse `w` and `tcost[w,s]` is the transportation cost from warehouse `w` to store `s`.

```
range Warehouses;  
range Stores;  
float fcost[Warehouses];  
float tcost[Warehouses,Stores];
```

Note that, once the decision of which warehouses to open is taken, the cheapest assignment of stores to warehouses is easily achieved by assigning each store to its closest warehouse. Hence the real decision variables of this problem is the opening status of each store. The model variables are:

```
var<LP>{float} open[Warehouses] (lp,0..1);  
var<LP>{float} x[Warehouses,Stores] (lp,0..1);
```

where `open[w]` is equal to 1, if warehouse `w` is open and `x[w,s]` is equal to one if store `s` is assigned to warehouse `w`. Note that the decision variables are declared as `float` variables, even though they need to be integer (0-1) in the final solution. The variables' integrality is enforced by the search.

```

//data to be read from input file
range Warehouses;
range Stores;
float fcost[Warehouses];
float tcost[Warehouses,Stores];

import cotln;
Solver<LP> lp();
var<LP>{float} open[Warehouses](lp,0..1);
var<LP>{float} x[Warehouses,Stores](lp,0..1);

minimize<lp>
    sum(w in Warehouses) fcost[w] * open[w] +
    sum(w in Warehouses, s in Stores) tcost[w,s] * x[w,s]
subject to {
    forall (w in Warehouses, s in Stores)
        lp.post(x[w,s] <= open[w]);
    forall (s in Stores)
        lp.post(sum(w in Warehouses) x[w,s] == 1);
}
using {
    bool notInteger;
    do {
        notInteger = false;
        selectMin (w in Warehouses : open[w].getValue() > 0.00001
                    && open[w].getValue() < 0.9999)
            (abs(1.0-open[w].getValue())) {
            notInteger = true;
            try<lp> lp.updateLowerBound(open[w],1);
                | lp.updateUpperBound(open[w],0);
            }
        } while (notInteger);
    }
}

```

Statement 22.2: MIP Model for the Warehouse Location Problem

The expression to minimize is the sum of fixed and transportation costs:

```
minimize<lp>
  sum(w in Warehouses) fcost[w] * open[w] +
  sum(w in Warehouses, s in Stores) tcost[w,s] * x[w,s]
```

The constraints state that, if a store is assigned to a warehouse, this warehouse must be open:

```
forall (w in Warehouses, s in Stores)
  lp.post(x[w,s] <= open[w]);
```

and that every store must be assigned to exactly one warehouse:

```
forall (s in Stores)
  lp.post(sum(w in Warehouses) x[w,s] == 1);
```

Search The non-deterministic search, in the **using** block, enforces the integrality of the **open** variables by creating two alternative branches, when the status of a warehouse is not an integer. In the left alternative the warehouse is forced to be open and in the right alternative it is closed. The search continues as long as there exist fractional **open[w]** variables. In order to avoid numerical precision problems, a value is considered to be 0, if smaller than 0.00001, and 1, if above 0.9999.

```
bool notInteger;
do {
  notInteger = false;
  selectMin (w in Warehouses : open[w].getValue() > 0.00001
            && open[w].getValue() < 0.9999)
    (abs(1.0-open[w].getValue())) {
    notInteger = true;
    try<lp> lp.updateLowerBound(open[w],1);
           | lp.updateUpperBound(open[w],0);
    }
} while (notInteger);
```

22.3 Column Generation

In this section, we use a cutting stock problem as a vehicle to demonstrate how to implement in COMET a column generation strategy for linear problems. The approach of this section leads to a master problem, in which the pricing sub-problems are solved using either MIP or CP, illustrating COMET's high capacity for creating hybridization schemes.

22.3.1 Cutting Stock Problem

In the cutting stock problem considered here, there is a supply of wooden planks of fixed size W , that are intended to be cut into smaller shelves. Shelves of different lengths need to be procured and there is a given demand for shelves of each length. In particular, the number of shelves of size $w_i \leq W$ that need to be produced is equal to b_i . The problem is then to cut the planks in a way that minimizes the total waste of wood. A possible linear model for this problem is the following:

$$\begin{aligned}
 \min \quad & \sum_j Y_j \\
 \text{subject to} \quad & \sum_i X_{ij} w_i \leq W \cdot Y_j, & \forall j \\
 & \sum_j X_{ij} \geq b_i, & \forall i \\
 & X_{ij} \in \mathbb{N} \\
 & Y_i \in \{0, 1\}
 \end{aligned}$$

where X_{ij} represents the number of shelves of size w_i assigned to plank j and Y_j is equal to 1, if the plank j is cut into at least one shelf. Unfortunately, this is not a very good formulation for the problem: it presents many symmetries, since planks can be exchanged within a solution, and also the linear relaxation is very weak. A stronger formulation is the following:

$$\begin{aligned}
 \min \quad & \sum_j X_j \\
 \text{subject to} \quad & \sum_j a_{ij} X_j \geq b_i, & \forall i \\
 & X_j \in \mathbb{N}
 \end{aligned}$$

where $[a_{1j}, \dots, a_{nj}]$ constitutes a configuration j that is a possible assignment of shelves to a plank: for each shelf i , the number of shelves of size w_i in this configuration j is a_{ij} . A configuration must satisfy the capacity constraint of the plank: $\sum_i a_{ij} w_i \leq W$. The variable X_j is the number of configurations j that are chosen in the final solution.

We are going to refer to this last formulation as the *master* problem. Formulating the problem in this way offers two main advantages:

- it avoids symmetries
- it has a stronger linear relaxation

The drawback is that all the configurations must be given in the beginning and this number can be huge. This is circumvented by generating new columns lazily, guided by the simplex algorithm used to solve the master problem. We can use results from linear programming theory, to determine whether there is a new configuration that can improve the objective function, or, on the contrary, the current solution is optimal, so it is not necessary to consider any other configuration. More precisely, a new configuration $[\alpha_1, \dots, \alpha_n]$ can improve the objective function if

$$1 - \sum_i \alpha_i y_i < 0$$

where y_i is the dual variable associated with the constraint $\sum_j X_{ij} \geq b_i$.

In mathematical programming theory, $1 - \sum_i \alpha_i y_i$ is called the reduced cost of the column (configuration) and the simplex algorithm can improve its objective, only if it can find a variable with a negative reduced cost. The problem of lazily finding a new configuration with a negative reduced cost is called the *pricing* problem, and it is also an integer linear optimization. In fact, it is a knapsack problem:

$$\begin{aligned} \min \quad & 1 - \sum_i \alpha_i y_i \\ \text{subject to} \quad & \sum_i \alpha_i w_i \leq W \\ & \alpha_i \in \mathbb{N} \end{aligned}$$

If the objective function is negative, then the new configuration $[\alpha_1, \dots, \alpha_n]$ must be added to the master problem along with its associated new variable X' . The modified master problem is re-optimized, generating a better objective function and new dual variables. The process is repeated until there is no more configuration with negative reduced cost. The objective function of the linear programming master problem is then proven optimal.

Statement 22.3 shows the data and main data structures used for this problem. The width of the large planks is 110. There are five different shelf lengths, given in the array `shelf`. The demand for each length is specified in the array `demand`.

```
int    W                = 110;
int    nbShelves        = 5;
range  Shelves           = 1..nbShelves;
int    shelf[Shelves]    = [20, 45, 50, 55, 75];
int    demand[Shelves]  = [48, 35, 24, 10, 8];
```

A configuration together with its associated variable is described by the class `Configuration`. The instance variable `_X` represents the number of times this configuration is selected in the final solution. The number of occurrences of each shelf length in the configuration is given by the instance variable array `_shelf`.

```
class Configuration {
  var<LP>{float} _X;
  int[] _shelf;
  Configuration(var<LP>{float} X,int[] shelf) { _shelf = shelf; _X = X; }
  var<LP>{float} X() { return _X; }
  int getShelf(int s) { return _shelf[s]; }
}
```

A stack `C` is created, in order to store the configurations that are generated in the solution process and are available to the master problem. In order to start with a feasible problem, the stack is initialized with some elementary configurations: one configuration for each shelf length, containing as many shelves of this length as possible.

```
stack{Configuration} C();
forall (i in Shelves) {
  int config[Shelves] = 0;
  config[i] = W/shelf[i];
  C.push(Configuration(new var<LP>{float}(lp),config));
}
range Configs = 0..C.getSize()-1;
```

```

import cotln;
int    W                = 110;
int    nbShelves        = 5;
range  Shelves          = 1..nbShelves;
int    shelf[Shelves]   = [20, 45, 50, 55, 75];
int    demand[Shelves] = [48, 35, 24, 10, 8];

class Configuration {
    var<LP>{float} _X;
    int[] _shelf;
    Configuration(var<LP>{float} X,int[] shelf) { _shelf = shelf; _X = X; }
    var<LP>{float} X() { return _X; }
    int getShelf(int s) { return _shelf[s]; }
}

stack{Configuration} C();
forall (i in Shelves) {
    int config[Shelves] = 0;
    config[i] = W/shelf[i];
    C.push(Configuration(new var<LP>{float}(lp),config));
}
range Configs = 0..C.getSize()-1;

```

Statement 22.3: Column Generation Approach to the Cutting Stock Problem (Part 1/2)

Statement 22.4 contains the main body of the solution. Inside the while loop, it alternates between the generation of a new configuration by solving the pricing problem and the introduction of the corresponding new column into the master problem. The process continues until no more configuration with negative reduced cost can be discovered.

The solution first solves the initial master problem. The linear constraint object corresponding to each shelf is stored into the array `constr`, that will allow us to retrieve the dual values, when solving the pricing problem.

```
Constraint<LP> constr[Shelves];
minimize<lp>
    sum(j in Configs) C{j}.X()
subject to
    forall (i in Shelves)
        constr[i] = lp.post(sum(j in Configs) C{j}.getShelf(i)*C{j}.X() >= demand[i]);
```

The first lines of the `while` loop solve the pricing problem, which is basically a knapsack problem. The value of the dual variables (float constants) are first retrieved from the constraint objects. The configuration is an array of variables `use`, that give, for each shelf, the number of times it is selected in the configuration. The only constraint is that the total length of the configuration cannot exceed the plank width `W`.

```
float cost[s in Shelves] = constr[s].getDual();
Solver<MIP> ip();
var<MIP>{int} use[Shelves] (ip,0..W);
minimize<ip>
    1 - sum(i in Shelves) cost[i]*use[i]
subject to
    ip.post(sum(i in Shelves) shelf[i] * use[i] <= W);
```

If it is not possible to find a new configuration with negative reduced cost, the process terminates, since the linear master program is optimal.

```
if (ip.getObjective().getValue().getFloat() >= 0.00000)
    break;
```

```

Constraint<LP> constr[Shelves];
minimize<lp>
    sum(j in Configs) C{j}.X()
subject to
    forall (i in Shelves)
        constr[i] = lp.post(sum(j in Configs) C{j}.getShelf(i)*C{j}.X() >= demand[i]);
cout << "obj: " << lp.getObjective().getValue().getFloat() << endl;
do {
    float cost[s in Shelves] = constr[s].getDual();
    Solver<MIP> ip();
    var<MIP>{int} use[Shelves](ip,0..W);
    minimize<ip>
        1 - sum(i in Shelves) cost[i]*use[i]
    subject to
        ip.post(sum(i in Shelves) shelf[i] * use[i] <= W);

    if (ip.getObjective().getValue().getFloat() >= 0.00000)
        break;
    else {
        Column<LP> col(lp);
        col.setObjectiveCoefficient(1.0);
        forall (i in Shelves)
            col.setGeqConstraintCoefficient(constr[i].getId(),use[i].getValue());
        var<LP>{float} n(lp,col);
        C.push(Configuration(n,all(i in Shelves) use[i].getValue()));
        cout << "obj: " << lp.getObjective().getValue().getFloat() << endl;
    }
} while (true);

```

Statement 22.4: Column Generation Approach to the Cutting Stock Problem (Part 2/2)

Otherwise, the new configuration can enter the master linear program. The **else** block modifies the linear program by introducing a new variable, as follows. First, it specifies the new coefficient in the objective function and a new coefficient in each linear constraint, using a **Column** object:

```
Column<LP> col(lp);
col.setObjectiveCoefficient(1.0);
forall (i in Shelves)
    col.setGeqConstraintCoefficient(constr[i].getId(),use[i].getValue());
```

Then it creates a new variable from this **Column** object, corresponding to the new configuration found. The effect is to automatically re-optimize the problem. The new **Configuration** object is then pushed into the stack of available configurations.

```
var<LP>{float} n(lp,col);
C.push(Configuration(n,all(i in Shelves) use[i].getValue()));
```

22.3.2 Hybridization

A column generation algorithm uses a linear master problem. For some problems, the pricing slave problem, that is in charge of discovering variables with negative reduced cost, is not necessarily linear. Using Constraint Programming to solve the pricing problem is a standard example of hybridization between LP and CP.

Statement [22.5](#) shows how to modify the column generation algorithm for the cutting stock problem, in order to solve the knapsack pricing problem using CP rather than IP. Although these two techniques are radically different behind the scenes, from the user's point of view the switch is straightforward: the syntax is almost the same, the only notable difference being the use of the **labelFF** method to perform the search.

```

do {
    float cost[s in Shelves] = constr[s].getDual();
    Solver<CP> cp();
    var<CP>{int} use[Shelves](cp,0..W);
    minimize<cp>
        1 - sum(i in Shelves) cost[i] * use[i]
    subject to
        cp.post(sum(i in Shelves) shelf[i] * use[i] <= W);
    using
        labelFF(use);

    if (cp.getObjective().getValue().getFloat() >= 0.00000)
        break;
    else {
        Column<LP> col(lp);
        col.setObjectiveCoefficient(1.0);
        forall (i in Shelves)
            col.setGeqConstraintCoefficient(constr[i].getId(),use[i].getValue());
        var<LP>{float} n(lp,col);
        C.push(Configuration(n,all(i in Shelves) use[i].getValue()));
    }
} while (true);

```

Statement 22.5: Cutting Stock Problem Using CP for Solving the Pricing Problem

V VISUALIZATION

Chapter 23

Comet's Graphical Interface

The COMET language provides a visualization module, called `qtvisual`, which is based on the Qt cross platform library. Visualization is typically used in COMET for visualizing the process of solving of a model and to show its solutions. The present chapter introduces only the API of `qtvisual`, independently from any solver. The chapters following it introduce visualization together with events and show how visualization can be separated from a CP or LNS model.

23.1 Visualizer and CometVisualizer

`CometVisualizer` and `Visualizer` are the basic visualization classes. The `CometVisualizer` class is a wrapper around `Visualizer`, that provides some facilities demonstrated in Section 25.2. Widgets can be added to either `CometVisualizer` or `Visualizer`. `CometVisualizer` does not inherit from `Visualizer`; a method `getVisualizer()` is available in `CometVisualizer` for giving access to the `Visualizer` object of the class.

Moreover `CometVisualizer` creates the main window that contains a set of standard buttons and a status bar:

```
import qtvisual;
CometVisualizer visu();
visu.pause();
```

Figure 23.1 shows the resulting main window. The main window is closed by default, when COMET exits. The line `visu.pause()` sets a yellow message in the status bar and forces `CometVisualizer` to wait for a click on the *Run* button, before continuing program execution. Alternatively, the line `visu.finish()` would wait for a click on the *Close* button, rather than the *Run*.

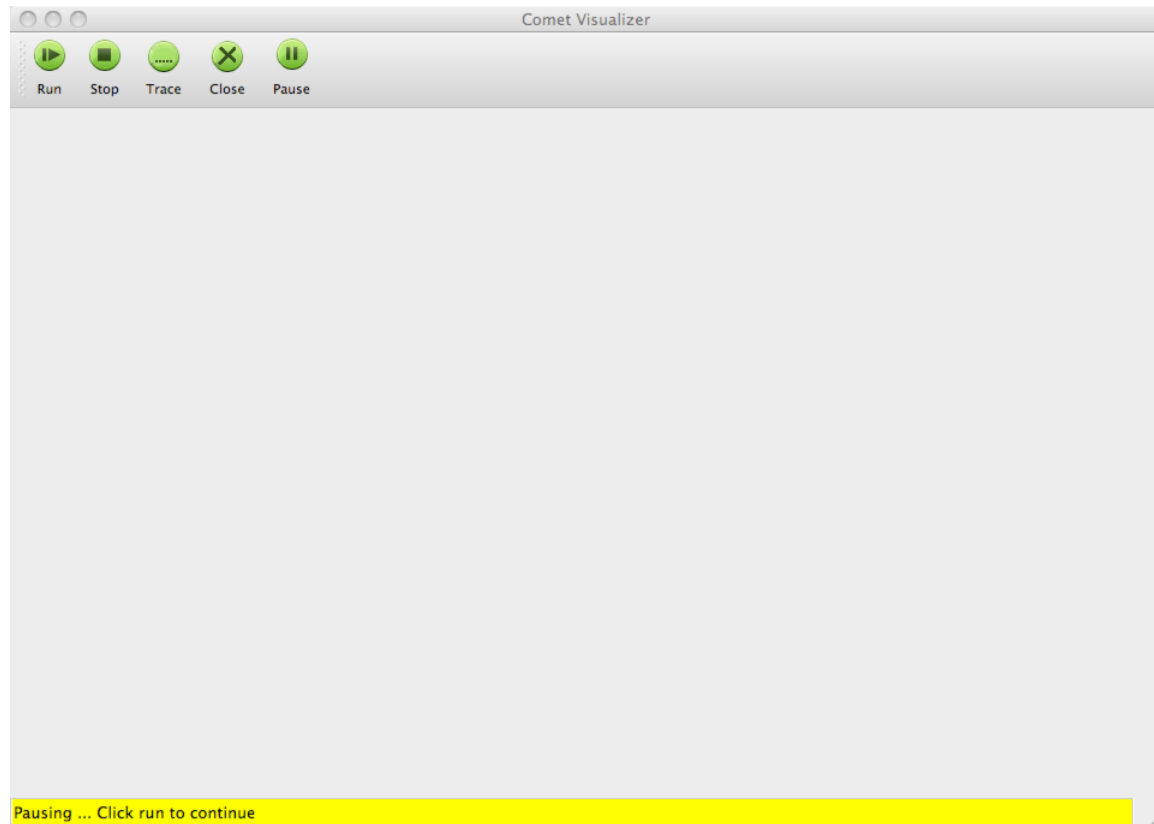


Figure 23.1: The standard main window created by COMET, with its buttons and status bar.

23.2 Adding Widgets through Notebook Pages

Widgets cannot be directly added to the main window. Instead, one has to use containers called *notebook pages*. A notebook page is a dock widget that serves as a container for other widgets. A dock widget can be moved and detached by dragging and dropping. Moreover, within each notebook pages, the contained widgets can be arranged in tabs.

In the following, the basic label widget is used in order to illustrate notebook pages. A label can be declared in the following way:

```
VisualLabel labelTop(visu.getVisualizer(),"Top Area");
```

Note that the label is not added to the main window yet. The extra call `visu.getVisualizer()`, returns the actual COMET visualizer of type `Visualizer`.

The default layout of the main window has four areas: left, right, bottom and top. Each area can contain a single notebook page. Let us now declare a label widget for each area of the main window:

```
VisualLabel labelTop(visu.getVisualizer(),"Top Area");  
VisualLabel labelLeft(visu.getVisualizer(),"Left Area");  
VisualLabel labelRight(visu.getVisualizer(),"Right Area");  
VisualLabel labelBottom(visu.getVisualizer(),"Bottom Area");
```

We add each label to a specific area of the main window through notebook pages:

```
visu.addLeftNotebookPage(labelLeft);  
visu.addRightNotebookPage(labelRight);  
visu.addTopNotebookPage(labelTop);  
visu.addBottomNotebookPage(labelBottom);
```

Note that, for creating notebook pages in the top area, one can equivalently use the instruction `addNotebookPage`. Figure 23.2 shows the resulting interface; note that each label is contained in a notebook page that can be dragged and dropped. The complete code to generate the figure is shown on Statement 23.1.

```
import qtvisual;
CometVisualizer visu();
Visuallabel labelTop(visu.getVisualizer(),"Top Area");
Visuallabel labelLeft(visu.getVisualizer(),"Left Area");
Visuallabel labelRight(visu.getVisualizer(),"Right Area");
Visuallabel labelBottom(visu.getVisualizer(),"Bottom Area");
visu.addLeftNotebookPage(labelLeft);
visu.addRightNotebookPage(labelRight);
visu.addTopNotebookPage(labelTop);
visu.addBottomNotebookPage(labelBottom);
visu.finish();
```

Statement 23.1: Putting a Label in Each Area Through Notebook Pages (`notebookPages.co`). See Figure 23.2.

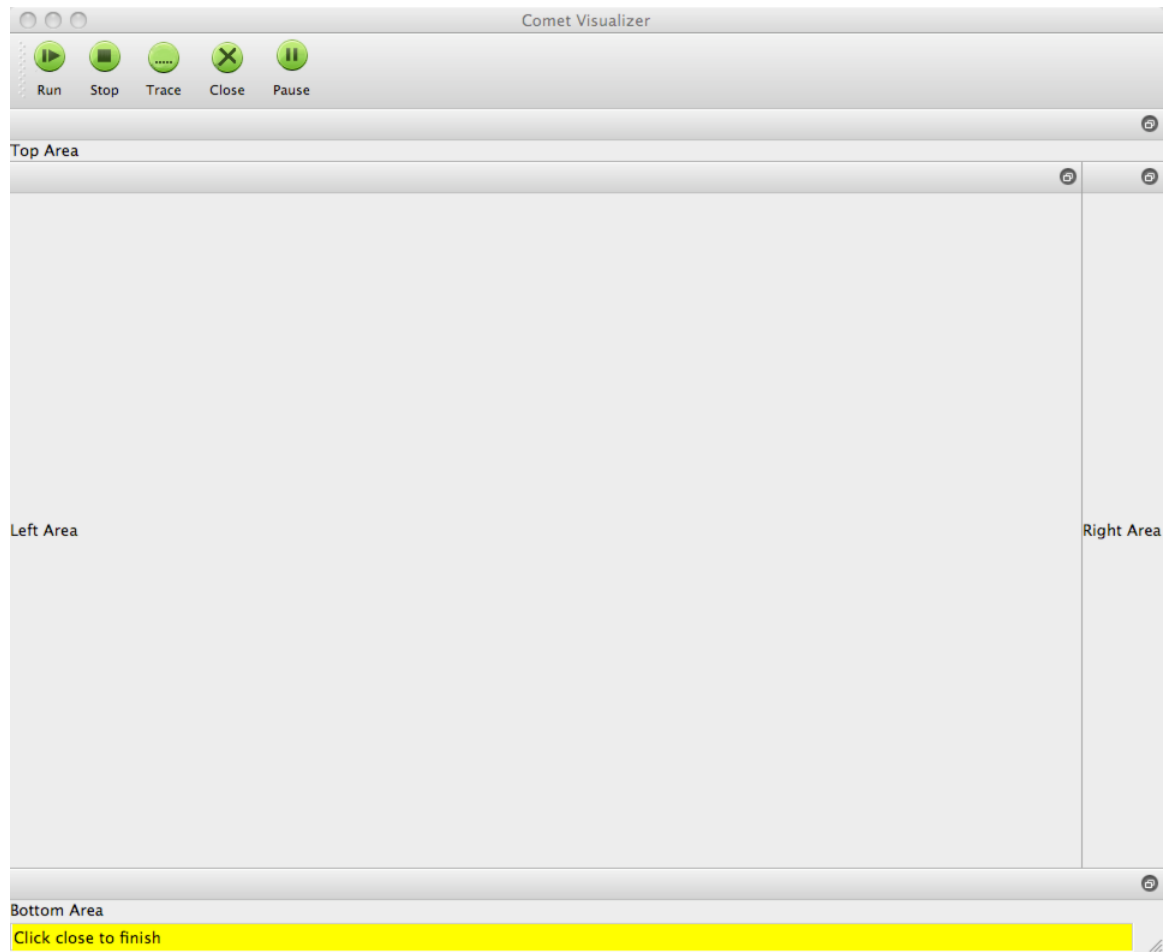


Figure 23.2: Each area is assigned to a label through a notebook page. The user can drag and drop notebook pages.

To extend this example, the user can choose to add an additional label to the left area, with the following two lines:

```
VisualLabel labelAdditional(visu.getVisualizer(),"Additional label");  
visu.addLeftNotebookPage(labelAdditional);
```

When more than one widget are added to a notebook page, they can either be stacked on the notebook page, or the notebook can place each widget into a separate tab, as shown in Figure 23.3.

Stacking an object can be done by specifying **false** as the second argument of the corresponding **addNotebook** command.

```
visu.addLeftNotebookPage(labelAdditional,false);
```

Figure 23.4 shows the resulting layout.

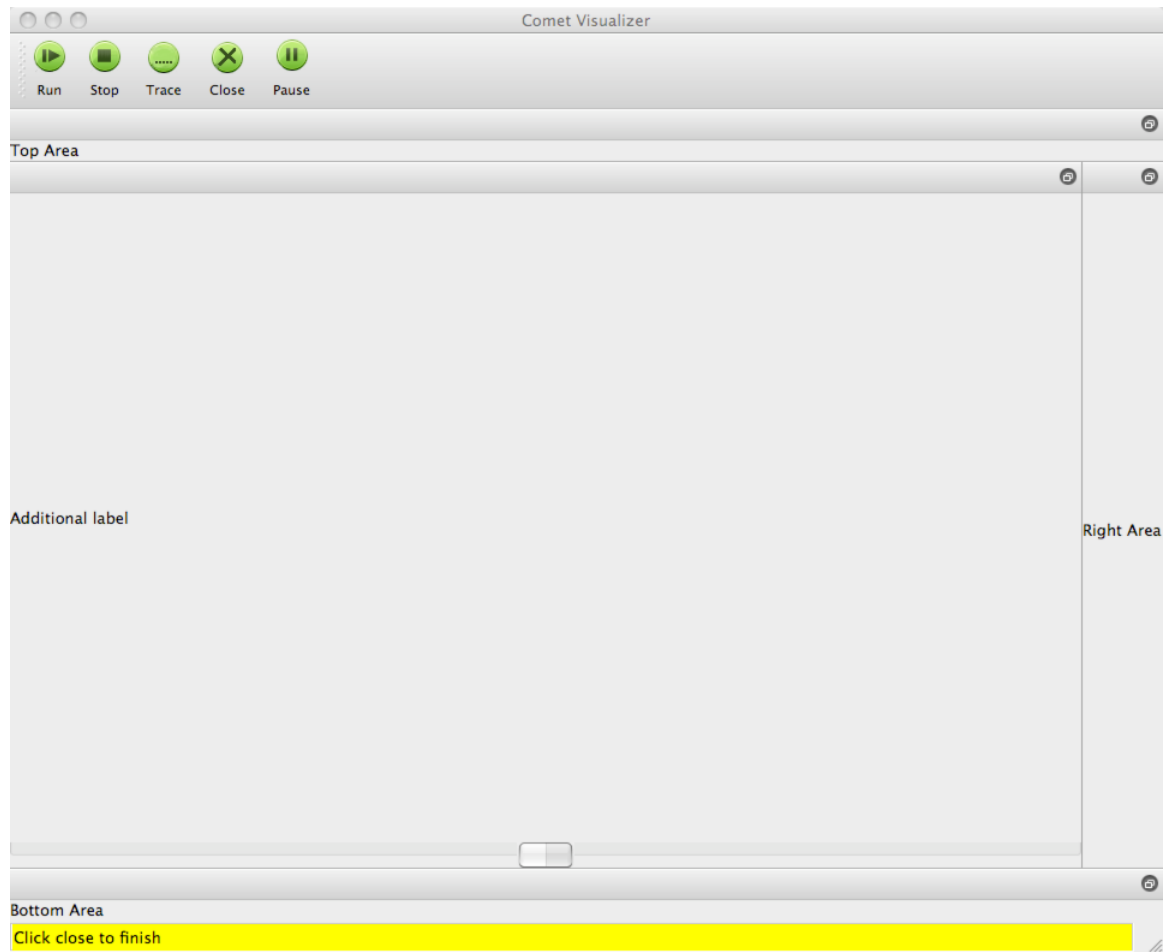


Figure 23.3: The additional label in the left area is added to the left notebook page. Tabs are automatically created since the left notebook page now contains two widgets.

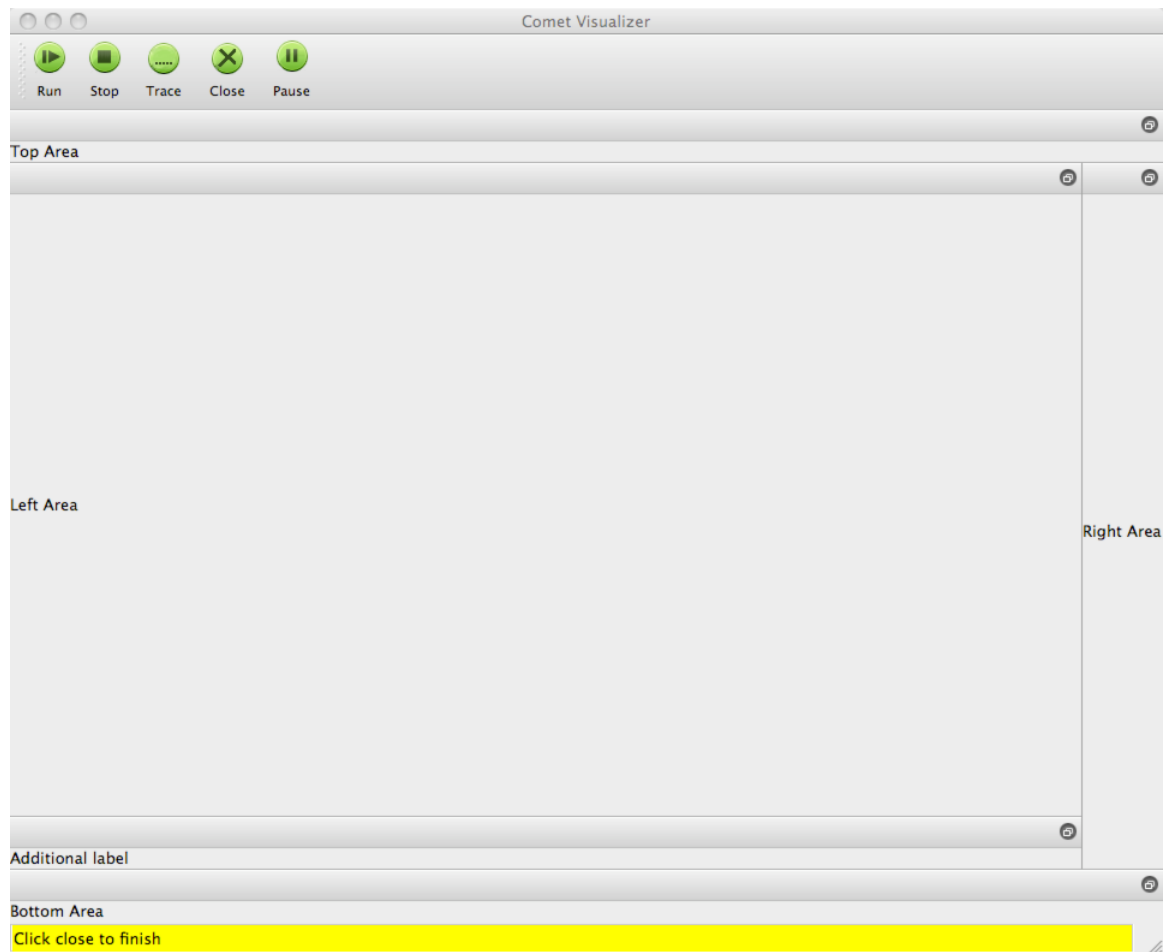


Figure 23.4: Two widgets share the same left notebook page and are simply stacked.

23.3 Dealing with Colors and Fonts

In the `Qt` visual API, colors are represented by strings, whose format follows the `Qt` conventions (class `QColor`) defined at <http://doc.trolltech.com/4.5/qcolor.html#setNamedColor>. A color string can take any name from the list of SVG color keywords provided by the World Wide Web Consortium; for example, “steelblue” or “gainsboro”. These colors are cross-platform. Also, the string “transparent” can be used to represent no color.

In addition to that, a color value can be specified using strings of hexadecimal digits. Depending on the desired level of granularity, there are four options for the color format:

- `#RGB`
- `#RRGGBB`,
- `#RRRGGBBB`,
- `#RRRRGGGBBBB`,

In this list of formats, each letter R, G, and B is a single hexadecimal digit corresponding to the values of “red”, “green” and “blue”, respectively. For example, the format `#RGB` uses 4 bits for each color value. Similarly, `#RRGGBB` allows for 8 bits per color, `#RRRGGBBB` for 12 bits and `#RRRRGGGBBBB` for 16 bits. For instance, the color red can be described using any of the strings `#F00`, `#FF0000` and `#FFF000000`.

In order to specify a font, the acceptable formats are “fontname,fontsize”, “fontname fontsize”, or just “fontname”. Font names can be selected among those accepted by `QFont` (<http://doc.trolltech.com/4.5/qfont.html#QFont-2>). Note that font definition is system-dependent: fonts that work on one operating system may not be available on other systems.

23.4 Text Tables

We now give an example of a more interesting widget in COMET's visualization. The `VisualTextTable` widget displays a two-dimensional table of labels with row and column headers. Creating a `VisualTextTable` is straightforward:

```
VisualTextTable textTable(visu.getVisualizer(), "Example", 1..4, 1..5);  
visu.addTopNotebookPage(textTable);
```

The first string denotes the table's name, which is also used when tabulating the table in a notebook page. The following two ranges are the range of the columns and rows. Labels on entries can be set as follows:

```
textTable.setLabel(2,3,"label");
```

Tooltips are yellow pop-ups that show up when hovering the mouse pointer over an object. They can prove useful for displaying additional information associated with the widget without obfuscating the visualization. Tooltips can, for example, be associated with table entries:

```
textTable.setToolTip(2,3,"toolTip");
```

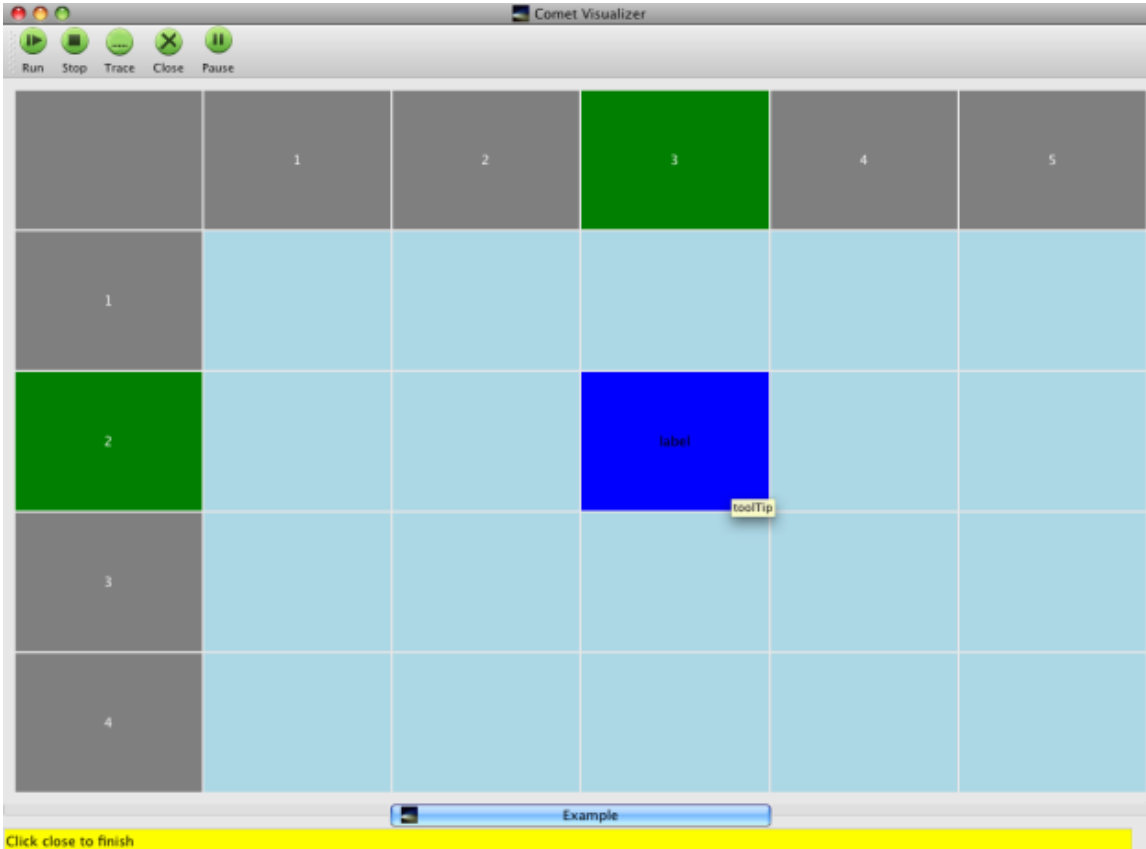
Adding a background color to a table entry is possible with the `setColor` method, while the `highlight` method can be used for coloring the corresponding row and column number:

```
textTable.setColor(2,3,"blue");  
textTable.highlight(2,3,"green");
```

The complete code for creating this visual table is shown on Statement [23.2](#) and the resulting `VisualTextTable` appears on Figure [23.5](#).

```
import qtvisual;  
CometVisualizer visu();  
VisualTextTable textTable(visu.getVisualizer(), "Example", 1..4, 1..5);  
visu.addNotebookPage(textTable);  
textTable.setLabel(2,3,"label");  
textTable.setToolTip(2,3,"toolTip");  
textTable.setColor(2,3,"blue");  
textTable.highlight(2,3,"green");  
visu.finish();
```

Statement 23.2: Using a Text table (`textTable.co`). See Figure [23.5](#).



Comet Visualizer

Run Stop Trace Close Pause

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|-------|---|---|
| 1 | | | | | |
| 2 | | | label | | |
| 3 | | | | | |
| 4 | | | | | |

Example

Click close to finish

Figure 23.5: Example of a `VisualTextTable` where entry (2,3) is colored and highlighted. Notice the tooltip over the table entry.

23.5 Drawing Boards

We conclude this chapter with the `VisualDrawingBoard` widget, which is a canvas for basic drawing. Many different drawing objects can be added on the canvas, including among others: `VisualGrid`, `VisualCircle`, `VisualArc`, `VisualLine`, `VisualPolyline`, and `VisualRectangle`. Visual drawing boards have several interesting features:

- They support an `autoFitContent` feature. This automatically zooms a `DrawingBoard` to fit all items into the scrolled area. The `fitContent()` method fits all items once without setting `autoFitContent`.
- The `VisualDrawingBoard` class also supports manual zooming: holding the `ALT` key you can zoom in and out by scrolling the mouse wheel up and down, respectively. Note that the zooming feature behaves best, when `autoFitContent` is set to `false`.
- Double-clicking centers on the area clicked.
- A buffering method can be used to add and modify added items in batch, to avoid flickering.

The following block of `COMET` code shows how to define a visual drawing board. As in the case of the `VisualTextTable` widget, the first argument of the constructor is the visualizer corresponding to the `CometVisualizer` object. The second argument determines with the canvas's name. The third argument is a boolean set to `true`, if the vertical coordinates are ordered from the bottom to the top, or `false` otherwise.

```
VisualDrawingBoard canvas(visu.getVisualizer(), "TEST1", true);
visu.addNotebookPage(canvas1);
canvas1.setBuffering(true);
// Add items here (e.g. , VisualLine, VisualCircle, etc. )
canvas1.setBuffering(false);
```

Note the use of the `setBuffering()` method, for adding or modifying items in batch: the behavior of the code between the two calls to `setBuffering` is equivalent to using an `atomic` block:

```
with atomic(v) {
    //Add items here
}
```

The following line forces the canvas to fit all the items added to it:

```
canvas1.autoFitContent(true);
```

Finally, the next lines add a number of graphical items to the canvas:

```
VisualCircle c1(canvas1,20,20,10);  
VisualCircle c1b(canvas1,50,50,10);  
c1.setCenter(20,20);  
VisualText txt1(canvas1, 20,20, "hi", "green", "Helvetica");  
VisualLine (canvas1, 20,0, 20,40,1,"#ffff00",VisualLineSolid);  
VisualLine (canvas1, 0,20, 40,20,1,"#ffff00",VisualLineSolid);
```

Statement [23.3](#) contains the complete code for the visual objects added above, and the resulting canvas is shown on Figure [23.6](#).

```
import qtvisual;
CometVisualizer visu();
VisualDrawingBoard canvas1(visu.getVisualizer(), "TEST1",true);
visu.addNotebookPage(canvas1);

canvas1.autoFitContent(true);
VisualCircle c1(canvas1,20,20,10);
VisualCircle c1b(canvas1,50,50,10);
c1.setCenter(20,20);
VisualText txt1(canvas1, 20,20, "hi", "green", "Helvetica");
VisualLine (canvas1, 20,0, 20,40,1,"#ffff00",VisualLineSolid);
VisualLine (canvas1, 0,20, 40,20,1,"#ffff00",VisualLineSolid);
visu.finish();
```

Statement 23.3: Declaring a VisualDrawingBoard (visualDrawingBoard.co). See Figure [23.6](#).

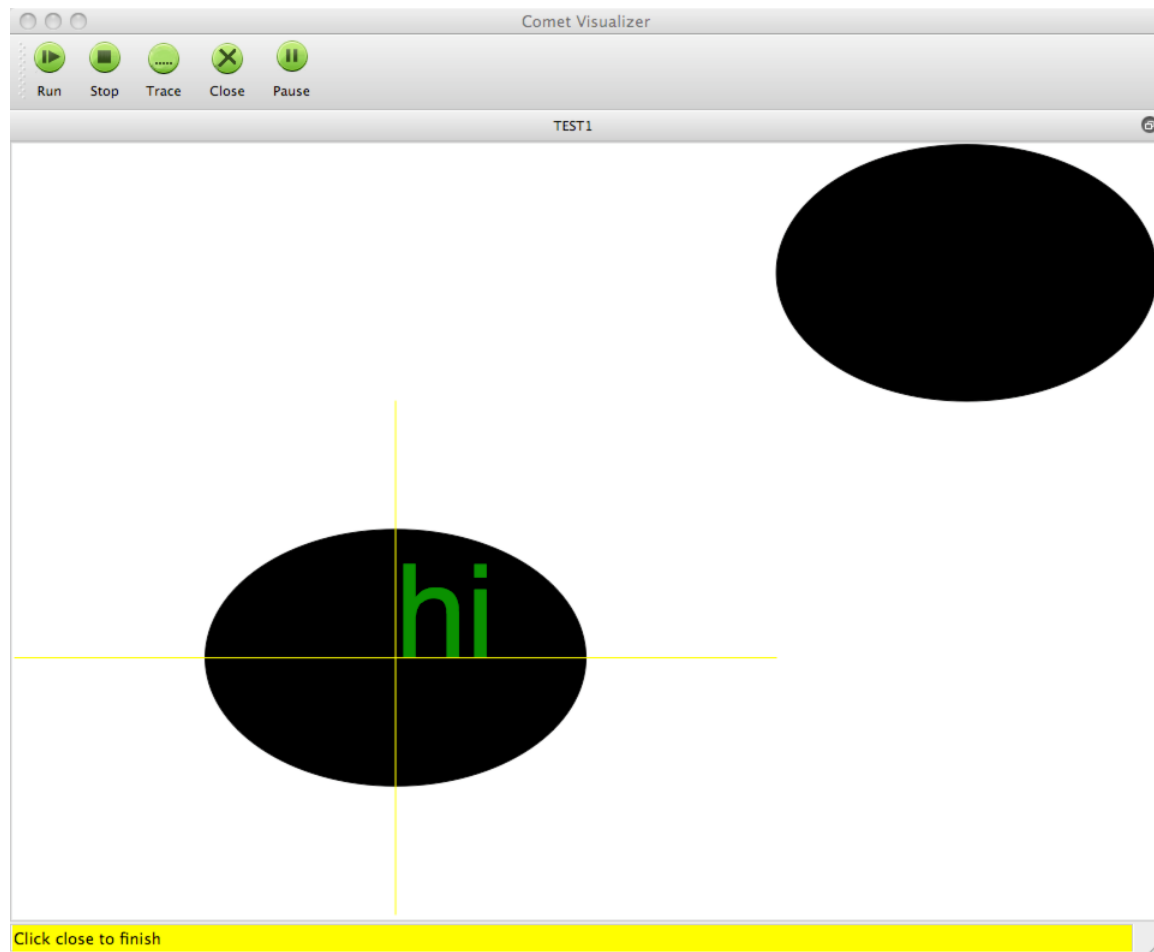


Figure 23.6: Example of a VisualDrawingBoard

Chapter 24

Visualization Events

In this chapter we show how to use event programming in connection with visualization. If you are not familiar with event programming, you should refer to Chapter 7. In COMET there is support for events related both to individual visual widgets and to general properties of the visualizer, such as events reacting to mouse input. Events can also be used to control and modify the layout of the visual interface.

24.1 Reacting to Widget Events

Events can be useful in a COMET code, if we want to add interactions with visual widgets. For example, Statement [24.1](#) first creates a button widget and then detects when the button is clicked. The widget used is the `VisualToolButton` widget, which is declared as follows:

```
VisualToolButton button(visu.getVisualizer(), "Hello World");  
visu.addNotebookPage(button);
```

Suppose that we would like to execute some code every time the button is clicked. The user can use the event `clicked()` defined for `VisualToolButton` with the instruction **whenever**, and include in the event's code block the instructions to be executed, whenever there is a click to the button.

```
whenever button@clicked() {  
    cout << "Button clicked" << endl;  
}
```

```
import qtvisual;
CometVisualizer visu();
VisualToolButton button(visu.getVisualizer(), "Hello World");
visu.addNotebookPage(button);
whenever button@clicked() {
    cout << "Button clicked" << endl;
}
visu.finish();
```

Statement 24.1: Reacting to a Clicked Button

24.2 Capturing Mouse Input

A COMET code can respond to mouse input by subscribing to events of the `Mouse` class. The `Mouse` object of a visualizer `visu` can be retrieved through the instruction `visu.getMouse()`. Suppose that we want to draw the following objects on a visual drawing board, along with a specified response to mouse input:

- a plain black circle that will update its position at every single click; the circle is re-centered at the coordinates of the mouse click
- a rectangle that can be dragged; its upper-left corner follows the position of the mouse pointer, when the user keeps the mouse button pressed and moves the pointer
- a line, one endpoint of which follows the mouse pointer

The complete code to do this is shown on Statement [24.2](#). It first creates a canvas and adds the above items into their initial position:

```
VisualDrawingBoard b(visu, "canvas");  
visu.addNotebookPage(b);  
VisualCircle c(b,0,0,5);  
VisualRectangle r(b,0,0,10,16,4);  
VisualLine l(b, 0, 0, 3, 3,0);
```

The code then gets a reference to the visualizer's mouse object, which is used when setting up the events that control the objects' behavior.

```
Mouse m = visu.getMouse();
```

We now show how to put in place these events. The plain black circle moves every time the mouse button is pressed:

```
whenever m@click(int x, int y) {  
    c.setCenter(x,y);  
}
```

The line's endpoint is updated whenever the mouse moves:

```
whenever m@mmove(int x, int y) {  
    l.setEnd(x,y);  
}
```

Finally, the rectangle's top-left corner is updated whenever the mouse is dragged:

```
whenever m@mdrag(int x, int y) {  
    r.setCorner(x,y);  
}
```

```
import qtvisual;
Visualizer visu();
VisualDrawingBoard b(visu, "canvas");
visu.addNotebookPage(b);
VisualCircle c(b,0,0,5);
VisualRectangle r(b,0,0,10,16,4);
VisualLine l(b, 0, 0, 3, 3,0);
Mouse m = visu.getMouse();

whenever m@click(int x, int y) {
    c.setCenter(x,y);
}
whenever m@mmove(int x, int y) {
    l.setEnd(x,y);
}
whenever m@mdrag(int x, int y) {
    r.setCorner(x,y);
}
sleepUntil visu@exiting();
```

Statement 24.2: Capturing Mouse Events (mouseEvents.co)

24.3 Updating an Interface Using Events

This section explains how a visual interface can be updated using events. Suppose you have an object representing a game board. One would like to update the interface each time the board object is updated. How can we separate the code for the board itself from the code that updates of the visualization?

This can be done using events, as demonstrated in the following example, shown on Statements 24.3 through 24.6. Statements 24.3 and 24.4 show how to create a class to represent the board. Statement 24.5 contains the code of the class responsible for the visualization, while Statement 24.6 gives an example of moving the game board pieces. The complete source code can be found in the file `playBoard.co`.

Game Board Representation The game board is represented by a class `PlayBoard`, which also deals with representing the pieces of two teams, A and B, and their placement on the board. The main class members, the constructor and some getter functions are shown on Statement 24.3, but the most interesting part of the class is shown on Statement 24.4, that deals with piece placement and sets up related events. There are two types of events used in this example, declared as follows:

```
Event changesA(int oldx, int oldy, int x, int y);  
Event changesB(int oldx, int oldy, int x, int y);
```

This declaration creates an event associated with class `PlayBoard`. An event `changesA` is generated every time the position of a piece of team A changes from the old coordinates, `oldx` and `oldy`, to the new coordinates, `x` and `y`. Events of type `changesB` are analogous. The class `PlayBoard` is responsible for notifying whenever the position of a piece changes. This is usually done in the setters of a class, which is also the case here. The COMET instruction **notify** generates the declared event.

```
void setPiecesPositionA(int id, int x, int y) {  
    if (x != posA_x[id] || y != posA_y[id]) {  
        notify changesA(posA_x[id], posA_y[id], x, y);  
        posA_x[id] = x;  
        posA_y[id] = y;  
    }  
}
```

```

class PlayBoard {
    range _X;           // column range
    range _Y;           // row range
    int   _nbPiecesA;    // number of team A pieces
    int   _nbPiecesB;    // number of team B pieces
    int[] posA_x;
    int[] posA_y;        // (x,y) positions of team A pieces
    int[] posB_x;
    int[] posB_y;        // (x,y) positions of team B pieces

    PlayBoard(range X, range Y, int nbPiecesA, int nbPiecesB) {
        _X = X;
        _Y = Y;
        _nbPiecesA = nbPiecesA;
        _nbPiecesB = nbPiecesB;
        posA_x = new int[1..nbPiecesA] = -1;
        posA_y = new int[1..nbPiecesA] = -1;
        posB_x = new int[1..nbPiecesB] = -1;
        posB_y = new int[1..nbPiecesB] = -1;
    }

    range getRangeX()    { return _X; }
    range getRangeY()    { return _Y; }
    int   getNbPiecesA() { return _nbPiecesA; }
    int   getNbPiecesB() { return _nbPiecesB; }

    [...]

```

Statement 24.3: Updating a Playing Board Using Events: I. Playboard Class (1/2) (playBoard.co)

```

[...]

Event changesA(int oldx, int oldy, int x, int y);
Event changesB(int oldx, int oldy, int x, int y);

// return true, if position (x,y) has a piece of team A
boolean isInA(int x, int y) {
    forall (i in 1..nbPiecesA : posA_x[i] == x && posA_y[i] == y)
        return true;
    return false;
}

// return true, if position (x,y) has a piece of team B
boolean isInB(int x, int y) {
    forall (i in 1..nbPiecesB : posB_x[i] == x && posB_y[i] == y)
        return true;
    return false;
}

// place team A piece with given id at position (x,y)
void setPiecesPositionA(int id, int x, int y) {
    if (x != posA_x[id] || y != posA_y[id]) {
        notify changesA(posA_x[id],posA_y[id],x,y);
        posA_x[id] = x;
        posA_y[id] = y;
    }
}

// place team B piece with given id at position (x,y)
void setPiecesPositionB(int id, int x, int y) {
    if (x!=posB_x[id] || y!=posB_y[id]) {
        notify changesB(posB_x[id],posB_y[id],x,y);
        posB_x[id] = x;
        posB_y[id] = y;
    }
}
}

```

Statement 24.4: Updating a Playing Board Using Events: I. Playboard Class (2/2) (playBoard.co)

Adding Visualization Once a class is augmented with events, it is easy to separate the visualization code from the class itself. For instance, suppose that we want to represent the chessboard as a `VisualTextTable`. Since pieces are all the same, pieces of team A are denoted on the board with the string “A” and pieces of team B with a string “B”. The `VisualTextTable` entries are text, so we use the method `setLabel()` to place the pieces.

Statement 24.5 defines a class `Visualization` that takes care of the visualization. The class constructor takes as parameter a `PlayBoard` object and puts in place an automatically updated visualization for it. The constructor of `Visualization` creates the main window and saves a reference to the board object. It then creates a `VisualTextTable` with the same number of rows and columns as the board, and adds it to the main window.

The interesting part concerns the update of the visual text table: some code is executed every time a `change` event pops up in the `PlayBoard` class. The last position is erased, unless a pieces of the other team is present, and the label of the new position is updated. Notice that the visualization and the update of the visualization are clearly separated from the functional code. Apart from event declaration, there is no need to add specific visualization code inside the `PlayBoard` class.

```

import qtvisual;
class Visualization {
    PlayBoard      _playboard;
    CometVisualizer _visu;
    Visualization(PlayBoard pb) {
        _playboard = pb;
        _visu = new CometVisualizer ();
        VisualTextTable textTable(_visu.getVisualizer(),"Board",
                                   _playboard.getRangeX(),_playboard.getRangeY());
        _visu.addNotebookPage(textTable);

        whenever _playboard@changesA(int oldx,int oldy, int x, int y) {
            if (oldx > 0 && oldy > 0 && (!_playboard.isInB(oldx,oldy)))
                textTable.setLabel(oldx,oldy,"");
            textTable.setLabel(x,y,"A");
        }

        whenever _playboard@changesB(int oldx,int oldy, int x, int y) {
            if (oldx > 0 && oldy > 0 && (!_playboard.isInA(oldx,oldy)))
                textTable.setLabel(oldx,oldy,"");
            textTable.setLabel(x,y,"B");
        }
    }
    void finish() { _visu.finish(); }
}

```

Statement 24.5: Updating a Playing Board Using Events: II. Visualization Class (playBoard.co)

Animating Piece Movement Let us now animate the pieces on the board, through the setters of the `PlayBoard` class. Statement 24.6 shows a way to do that. We first declare the board and its visualization:

```
boolean useVisualization = true;
int n = 4;
PlayBoard playboard(1..n,1..n,n,n);
if (useVisualization)
    Visualization visu(playboard);
```

Notice how the visualization is textually separated from the declaration of `playboard`. We then add animation to the board, by simply changing the position of the pieces through the setters of the `playboard` object:

```
int posAx = 1;
int posBx = playboard.getRangeY().getUp();
while (posBx!=0) {
    forall (i in 1..playboard.getNbPiecesA())
        playboard.setPiecesPositionA(i,posAx,i);
    posAx++;
    forall (i in 1..playboard.getNbPiecesB())
        playboard.setPiecesPositionB(i,posBx,i);
    posBx--;
    sleep(1000); // wait for one second
}
```

The top row is filled with pieces of team A, while the bottom row is filled with pieces of team B. The pieces of each team are moved in line, until they reach the opposite side of the board. The visualization simply reacts to the changes of the state stored in the object `playboard`. Figures 24.1 show how the visualization progresses.

Now suppose that we would like to replace this interface with a fancier visualization of the board using an image. For example, we could consider a `VisualDrawingBoard` with a `VisualGrid`. This can be done without touching the core class `PlayBoard`: it suffices to write a new class `VisualizationDrawingBoard` that would use events for updating the drawable objects.

```
boolean useVisualization = true;
int n = 4;
PlayBoard playboard(1..n,1..n,n,n);
if (useVisualization)
    Visualization visu(playboard);

int posAx = 1;
int posBx = playboard.getRangeY().getUp();
while (posBx!=0) {
    forall (i in 1..playboard.getNbPiecesA())
        playboard.setPiecesPositionA(i,posAx,i);
    posAx++;
    forall (i in 1..playboard.getNbPiecesB())
        playboard.setPiecesPositionB(i,posBx,i);
    posBx--;
    sleep(1000); // wait for one second
}
```

Statement 24.6: Updating a Playing Board Using Events: III. Moving Pieces on the Board
(playBoard.co)

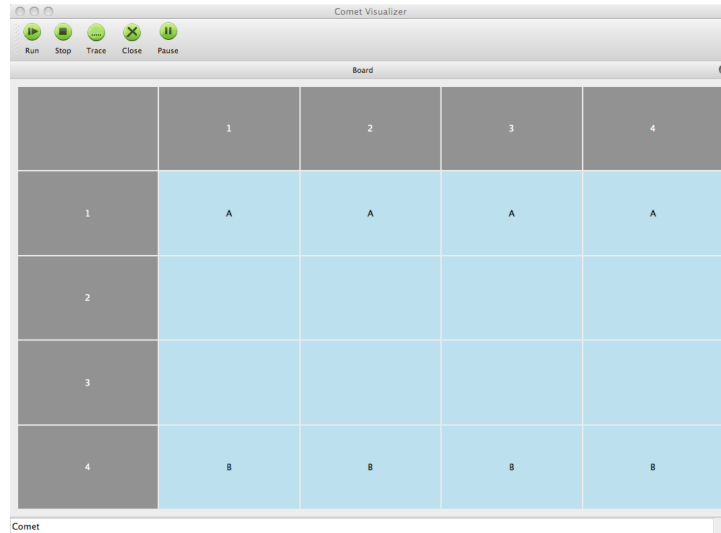


Figure 24.1a: Example of an animated board. The visualization is updated whenever there is a change to the state of the `PlayBoard` object (Initial configuration)

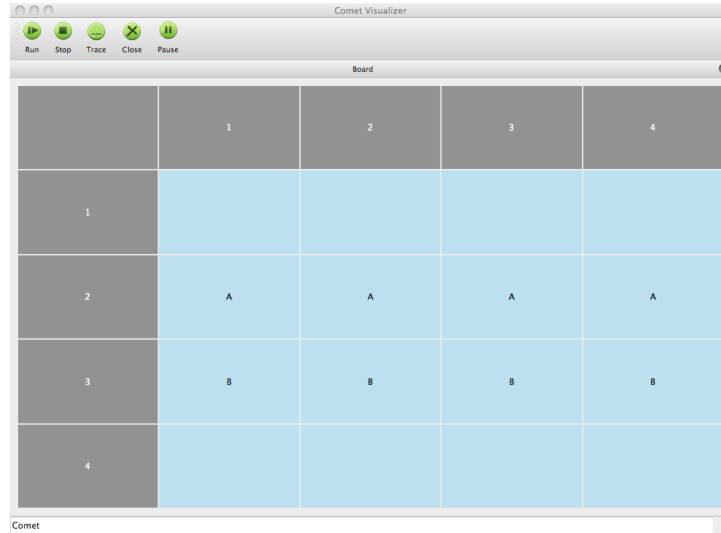


Figure 24.1b: Example of an animated board: After step 1

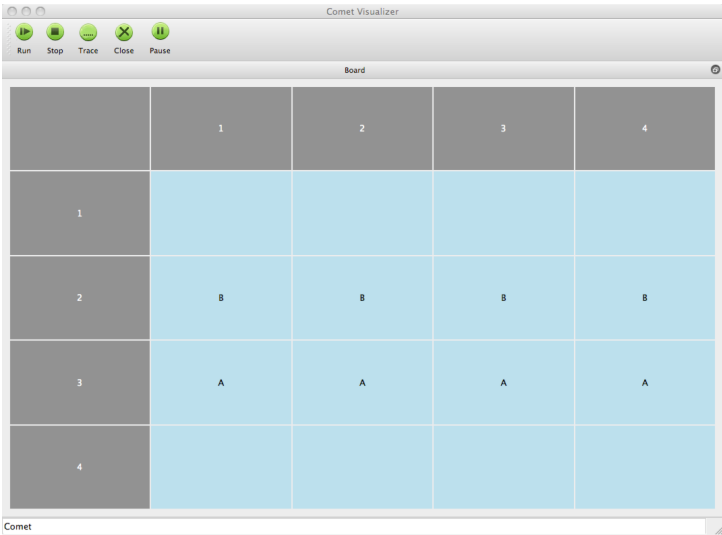


Figure 24.1c: Example of an animated board: After step 2

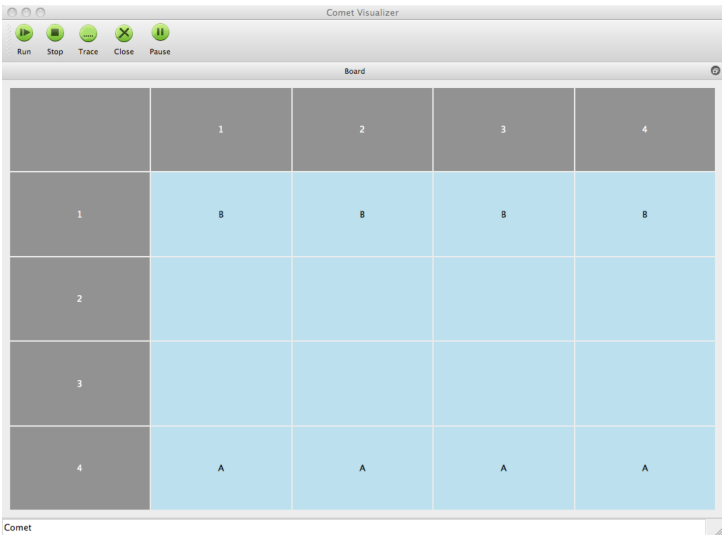


Figure 24.1d: Example of an animated board: Final state

Chapter 25

Visualization Examples

One of the key features of COMET is the ability to visualize Constraint Programming and Local Search models in a simple, textually separated and declarative way. Visualization is a very useful tool in the initial design phase of CP or CBLS models, especially in cases where fine-tuning is necessary, or when one wants to have an idea of the most or least active areas of the solution space. In this chapter, we demonstrate how visualization can be easily built for CP and CBLS, through three representative examples.

25.1 Visualization for the Queens Problem (CBLS)

In this section, facilities from the `CometVisualizer` class are used in order to quickly generate a visualization for the local search solution to the Queens problem described in Section 18.1. Statement 25.1 extends the CBLS model of that section (Statement 18.1) to include visualization.

A natural visual interface for the Queens is a `VisualTextTable` and a `2DPlot`: the first one to represent the chess-board and the second one to show how violations change over time. However, COMET provides an easier way to visualize this model. The actual code consists of just four lines:

```
CometVisualizer visu();  
visu.animate(queen,S,"queens");  
visu.pauseOn(queen);  
visu.plotViolations(S.violations(),queen);
```

Since arrays of variable are very common, `CometVisualizer` offers a method `animate()` to show the assignment of an array of N variables on a $D \times N$ grid, where D is the size of the domain. All variables are assumed to have the same domain.

```
visu.animate(queen,S,"queens");
```

Because the animation of `VisualTextTable` is encapsulated, `CometVisualizer` also provides a method `pauseOn`, that pauses whenever there is a change into its input array of variables.

```
visu.pauseOn(queen);
```

`CometVisualizer` also offers ways to plot the evolution of a variable upon successive moves:

```
visu.plotViolations(S.violations(),queen);
```

The first argument is the plotted variable; the second argument is the variable array that the first argument depends on. Figure 25.1 shows the resulting interface.

```

import cotls;
import qtvisual;
int    n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);
S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size)(queen[i] + i)));
S.post(alldifferent(all(i in Size)(queen[i] - i)));
m.close();

CometVisualizer visu();
visu.animate(queen,S,"queens");
visu.pauseOn(queen);
visu.plotViolations(S.violations(),queen);

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
    selectMax (q in Size) (S.violations(queen[q]))
        selectMin (v in Size) (S.getAssignDelta(queen[q],v))
            queen[q] := v;
    it++;
}

visu.finish();

```

Statement 25.1: CBLS Model for the Queens Problem with Visualization ([queensLS-visu.co](#)). See Figure 25.1 for the resulting interface.

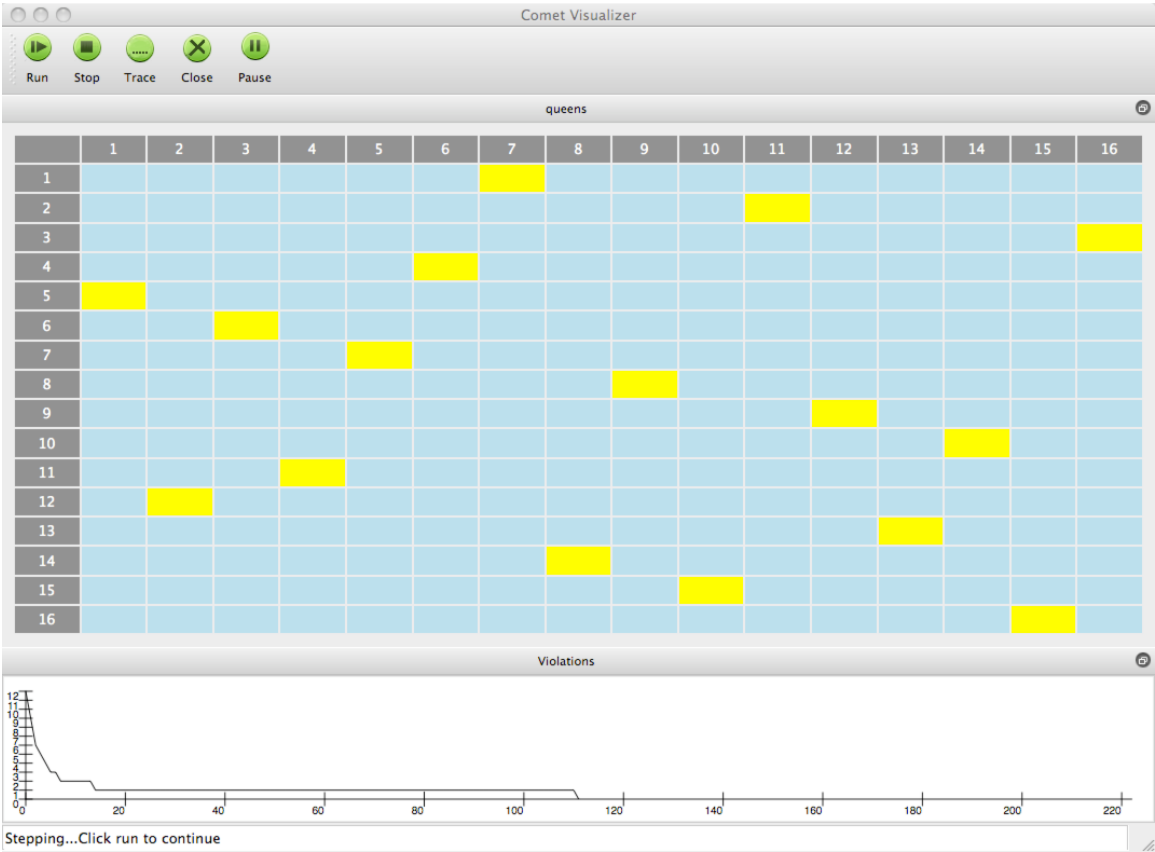


Figure 25.1: Visualization for the Queens Problem (CBLS)

25.2 Animated Visualization for Time Tabling (CBLS)

This section builds a visualization component for the time tabling problem described in Section 19.4. The code for a CBLS solution to that problem can be found on Statements 19.2, 19.3, and 19.4. Here is a summary of the model:

```
import cotls;
Solver<LS> ls = new Solver<LS>();
ConstraintSystem<LS> S(ls);

var{int} X[Rooms,Slots](ls,EventsExtended);
forall (s in Slots, r1 in Rooms, r2 in Rooms : r1 < r2)
    S.post(conflictMatrix[X[r1,s],X[r2,s]] == 0);

S.close();
ls.close();

set{int} availableEvents = collect(e in EventsExtended) e;
forall (r in Rooms) by (possEventsInRoom[r].getSize())
    forall (s in Slots)
        selectMin (e in availableEvents inter possEventsInRoom[r])
            (possRoomsOfEvent[e].getSize()) {
            X[r,s] := e;
            availableEvents.delete(e);
        }

while (S.violations() > 0)
    select (r1 in Rooms, s1 in Slots : S.violations(X[r1,s1]) > 0)
        selectMin (r2 in Rooms : possEventsInRoom[r2].contains(X[r1,s1]),
            s2 in Slots : possEventsInRoom[r1].contains(X[r2,s2]))
            (S.getSwapDelta(X[r1,s1],X[r2,s2]))
            X[r1,s1] := X[r2,s2];
```

The decision variables consist of a matrix $X[,]$ that assigns events to rooms and slots. The complete code is included in file `timetableLS.co`, that reads data from file `timetable.data`. Statement 25.2 contains the code that creates the animated visualization interface shown on Figure 25.2.

```

// VISUALIZER
import qtvisual;
CometVisualizer visu();
VisualDisplayTable T = visu.getDisplayTable("Time-Tabling Problem",Rooms,Slots);
forall (r in Rooms,s in Slots) {
    if (S.violations(X[r,s]) > 0)
        T.setColor(r,s,"red");
    else
        T.setColor(r,s,"green");
    whenever S.violations(X[r,s])@changes(int a,int b)
        if (b > 0)
            T.setColor(r,s,"red");
        else
            T.setColor(r,s,"green");
}
visu.plotViolations(S.violations(),X);
visu.pause();

```

Statement 25.2: CBLS Model for Time-Tabling: IV. Visualization

Visualization is built using a `VisualDisplayTable` widget `T`, each entry of which represents a room-slot pair of the timetable. The widget parameters are simply a string for the table's title and a range for each dimension.

```
VisualDisplayTable T = visu.getDisplayTable("Time-Tabling Problem",Rooms,Slots);
```

We then paint each entry (r,s) of the display table red or green, depending on whether the corresponding entry $X[r,s]$ of the timetable matrix has violations. The initial coloring takes place right after finding an initial timetable, using a loop that goes through all rooms and slots:

```
forall (r in Rooms,s in Slots) {
  if (S.violations(X[r,s]) > 0)
    T.setColor(r,s,"red");
  else
    T.setColor(r,s,"green");
}
```

Each entry of matrix X is checked for violations, and the corresponding display table entry is painted with the appropriate color using the method `setColor` available for visual display tables. In order to create an animation effect, the same loop installs, for each entry (r,s) , an event handler, that gets activated every time there is a change in the number of violations in the corresponding position of the timetable. The event defined is a `changes` event used with the `whenever` instruction.

```
whenever S.violations(X[r,s])@changes(int a,int b)
```

The code block of the event is similar to the code that performs the initial coloring. The visualization code concludes with an instruction for plotting the total number of violations during the search:

```
visu.plotViolations(S.violations(),X);
```

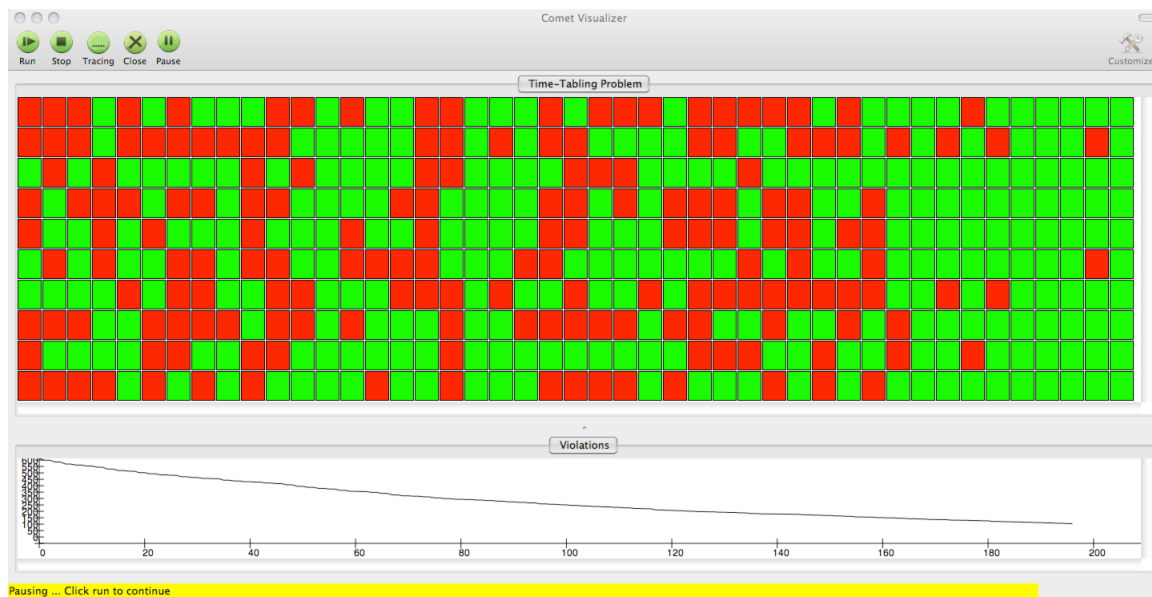


Figure 25.2: Visualization for a time-tabling problem. It displays a table representing room-slot pairs of the timetable. Red cells correspond to timetable entries with violations. In the bottom, there is a plot of the total number of violations during the search.

25.3 Visualization for Job Shop Scheduling (CP)

This section builds a visualization for the Job Shop problem, described in Section 16.1. In the Job Shop problem, we are given a set of jobs, each of which consists of a sequence of tasks to be executed on some machine. Each machine can execute one task at a time and the goal is to minimize the project completion time (the makespan). We use the model of Statement 16.1, which is summarized here:

```
import cotfd;
range Jobs;                      // read from file
range Tasks;                     // read from file
range Machines = Tasks;
int duration[Jobs,Tasks]; // read from file
int machine[Jobs,Tasks]; // read from file

int horizon = sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP> cp(horizon);
Activity<CP> a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP> makespan(cp,0);
UnaryResource<CP> r[Machines](cp);

minimize<cp>
    makespan.start()
subject to {
    forall (j in Jobs, t in Tasks : t != Tasks.getUp())
        a[j,t].precedes(a[j,t+1]);
    forall (j in Jobs)
        a[j,Tasks.getUp()].precedes(makespan);
    forall (j in Jobs, t in Tasks)
        a[j,t].requires(r[machine[j,t]]);
}
using {
    forall (m in Machines) by (r[m].localSlack(),r[m].globalSlack())
        r[m].rank();
    makespan.scheduleEarly();
}
```

This section suggests a visualization that shows the ordering between jobs, tasks, and machine usage, combined with a plot of the makespan over time. A representation commonly used in scheduling is the Gantt chart. For the visualization of the scheduling, we are going to use a specialized Gantt chart widget called `VisualGantt`. Statements 25.3 and 25.4 show the code for a class `Visualization`, that is used for managing all information necessary for visualizing the Job-Shop model.

In what follows, we describe the key parts of this class. The class creates two Gantt charts: one to represent the schedule of the tasks within each job, and one to show the schedule on each individual machine. We first describe the chart showing the schedule within each job:

```
VisualGantt gantt = _visu.getGantt("Job",_Jobs,_Machines,1500,"gray");
_visu.addNotebookPage(gantt);
```

The first argument of the `VisualGantt` constructor associates a name with the Gantt chart; the second argument is the range of jobs, that correspond to the Gantt chart rows; the third argument, `_Machines`, is ignored in this chart; the fourth argument gives the maximum displayed value on the horizontal axis; the last argument specifies the background color.

The next step is to associate with each row the set of activities to be displayed there. First, we switch to buffering mode. While in buffering mode, updates to the visualization are performed in batch, resulting to faster animation and faster initialization.

```
gantt.setBuffering(true);
```

Then, we introduce an array of `VisualActivity` widgets to visualize the tasks. In the CP scheduling framework, the class `VisualActivity` is the visual counterpart of the `Activity` class. `VisualActivity` objects can be added to Gantt charts. In this example, visual activities are declared as follows:

```
_vact = new VisualActivity[j in _Jobs,m in _Machines]
    (gantt, j, _act[j,m].getEST(), _act[j,m].getMinDuration(),
     cm[_machine[j,m]], IntToString(_machine[j,m]));
```

```

class Visualization {
    range                _Jobs;
    range                _Tasks;
    range                _Machines;
    Activity<CP>[,]      _act;
    int[,]               _machine;
    int                  _best;

    CometVisualizer      _visu;
    VisualActivity[,]     _vact;
    VisualActivity[,]     _vactm;

    //update the position of the visual activities
    void update() {
        forall (j in _Jobs,m in _Machines) {
            //set the start time of the earliest possible time
            _vact[j,m].setRelease(_act[j,m].getEST());
            _vactm[j,m].setRelease(_act[j,m].getEST());
        }
    }
    void finish() {
        update();
        _visu.finish();
    }
    [...]
}

```

Statement 25.3: Visualization for Job Shop Using Gantt Charts (1/2). See Figure [25.3](#)

```

[...]
Visualization(var<CP>{int} makespan,Activity<CP>[,] act,int[,] machine) {
    _Jobs      = act.getRange(0);
    _Tasks     = act.getRange(1);
    _Machines  = _Tasks;
    _act       = act;
    _machine   = machine;
    _best      = System.getMAXINT();

    string color[0..9]      = ["wheat","LightBlue","white","green","cyan",
                                "red","blue","yellow","brown","orange"];
    string cm[m in _Machines] = color[m % 10];
    _visu = new CometVisualizer();
    VisualGantt gantt = _visu.getGantt("Job",_Jobs,_Machines,1500,"gray");
    _visu.addNotebookPage(gantt);
    gantt.setBuffering(true);
    _vact = new VisualActivity[j in _Jobs,m in _Machines]
        (gantt, j, _act[j,m].getEST(), _act[j,m].getMinDuration(),
         cm[_machine[j,m]], IntToString(_machine[j,m]));
    gantt.setBuffering(false);

    VisualGantt ganttm = _visu.getGantt("Machine",_Machines,_Jobs,1500,"wheat");
    _visu.addNotebookPage(ganttm);
    ganttm.setBuffering(true);
    _vactm = new VisualActivity[j in _Jobs,m in _Machines]
        (ganttm, _machine[j,m], _act[j,m].getEST(), _act[j,m].getMinDuration(),
         cm[_machine[j,m]], IntToString(j));
    ganttm.setBuffering(false);

    Visual2DPlot plot = _visu.get2DPlot("Makespan",500,1500);
    _visu.addBottomNotebookPage(plot);

    Integer it(0);
    whenever makespan@onValueBind(int val) {
        update();
        plot.addPoint(it.getValue(),val);
        gantt.setMakespan(val);
        ganttm.setMakespan(val);
        it++;
    }
    _visu.pause();
}

```

Statement 25.4: Visualization for Job Shop Using Gantt Charts (2/2). See Figure [25.3](#)

A visual activity is created for each job and machine. In the constructor of `VisualActivity`, the first argument is the Gantt chart on which it is contained, while the second argument gives the row number; the third and fourth arguments specify the earliest possible starting time and the minimum duration, respectively, of the activity associated with job j and task m ; the fifth argument is the activity's color, and, finally, the sixth argument indicates the machine associated with the activity. Once the visual activities have been allocated, we can exit from buffering mode:

```
gantt.setBuffering(false);
```

The visualization also creates a Gantt widget to show the schedule from the point of view of each individual machine, i.e., the sequence of activities on each machine:

```
VisualGantt ganttm = _visu.getGantt("Machine",_Machines,_Jobs,1500,"wheat");
_visu.addNotebookPage(ganttm);
```

A new set of visual activities is created for the tasks, in connection to the new Gantt chart:

```
_vactm = new VisualActivity[j in _Jobs,m in _Machines]
    (ganttm, _machine[j,m], _act[j,m].getEST(), _act[j,m].getMinDuration(),
     cm[_machine[j,m]], IntToString(j));
```

This time, the row corresponds to the id, `_machine[j,m]`, and the string label displayed on the activities is the associated job id. Note that, in this particular case, there is no need to create special events to deal with the activities: the `VisualActivity` class takes care of the related events behind the scenes, and automatically updates the Gantt chart.

The last part of the `Visualization` class deals with plotting the makespan over time. The next two lines define a two-dimensional plot widget:

```
Visual2DPlot plot = _visu.get2DPlot("Makespan",500,1500);  
_visu.addBottomNotebookPage(plot);
```

Every time the makespan activity is assigned a value, a new point is added to the plot, and a dedicated method `update` is called, to update the visual activities:

```
Integer it(0);  
whenever makespan@onValueBind(int val) {  
    update();  
    plot.addPoint(it.getValue(),val);  
    gantt.setMakespan(val);  
    ganttm.setMakespan(val);  
    it++;  
}
```

A reasonable question here is why the local variable `it` is of type `Integer` instead of the primitive type `int`. The answer is that the block code following the **whenever** keyword is a closure, and the environment captured by closures is the stack. Therefore, the integer `it` must be an object (allocated on the heap), otherwise its value would be reset to zero, each time the closure associated with `onValueBind` is called. This behavior of closures is explained in more detail in [Section 6.1](#).

A snapshot of the resulting visual interface can be seen in [Figure 25.3](#). The complete source code can be found in the file `jobshopDemo.co`.

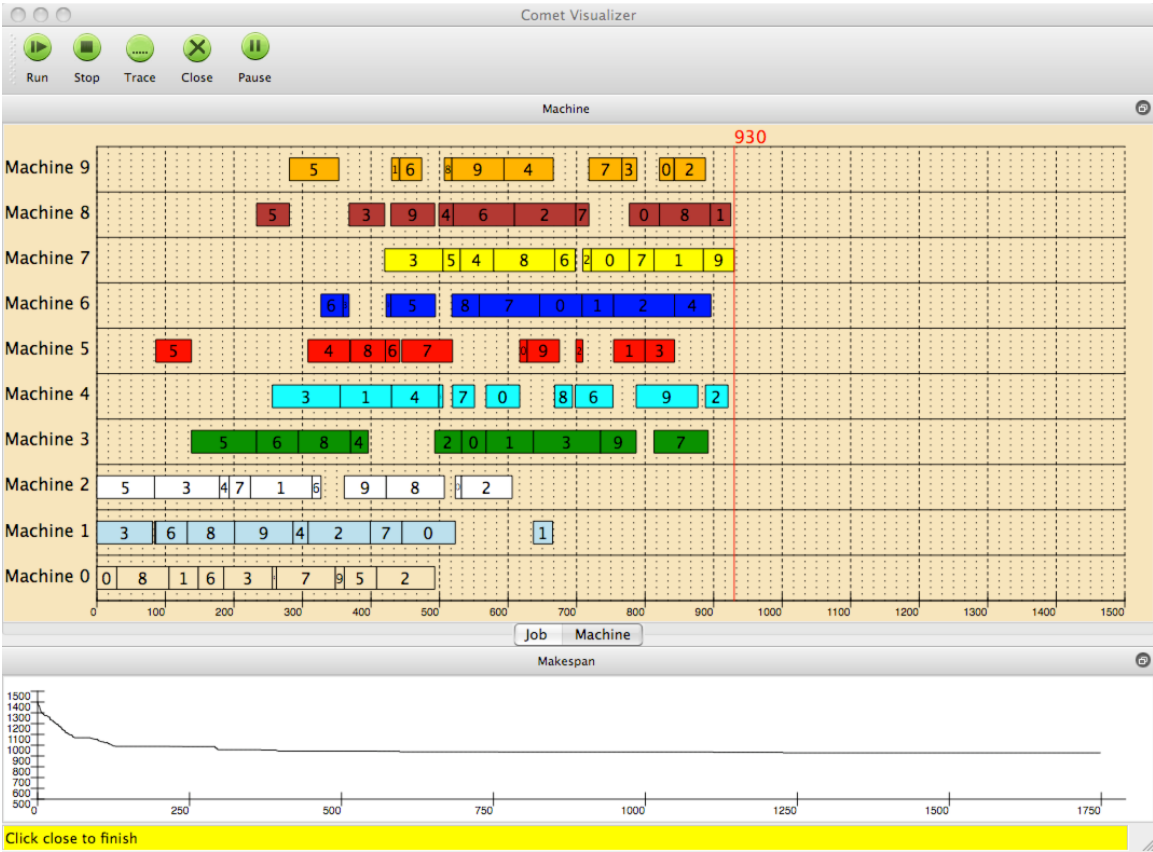


Figure 25.3: Visualization for Unary Job Shop Using Gantt Charts (Snapshot)

VI INTERFACING COMET

Chapter 26

Database Interface

The database connection module, `cotdb`, allows COMETcode to access data stored in an external database. The API includes methods to execute SQL queries on the database and to retrieve the results. Currently, the `cotdb` module only supports an Open Database Connectivity (ODBC) engine. Most modern databases provide an ODBC driver.

26.1 Setting up ODBC

Before connecting to a database, the relevant ODBC driver must be installed. This is typically provided by the vendor of the database. Next, the ODBC data source must be set up. On Windows this is done using the ODBC Data Source Administrator available through the Administrator Tools in the Control Panel. On Mac OS, a GUI is provided by the ODBC Administrator tool in /Applications/Utilities.

On Linux (and Mac OS) there are configuration files for setting up the ODBC Data source. For instance, after installing the MySQL ODBC driver, the configuration file `/etc/odbc.ini`, or `~/.odbc.ini` may include the following information:

```
[testMYSQL]
Driver      = MySQL
Description = testing MySQL
Database    = testing
Password    = my-secure-password
Server      = my-server
User        = my-user-id
```

Instead of putting the User and Password in the configuration file, these can be included in the database connection string described later. The Data Source Name (DSN), `testMYSQL` in the above example, is the name that ODBC will use to identify the data source. The Description is text for documentation purposes. The Server is the name or IP address of the host running the database. The User and Password are the credentials used to connect to the database. The Database field contains the schema that will be made available through this Data Source. GUI tools, such as the ODBC Data Source Administrator on Windows, have fields similar to those shown in the configuration file example above.

The HTML Documentation included with COMET provides examples and screenshots for installing and setting up ODBC. Navigate to the Get Started page and click on the Database Setup link.

26.2 Connecting to the Database

The two main classes for establishing a connection are the `DBEngine` and `DBConnection` classes, shown in the example below. The `DBEngine` class is used to specify the database protocol to use. Currently only the `'odbc'` argument is supported.

Next, the connection is established by passing the connection string to the `connect` method of `DBEngine`. This returns a `DBConnection` object that can be used to execute queries. In this example, the connection string contains the DSN, the User and the Password. Each *key=value* pair ends with a semicolon. If the User and Password are specified in the configuration file, then they can be left out of the connection string.

```
import cotdb;
DBEngine db("odbc");
DBConnection dbc =
    db.connect("DSN=testMYSQL;User=my-user-id;Password=my-secure-password;");
```


26.3 Performing Queries

The `execute` method of the `DBConnection` object executes a query a single time returning a boolean indicating whether the query succeeded. Queries executed in this way should not take any parameter or produce results. Instead, this method is meant for queries that have side effects on the database. For instance, the following example inserts a row into an already existing table named `student`.

```
bool ok=dbc.execute("insert into student (id,name,score) values (1,'Alice',0.90)");
```

Preparing Queries For queries that have parameters or produce results, the `prepare` method should be used. This method is used for preparing queries that will be used more than once. For example, the following code creates a query that selects every row from the `student` table:

```
DBStatement allStudents = dbc.prepare("select * from student");
```

Parameterized Queries To set parameters in a query, you should place a `'?'` character in the query string and call the `setParam` method of the `DBStatement`. For example, the following code selects from the `student` table the rows for which `score` is at most equal to a value specified by a parameter. The parameter is set to 0.7 for the particular query execution, using `setParam`.

```
DBStatement badScores = dbc.prepare("select * from student where score <= ?");  
badScores.setParam(1,0.7).execute();
```

The first argument to `setParam` is the index of the parameter starting from 1. The second argument determines the value for the parameter, and can be of type `int`, `float`, `string`, `Date`, or `Time`.

26.4 Retrieving Data

For queries that produce results, data is returned using a `DBRow` object through the `fetch` method of `DBStatement`. The `fetch` method returns `null`, when there are no more rows to retrieve. The following code uses the prepared `select` statement shown above to print every row returned:

```
DBStatement badScores = dbc.prepare("select * from student where score <= ?");
badScores.setParam(1,0.7).execute();
DBRow r = badScores.fetch();
cout << "Bad Scores:" << endl;
while (r != null) {
    cout << r.getInt(1)
        << " | " << r.getString(2)
        << " | " << r.getDouble(3) << endl;
    r = badScores.fetch();
}
```

The `DBRow` class includes methods `getInt`, `getDouble`, `getString`, `getDate`, `getTime` to retrieve the data as a `int`, `float`, `string`, `Date`, or `Time`, respectively.

After calling `fetch`, any previous `DBRow` instance may have an undefined internal state. Maintaining references to returned `DBRow` objects can cause errors. Instead, any returned data should be copied into a separate data structure. For example, the following code is incorrect:

```
DBStatement allStudents = dbc.prepare("select * from student");
DBRow r = allStudents.fetch();
queue<DBRow> allrows();      // WRONG!
while (r != null) {
    allrows.enqueueBack(r);  // WRONG!
    r = allStudents.fetch();
}
```

In the above example, the behavior is undefined and may lead into a situation, in which all queued rows refer to the same internal data. Instead, this could be rewritten as follows:

```
DBStatement allStudents = dbc.prepare("select * from student");
DBRow r = allStudents.fetch();
tuple StudentData { int id; string name; float score; }
queue<StudentData> allrows2();
while (r != null) {
    allrows.enqueueBack(StudentData(r.getInt(1),r.getString(2),r.getDouble(3)));
    r = allStudents.fetch();
}
```

The data is now copied into a tuple `StudentData` and can be safely used later in the code.

Chapter 27

XML Interface

The `cotxml` module allows the user to interface COMET programs with eXtensible Markup Language (XML) files. It exposes classes that represent and closely follow the Document Object Model (DOM) Level 1 Specification, as described in:

<http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html>

This chapter describes how to read and write to XML files, illustrating the methods available to this end through a small working example, introduced in the beginning of the chapter.

27.1 Sample XML File

We start by presenting a simple XML file that we will serve as a running example throughout this section. The XML file presents a declaration, a comment and then a root element tagged **Delivery**. Inside the root element there are two main elements: **Products** (with children **Product**) and **Route**. The latter presents a list of **Depot** and/or **City** both having as possible child elements **Load** or **Unload**.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!--Created by COMET/XML-->
<Delivery>
  <Products>
    <Product pID="A" revenue="35">Product A from main factory</Product>
    <Product pID="B" revenue="15" maxQty="100">Product B outsourced</Product>
  </Products>
  <Route>
    <Depot ID="HQ">
      <Load pID="A" qty="100"/>
      <Load pID="B" qty="50"/>
    </Depot>
    <City ID="Providence">
      <Unload pID="A" qty="50"/>
    </City>
    <City ID="Boston">
      <Unload pID="A" qty="50"/>
      <Load pID="B" qty="50"/>
    </City>
    <City ID="Manchester">
      <Unload pID="B" qty="50"/>
    </City>
    <Depot ID="Albany"/>
  </Route>
</Delivery>
```

Attributes are defined for the elements **Product**, **City**, **Depot**, **Load** and **Unload**. The document is depicted as a DOM in Figure [27.1](#)

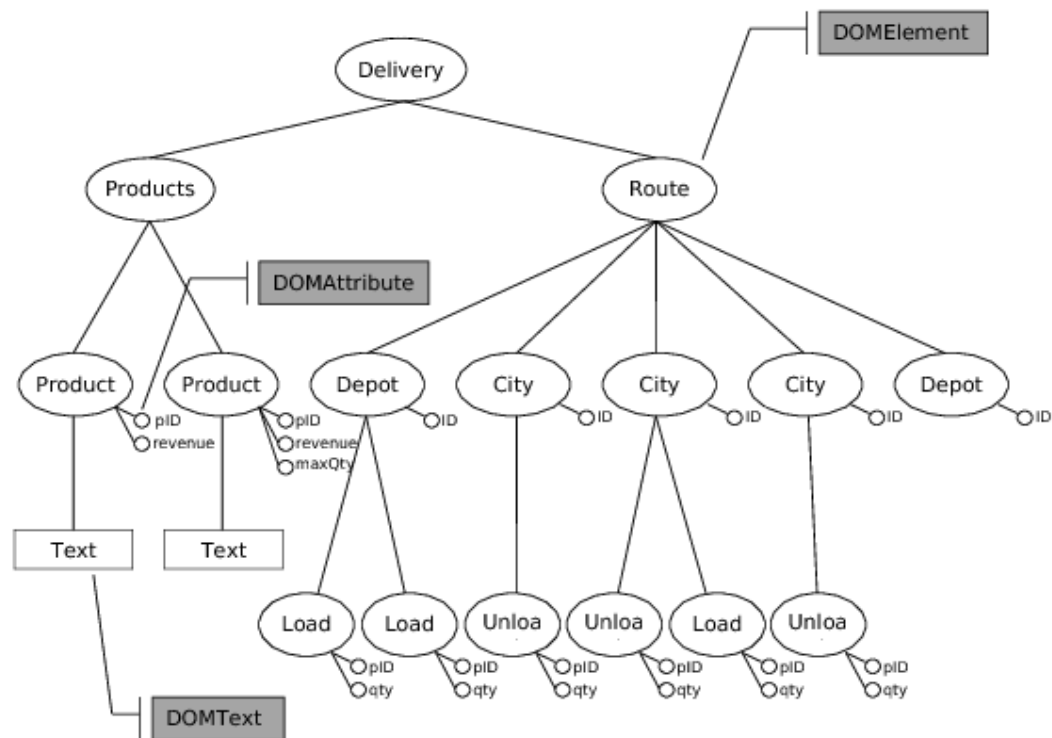


Figure 27.1: The Document Object Model of the Example File

27.2 Reading from an XML File

To give a walk-through of the XML interface, we first describe two simple examples that read data from the file defined in the previous section. The first example prints the list of depots and cities the route passes through, while the second one prints product information.

Reading Depots and Cities Statement 27.1 shows the code to read and print the depots and cities of the route. It first imports the XML interface module, `cotxml` and then loads the XML file. The access point to an XML file is the COMET class `DOMDocument`, whose constructor receives the name of the file to parse. Once a `DOMDocument` object is created, the method `documentElement()` of class `DOMDocument` is called to retrieve the root element of the document. In the current example, this is the element tagged `Delivery`.

```
import cotxml;
DOMDocument doc("example.xml");
DOMELEMENT root = doc.documentElement();
```

Note that all the classes involved in the DOM derive from `DOMNode`, which provides the navigation API to traverse the entire document: it allows to access the siblings or the children of the current node, or to add or replace nodes.

From any element, it is possible to access the first child node or the first child node with a given tag, respectively, with the methods `firstChildElement()` and `firstChildElement(string)` of class `DOMELEMENT`, respectively. If no child element exists, then the methods return `null`. These methods are used for retrieving the elements corresponding to the route and its first stop (the depot HQ in our case).

```
DOMELEMENT route = root.firstChildElement("Route");
DOMELEMENT stop = route.firstChildElement();
```

To iterate over the set of child elements, it suffices to retrieve the first one and then use the method `nextSiblingElement()` of `DOMElement` to get the next sibling. Note that the iteration goes through the child elements in the same order as they are stored in the XML file. To access an attribute of an element, we can use the method `attribute(string)` of `DOMElement`, that returns an instance of the class `DOMAttr`; this class provides the methods to retrieve the attribute's value as a **string**, an **int** or a **float**.

```
while (stop != null) {  
    cout << stop.attribute("ID").toString() << endl;  
    stop = stop.nextSiblingElement();  
}
```

The final output of the Statement [27.1](#) is the following:

```
HQ  
Providence  
Boston  
Manchester  
Albany
```

Note that it would have been easy to restrict the output to only the cities, by replacing the line that reads the first stop with:

```
DOMElement stop = route.firstChildElement("City");
```

and the line that gets the next stop inside the **while** loop with:

```
stop = stop.nextSiblingElement("City");
```

```
import cotxml;
DOMDocument doc("example.xml");
DOMELEMENT root = doc.documentElement();
DOMELEMENT route = root.firstChildElement("Route");
DOMELEMENT stop = route.firstChildElement();
while (stop != null) {
    cout << stop.attribute("ID").toString() << endl;
    stop = stop.nextSiblingElement();
}
```

Statement 27.1: Reading and Printing from a Sample XML File: Reading Route Data

Reading Products In the next example, we will read the section of the XML file regarding the products, print all related information, and give a warning, for each product that has a maximum-quantity attribute specified.

The solution is very similar to the one of the previous example, and is shown on Statement 27.2. After creating a `DOMDocument` object, this is used to get the element tagged `Products`. In order to access the child elements tagged `Product`, we use the method `elementsByTagName(string)` of class `DOMElement`; this method fetches all the descendents with the specified tag, *at any depth*, and returns them as an array.

```
DOMDocument  doc("example.xml");
DOMElement   root      = doc.documentElement();
DOMElement   products  = root.firstChildElement("Products");
DOMElement[] product   = products.elementsByTagName("Product");
```

The code then iterates over the array and prints the attributes `pID` and `revenue`.

```
DOMAttr id  = product[i].attribute("pID");
DOMAttr rev = product[i].attribute("revenue");
cout << id.toString() << ": " << description;
cout << ", rev: " << rev.toString() << endl;
```

The method `hasAttribute(string)` of class `DOMElement` can prove very handy in checking whether an element has a particular optional attribute. We use this method to check if a product has a maximum capacity, and issue a warning, if it does:

```
if (product[i].hasAttribute("maxQty")) {
    DOMAttr maxQty = product[i].attribute("maxQty");
    cout << "Warning: max quantity " << maxQty.toString() << endl;
}
```

In some applications, it may also be useful to know directly the set of attributes defined in a specific element. To do so, the class `DOMElement` provides the method `attributes()`, that returns a dictionary, whose keys are the names of the attributes and whose values are instances of `DOMAttr`.

```
import cotxml;
DOMDocument doc("example.xml");
DOMELEMENT root = doc.documentElement();
DOMELEMENT products = root.firstChildElement("Products");
DOMELEMENT[] product = products.elementsByTagName("Product");
forall (i in product.getRange()) {
    string description = product[i].toText();
    DOMAttr id = product[i].attribute("pID");
    DOMAttr rev = product[i].attribute("revenue");
    cout << id.toString() << ": " << description;
    cout << ", rev: " << rev.toString() << endl;
    if (product[i].hasAttribute("maxQty")) {
        DOMAttr maxQty = product[i].attribute("maxQty");
        cout << "Warning: max quantity " << maxQty.toString() << endl;
    }
}
```

Statement 27.2: Reading and Printing from a Sample XML File: Reading Product Data

In order to retrieve the text content of an element, for example the product description in our case, we use the method `toText()` of `DOMElement`.

```
string description = product[i].toText();
```

It is worth pointing out that this method returns all the text content of the current element and of its descendents at any depth. In case we would like to retrieve just the text of the current element, discarding the text of its descendants, we would have to iterate over the child nodes (`DOMNode`), and print only the nodes that are instances of `DOMText`, i.e. whose type is `DOMTextNode`.

Running the code of Statement [27.2](#) produces the following output:

```
A: Product A from main factory, rev: 35
B: Product B outsourced, rev: 15
  Warning: max quantity 100
```

27.3 Writing to an XML File

In this section, we will explore the writing capabilities of the `cotxml` module. To do so, we will try to rewrite the same file presented in Section 27.1, only including product-related information.

Rewriting Product Data Statement 27.3 shows the code for rewriting the product data into the XML file. The classes that come into play are the same as for reading. Once again, we first need to create an object `doc` of class `DOMDocument`, the difference with reading being that the empty constructor is used, when creating a new file from scratch. This class is also used as a factory to create elements, attributes, comments, text content. For instance, the method `createElement(string)` of `DOMDocument` is employed to create an element tagged `Delivery` that is then added to the main document.

```
DOMDocument doc();
DOMELEMENT root = doc.createElement("Delivery");
doc.appendChild(root);
```

The same procedure is followed, in order to create the element `Products`, appended to `Delivery`, and the two `Product` elements, appended to `Products`.

```
DOMNode products = root.appendChild(doc.createElement("Products"));
DOMELEMENT productA = doc.createElement("Product");
DOMELEMENT productB = doc.createElement("Product");
products.appendChild(productA);
products.appendChild(productB);
```

```
import cotxml;
DOMDocument doc();
DOMELEMENT root = doc.createElement("Delivery");
doc.appendChild(root);
DOMNode products = root.appendChild(doc.createElement("Products"));
DOMELEMENT productA = doc.createElement("Product");
DOMELEMENT productB = doc.createElement("Product");
products.appendChild(productA);
products.appendChild(productB);
productA.setAttribute("pID", "A");
productA.setAttribute("revenue", 35);
productB.setAttribute("pID", "B");
productB.setAttribute("revenue", 15);
productB.setAttribute("maxQty", 100);
productA.appendChild(doc.createTextNode("Product A from main factory"));
productB.appendChild(doc.createTextNode("Product B outsourced"));
doc.save("example.xml");
```

Statement 27.3: Writing to a Sample XML File: Rewriting Product Data

In order to add attributes to the product elements, it is possible to directly use the method `setAttribute(string,string)` of `DOMElement`; note that the method is overloaded to also allow for setting integer or float attributes.

```
productA.setAttribute("pID","A");
productA.setAttribute("revenue",35);
productB.setAttribute("pID","B");
productB.setAttribute("revenue",15);
productB.setAttribute("maxQty",100);
```

Finally, the text content of the products is inserted with the `appendChild` method, after it is first created using the method `createText(string)` of `DOMDocument`. The XML document is saved to a file by calling the method `save(string)` with the desired filename.

```
productA.appendChild(doc.createTextNode("Product A from main factory"));
productB.appendChild(doc.createTextNode("Product B outsourced"));
doc.save("example.xml");
```

Modifying the XML File We will now describe briefly some useful methods for modifying an XML file. Suppose that we need to add a new product with ID A2 and revenue 30. This product needs to be inserted after product A. The following code describes how to do this. After creating the element, its attributes and its text content, the method `insertAfter(DOMNode,DOMNode)` of class `DOMNode` is called, to insert the newly created element after product A.

```
DOMElement newProduct = doc.createElement("Product");
newProduct.setAttribute("pID","A2");
newProduct.setAttribute("revenue",30);
productA.appendChild(doc.createTextNode("Product A2 from main factory"));
products.insertAfter(newProduct, productA);
```

We can add an XML comment after the new product as follows:

```
DOMComment comment = doc.createComment("Outsourced products");  
products.insertAfter(comment, newProduct);
```

Finally, if we would like to remove a particular node it suffices to call the method `removeChild(DOMNode)`. For instance, we can remove product A as follows:

```
products.removeChild(productA);
```

The final XML file created after executing Statement 27.3 combined with the modifications just described will have the following contents:

```
<?xml version='1.0' encoding='ISO-8859-1'?>  
<!--Created by COMET/XML-->  
<Delivery>  
  <Products>  
    <Product revenue="30" pID="A2" >Product A2 from main factory</Product>  
    <!--Outsourced products-->  
    <Product revenue="15" pID="B" maxQty="100" >Product B outsourced</Product>  
  </Products>  
</Delivery>
```


Chapter 28

C++ and Java Interface

This chapter explains how to interface COMET with C++ and JAVA. The interface acts as a bridge between a COMET code and a C++ or Java code, and allows them to communicate data through input and output. In a typical setting, COMET is used for solving an optimization problem and the C++ code is responsible for providing the data, running the COMET code and reading its output. The Java interface is a wrapper around the C++ interface using Java Native Interface (JNI).

28.1 C++ Interface

Using the COMET system as a library from C++ involves writing two pieces of code: the COMET code to solve the optimization problem and the C++ code to interface with COMET. The interface is simple and only involves the input and output of the COMET program. Only minimal changes need to be made to a COMET program, in order to use it from within a C++ code. The remainder of this section describes the COMET code to be used as a library, the C++ code to run COMET, and, finally, how to compile and run the program.

The Comet Code Statement 28.1 shows a CP COMET code for the Job-Shop Scheduling Problem. Lines related to the C++ interface are indicated using a comment. The code receives input for the machines required and the duration of each task in the form of two two-dimensional arrays of int. After solving, it outputs the makespan. Statements 28.2 and 28.3, which will be described shortly, show the corresponding C++ code, that reads the input file and gets back the makespan from the COMET output.

The key parts in the above code are the `inputInt`, `inputIntArrayArray` and output methods. Currently, the `System` object supports the following input methods:

| | |
|--------------------------|---------------------------------|
| <code>int</code> | <code>inputInt</code> |
| <code>float</code> | <code>inputFloat</code> |
| <code>int []</code> | <code>inputIntArray</code> |
| <code>float []</code> | <code>inputFloatArray</code> |
| <code>set{int}</code> | <code>inputIntSet</code> |
| <code>set{int} []</code> | <code>inputIntSetArray</code> |
| <code>int [] []</code> | <code>inputIntArrayArray</code> |

Each of these methods take a string argument, which is the name given to the input in the C++ code. Note that the type of the input from C++ must match exactly the input method called. That is, if an `int` is read from C++ using a `CometInt` handle and the name “x”, then this must be read in COMET using `inputInt("x")`. This will become more clear after examining the C++ code in Statements 28.2 and 28.3. For each of the above input methods, there is a corresponding output method, whose arguments are the string name of the output handle and the value to output, as shown in the example of Statement 28.1.

```

import cotfd;
int    nbJobs      = System.inputInt("nbJobs");  /* INPUT METHODS */
int    nbTasks     = System.inputInt("nbTasks"); /* INPUT METHODS */
range  Jobs        = 0..nbJobs-1;
range  Tasks       = 0..nbTasks-1;
range  Machines    = Tasks;
int    nbActivities = nbJobs * nbTasks;
range  Activities   = 0..nbActivities+1;
int    duration[Jobs,Tasks];
int    machine[Jobs,Tasks];

int [] [] req = System.inputIntArrayArray("req"); /* INPUT METHODS */
int [] [] dur = System.inputIntArrayArray("dur"); /* INPUT METHODS */
forall (j in Jobs, t in Tasks) {
    int m = req[j][t];
    int d = dur[j][t];
    machine[j,t] = m;
    duration[j,t] = d;
}

int horizon = sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP> cp(horizon);
Activity<CP> a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP> makespan(cp,0);
UnaryResource<CP> r[Machines](cp);
minimize<cp>
    makespan.start()
subject to {
    forall (j in Jobs, t in Tasks : t != Tasks.getUp())
        a[j,t].precedes(a[j,t+1]);
    forall (j in Jobs)
        a[j,Tasks.getUp()].precedes(makespan);
    forall (j in Jobs, t in Tasks)
        a[j,t].requires(r[machine[j,t]]);
}
using {
    forall (m in Machines) by (r[m].localSlack(),r[m].globalSlack())
        r[m].rank();
    makespan.scheduleEarly();
}
System.output("makespan",makespan.start());    /* OUTPUT METHODS */

```

Statement 28.1: CP Model for Job-Shop Scheduling with Enhancements for C++ Interfacing

The C++ Code The C++ code reads job-shop data from the file (in our case the `mt10` instance). It skips the documentation line and then reads the number of jobs and tasks, and enters the machine and duration data into the arrays `req` and `dur`. (Statement 28.2)

After that, the program creates a `CometSystem` and retrieves the `CometOptions` object from the system. (Statement 28.3) Some lines in the C++ code are installation-dependent. The method `addInc` adds a search path for files included in the COMET program with the `include` or `import` statement. The directory containing the license file (`license.col`) should also be added using `addInc`. Similarly, the method `addPlug` adds a search path for plug-in libraries loaded with an `import`, such as the `cotfd` module used in this example.

The C++ program copies the read data into handles for integer and array inputs and then calls `addInput` on the system. Finally, the program solves the given problem and retrieves the output. Note that the `solve` method could throw an exception. This example catches a `CometException`, the ancestor class of all exceptions thrown by the library.

As mentioned above, the types of the handles used for input and output must match those used in the COMET code. For example, the `CometInt` handle is used for `nbJobs` and read in COMET with `inputInt`. Using another type of handle or another input method would result to an error. In an analogous fashion, output handles must match output methods.

Compiling under Mac OS X On Mac OS X the C++ library is packaged as a framework in `/Library/Frameworks/Comet.framework`. Within your C++ code, include the COMET header file with

```
#include <Comet/cometlib.H>
```

Since the COMET framework is in a standard location, no extra compiler flags are necessary. When linking, add the flag `'-framework Comet'`. Assuming that the example above is stored in file `jstest.cpp`, the commands for compilation would then be:

```
g++ -c jstest.cpp
g++ -framework Comet jstest.o -o jstest
```

```
#include "cometlib.H"
#include <iostream>
#include <fstream>
using namespace std;
using namespace Comet;

int main(int argc, char* argv[]) {
    ifstream input("../JOBSHOP/mt10.txt");
    int nbJobs;
    int nbTasks;
    int** req;
    int** dur;
    char buf[512];
    input.getline(buf, 512, '\n');
    input >> nbJobs;
    input >> nbTasks;
    req = new int*[nbJobs];
    for (int i = 0; i < nbJobs; i++)
        req[i] = new int[nbTasks];
    dur = new int*[nbJobs];
    for (int i = 0; i < nbJobs; i++)
        dur[i] = new int[nbTasks];

    for (int i = 0; i < nbJobs; i++)
        for (int j = 0; j < nbTasks; j++) {
            input >> req[i][j];
            input >> dur[i][j];
        }

    [...]
```

Statement 28.2: C++ Code for Interfacing with COMET Code for Job-Shop Scheduling (1/2)

```

[...]
```

```

CometSystem sys;
CometOptions opt = sys.getOptions();
opt.addInc("dir/of/comet_includes");      /* SYSTEM DEPENDENT */
opt.addPlug("dir/of/comet_plugins");      /* SYSTEM DEPENDENT */
opt.setJit(2);
opt.setFilename("ljobshop.co");

CometInt cnbj(nbjJobs);
sys.addInput("nbJobs",cnbj);              /* INPUT METHODS */
CometInt cnbt(nbTasks);
sys.addInput("nbTasks",cnbt);            /* INPUT METHODS */
CometDataArray creq(nbjJobs);
for (int i = 0; i < nbJobs; i++) {
    CometIntArray a(nbTasks, req[i]);
    creq.set(i,a);
}
CometDataArray cdur(nbjJobs);
for (int i = 0; i < nbJobs; i++) {
    CometIntArray a(nbTasks, dur[i]);
    cdur.set(i,a);
}
sys.addInput("req",creq);                /* INPUT METHODS */
sys.addInput("dur",cdur);                /* INPUT METHODS */
try {
    int status = sys.solve();
    CometInt makespan;
    sys.getOutput("makespan",makespan);   /* OUTPUT METHODS */
    cout << "Status=" << status << endl;
    cout << "Makespan=" << makespan << endl;
}
catch (const CometException& e)
    cout << e << endl;
}

```

Statement 28.3: C++ Code for Interfacing with COMET Code for Job-Shop Scheduling (2/2)

Compiling under Linux To use the COMET library on Linux, you need to add the directories containing the COMET header files and library files to the include and library search paths respectively. For example, if COMET is installed in your home directory \$HOME, then these directories are \$HOME/Comet/include and \$HOME/Comet/lib, respectively. It is also necessary to include all the COMET libraries, when linking your program. The following Makefile compiles the C++ code of our example, assuming it is stored into the file `jstest.cpp`. The environment variables `COMETINCLUDE_DIR` and `COMETLIB_DIR` should point to the location of the COMET header files (*.h) and library files (*.so), respectively.

```
# Location of Comet installation:
COMET_DIR=$(HOME)/Comet
COMET_INCLUDE=$(COMET_DIR)/include
COMET_LIB=$(COMET_DIR)/lib

CC=g++
CFLAGS= -I$(COMET_INCLUDE) -c -g
LFLAGS= -L$(COMET_LIB) -lcometlib -Wl,-rpath,$(COMET_LIB)

all : jstest

jstest: jstest.o
    $(CC) $(LFLAGS) $< -o $@

%.o : %.cpp
    $(CC) $(CFLAGS) $<
```

Within your C++ code, include the COMET header file with

```
#include <cometlib.H>
```


Compiling under Windows The Windows installation includes the header files in the include subdirectory of the main COMET installation directory (e.g., C:\Program Files\Dynadec\Comet). The library file `cometlib.lib` is located in the compiler directory, along with the main executable. COMET was compiled with Microsoft Visual Studio 2008. The instructions below assume that a compatible version of Visual Studio is installed. Below are the commands to compile the current example.

```
set COMETDIR="C:\Program Files\Dynadec\Comet"
cl /MT /I%COMETDIR%\include jstest.cpp /Fejstest.exe /EHsc
/link /LIBPATH:%COMETDIR%\compiler cometlib.lib data.lib
kernel32.lib user32.lib gdi32.lib advapi32.lib ODBC32.lib
```

The flag `/MT` indicates that it is linking against the `LIBCMT.lib` runtime library; the flag `/EHsc` enables exception handling; the `.lib` files after `cometlib.lib` are standard Windows libraries. The `include` statement in the C++ code is the same as with Linux. You also need to add the location of the COMET dll files (C:\Program Files\Dynadec\Comet\compiler) to the PATH environment variable.

28.2 Java Interface

The `CometSystem` class provides access to the COMET system from the Java programming language. It is a wrapper around the C++ API using the Java Native Interface (JNI). The main class `CometSystem` is used to specify inputs and outputs. The `CometOptions` class can be used to specify other options, such as, JIT level, deterministic mode, and include paths.

The main difference with the C++ interface is that, instead of using `CometHandle` subclasses for input/output, the Java interface directly uses built-in Java types, including `int`, `double`, `int[]`, `double[]`, `int[][]`, `double[][]`, and `Set<Integer>`. Below is a simple example to illustrate the input/output mechanism:

```
import comet.*;
import java.util.*;
class Test {
    public static void main(String[] args) {
        CometSystem sys = new CometSystem();

        CometOptions o = new CometOptions();
        o.setFilename("test1.co");
        sys.setOptions(o);

        int n = 13;
        sys.addInput("n", n);                                /* INPUT METHODS */
        try {
            sys.solve();
            Set<Integer> s = sys.getIntSetOutput("s");        /* OUTPUT METHODS */
            if (s != null) {
                for (Integer i : s)
                    System.out.print(i+" ");
                System.out.println();
            }
            else
                System.out.println("java: s is null");
        }
        catch (CometException e)
            System.out.println("In Java: caught:"+e);
    }
}
```

The `CometSystem` class loads the JNI libraries and initializes the COMET System. A `CometOptions` object is used to specify the COMET program to run (`test1.co` in this example). The corresponding COMET code uses the input/output from the Java program, to return a set of even numbers between 1 and `n`, as shown below:

```
int n = System.inputInt("n");
set{int} s = filter(i in 1..n ) (i % 2 ==0);
System.output("s",s);
```

Compilation To compile the Java program, you should use the command:

```
javac -cp path/to/jcomet.jar Test.java
```

On Windows, the default location for `jcomet.jar` is `C:\Program Files\Dynadec\Comet\compiler`. On Linux, `jcomet.jar` is located under `Comet/lib` in the location that COMET was installed. On Mac OS, `jcomet.jar` is installed at `/Library/Frameworks/Comet.framework/Resources`. Instead of using the `-cp` flag, you could add `jcomet.jar` to the environment variable `CLASSPATH`.

To run the program, you can use the following command:

```
java -cp path/to/jcomet.jar:. -Djava.library.path=path/to/jnicomet/libraries Test
```

In this case, the `-cp` flag is used, in order to add the file `jcomet.jar` and the current directory, in which `Test.class` is stored, to the class path. The `java.library.path` variable is the directory that contains the COMET JNI library files.

On Windows, the JNI library files are located at `C:\Program Files\Dynadec\Comet\compiler` under the name `jnicomet.dll` and `jnicometloader.dll`. On Linux, the libraries are located at `Comet/lib`, and are named `libjnicomet.so` and `libjnicometloader.so`. Finally, on Mac OS, the libraries are located at `/Library/Frameworks/Comet.framework/Resources/` under the name `jnicomet.dylib` and `jnicometloader.dylib`.

Bibliography

- [1] P. Van Hentenryck, Y. Deville, and C-M. Teng. A generic arc consistency algorithm and its specializations. *Journal of Artificial Intelligence*, 57:291–321, 1992. [263](#)
- [2] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. MIT Press, Cambridge, MA, USA, 2005. [316](#)
- [3] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999. [286](#)

Index

- AC3, [258](#)
- AC5, [263](#), [271](#)
- activity, [275](#)
- Activity<CP>, [279](#)
- all, [32](#), [41](#)
- All-Interval Series Problem (cbls), [344](#)
- alldifferent, [160](#), [174](#), [210](#), [247](#)
- alldifferent (cbls), [318](#), [323](#), [364](#)
- alldifferent (cp), [125](#)
- arccos, [16](#)
- arcsin, [16](#)
- arctan, [16](#)
- argMax, [41](#)
- argMin, [41](#)
- arguments (command line), [8](#)
- aspiration criteria, [387](#), [438](#)
- assert, [64](#)
- atleast, [254](#)
- atleast (cp), [126](#)
- atleast (soft), [138](#)
- atLeastNValue, [249](#)
- atLeastNValue (cp), [137](#)
- atmost (cbls), [367](#), [371](#)
- atmost (cp), [126](#)
- Auto, [119](#)
- balanced academic curriculum problem, [111](#)
- balancing constraints (cp), [140](#)
- BDSController, [204](#), [223](#)
- bijection constraint, [186](#)
- bin-packing, [163](#), [206](#)
- binary constraints (cp), [121](#)
- binary knapsack (cp), [128](#)
- bindValue, [273](#)
- binpackingLB, [163](#)
- bool, [17–18](#)
- Boolean, [17–18](#)
- bound, [201](#)
- breakpoint, [6](#)
- C++ interface, [544](#)
- call, [76](#)
- call(), [397](#)
- cardinality, [111](#)
- cardinality (cp), [127](#)
- cardinality constraint combinator (cbls), [371](#)
 - atmost, [371](#)
 - exactly, [338](#), [371](#)
- casting float to int, [16](#)
- catch, [74](#)
- ceil, [15](#)
- channeling, [216](#)
- channeling (cp)
 - set variables, [150](#)
- channeling constraint, [249](#)
- circuit, [167](#)

- circuit constraints (cp), 130
- class, 63–68
- closure, 76–77
- closures, 396
- collect, 40–41
- color, 480
- column generation, 461
- comet runtime options, 4
- CometVisualizer, 472
- Comparable, 46, 71
- compositionality of constraints, 370
- concurrency, 309
- condition, 85
- consistency level, 119
- Consistency<CP>, 119
- constraint (cbls), 322, 362
 - alldifferent, 364, 366
 - atmost, 367
 - combinatorial, 366
 - knapsack, 368
 - numerical, 366
 - sequence, 369
- constraint (cp), 103
 - alldifferent, 125
 - atleast, 126
 - atLeastNValue, 137
 - atmost, 126
 - binary, 121
 - binaryKnapsack, 128
 - cardinality, 127
 - circuit, 130
 - deviation, 140, 141
 - element, 123
 - exactly, 127
 - in extension, 124
 - indexing, 123
 - minAssignment, 125
 - minCircuit, 131
 - multiknapsack, 129
 - product, 121
 - regular, 135
 - sequence, 132
 - softAtLeast, 138
 - softAtMost, 138
 - softCardinality, 139
 - spread, 140
 - stretch, 133
 - sum, 121
 - table, 124
- constraint system (cbls), 322, 371
- constraint-directed search (cbls), 335–338
- Constraint<LS>, 362
- continuation, 77–78
- cos, 16
- cotdb, 523
- cotln, 456
- cotxml, 529
- counter, 83, 341
- cout, 73
- cumulative job shop problem, 284
- cumulative scheduling, 284
- cutting stock, 461
- database connection, 523
- debugger, 6
- decision variables, 206
- decomposition, 461
- default search, 208
- density function, 56
- deterministic mode, 4
- deviation (cp), 140, 141
- DFSController, 203
- dict, 42–43
- dictionary, 42–43
- discrepancy search, 204
- discrete resource, 284
- diversification, 429
- do while, 48–49
- domain labelling, 214
- domain splitting, 215
- Drawing, 484

- dual, 216
- dual modeling, 186
- dual value (simplex), 465
- dynamic symmetry breaking, 163, 225
- element constraint, 104
- element constraint (cp), 123
- element constraint in a matrix, 249
- element invariant, 356
- enum, 62
- environment variable, 9
- equality, 69–70
- equality constraint, 269
- eternity problem, 186
- events, 79–83, 97
 - restarts (cbbs), 342
 - tabu search (cbbs), 341
 - var<CP>{int}, 178
 - visualization, 489
- exactly, 160, 174
- exceptions, 74
- expression constraints (cp), 121
- extends, 68
- extension constraints, 124
- file reading, 24
- file writing, 25
- filter, 40–41
- find, 21
- finite automaton, 135
- finite domain variables, 102
- first-class expression, 348
- first-fail, 213
- first-fail heuristic, 109
- first-success, 213
- fixpoint, 103, 117
- Float, 15–16
- float, 15–16
- floor, 15
- font, 480
- for, 48–49
- forall, 49–51
- foreveron, 343
- frame, 7
- function, 60–62
- Function<LS>, 346
- getLNSFailureLimit, 238
- getLNSTimeLimit, 238
- getMove(), 397
- getSolution(), 108
- global slack, 277
- greedy search, 219
- greedy to discrepancy search, 223
- hasMove(), 397
- heap, 46, 72
- heap sort, 358
- hello world of CP, 109
- hello world of LP, 456
- heuristic for optimization, 218
- hybridization CP-CBLS, 241
- hybridization LS-CP, 232
- if, 48
- implements, 67
- import, 9
- include, 9
- incremental, 263
- incremental variable (cbbs), 320, 352
- indexing constraint (cp), 123
- inheritance, 68
- int, 12–14
- Integer, 12–14
- intensification, 391
- inter, 39
- interface, 67
 - from C++, 544
 - from Java, 551
- intersection, 39
- IntToString, 13
- invariant, 355

- combinatorial, 356
- count, 357
- distribute, 357
- element, 356
- numerical, 356
- set invariant, 358
- invariant planner (cbls), 447
 - InvariantPlanner<LS>, 447
- Invariant<LS>, 447
- InvariantPlanner<LS>, 447
 - addsource(), 447
 - addTarget(), 447
- isLastLNSRestartCompleted, 238
- iterated tabu search, 433
- Java interface, 551
- job (scheduling), 279
- job shop problem, 277
- knapsack (cbls), 368
- knapsack constraints (cp), 128
- label, 208
 - set variables, 157
- labeled dice problem, 160
- labelFF, 109, 113, 213
 - set variables, 157
- large neighborhood search, 232
- large neighboring search, 284
- lexicographic ordering, 21
- linear programming, 456
- LNS (Scheduling), 287
- lns with initial solution, 234
- lnsOnFailure, 233
- local slack, 277
- local solver, 320
- logical combinators (cbls), 370
 - conjunction, 370
 - disjunction, 370
- logical constraint, 269
- Magic Series Problem (cbls), 338
- Magic Squares Problem (cbls), 326
- makespan, 277
- max-min conflict search, 323, 340
- maximize, 111
- memory allocation, 4
- min-circuit constraint (cp), 130
- minAssignment (cp), 125
- minimize, 111
- monitor, 85
- mouse, 492
- multi-knapsack (cp), 129
- multiknapsack, 111, 163
- mutex, 85
- n-queens, 109
- N-Queens Problem (cbls), 318
- neighbor selector, 396
 - call(), 397
 - getMove(), 397
 - hasMove(), 397
 - AllNeighborSelector, 396
 - call(), 400
 - KMinNeighborSelector, 396
 - MinNeighborSelector, 396
- neighbors, 396
- no-cycle constraint (cp), 130
- non overlap constraints, 169
- non-deterministic search, 105, 197
- notebook page, 474
- numerical constraint (cbls), 366
 - dis-equation, 334
 - equation, 328, 334
- objective function (cbls), 346
 - FloatBoolSumObj<LS>, 428
 - FloatFunctionOverBool<LS>, 423
 - FloatSumMinCostFunctionOverBool<LS>, 426
 - Function<LS>, 346
- ODBC, 523
- onBounds, 119
- onDomains, 119

- onFailure, 163, 169
- onFailure (select), 54
- onRestart, 233
- onValueBind, 178
- onValues, 119
- onValueUnbind, 178
- operator, 69–70
- optimization, 111
- options, 4
- over-constrained, 252
- over-constrained problem, 245

- parall, 85
- path, 9
- permutation, 32
- personnel scheduling, 252
- PERT scheduling, 277
- post, 116, 271
- precedes, 277
- prefix, 19
- pricing problem, 467
- print, 73
- probability, 56
- product constraint (cp), 121
- Progressive Party Problem (cbbs), 382
- propagate, 269

- qtvisual, 471
- quadratic assignment problem, 219
- queens, 210
- queue, 45

- randomized tabu list, 438
- rank, 277
- rank (of a range), 28
- redundant constraints, 169
- redundant variables, 174
- regular (cp), 135
- reified constraint, 269
- relaxation, 245
- relaxPos, 287

- replay mode, 5
- resource, 276
- restart, 186
- restart (cbbs), 341, 393
- restartOnFailureLimit, 230
- restarts, 228
- rostering, 252
- round, 15

- scenes allocation problem, 225
- Scheduler<CP>, 277
- scheduleTimes, 286
- scheduling with CP, 275
- search, 105
- search heuristic, 213
- search tree, 195
- SearchController, 203
- select, 53–54
- selectCircular, 58
- selectFirst, 57
- selectMax, 54–56
- selectMin, 54–56
- selectPr, 56–57
- semaphore, 85
- Send More Money Problem (cbbs), 332
- sequence (cbbs), 369
- sequence constraint (cp), 132
- set, 36–41
- set difference, 39
- set variables (cp), 143
- setLNSFailureLimit, 238
- setLNSTimeLimit, 238
- setTimes, 286
- silent mode, 4
- sin, 16
- slack, 277
- Social Golfers Problem (cbbs), 435
 - Meet User-Defined Invariant Class, 448
 - SocialTournament Class, 435, 441
- soft global constraints, 137
- soft-alldifferent, 137, 249

- soft-balance, 140
- softAtLeast (cp), 138
- softAtMost, 138
- softAtMost (cp), 138
- softCardinality (cp), 139
- solutions, 390
- solve, 195
- solveall, 198
- Solver<LP>, 456
- Solver<MIP>, 465
- sort, 32
- sortPerm, 32
- source variable, 447
- spread (cp), 140
- stack, 44
- startWithRestart, 234
- StateResource, 292
- Steel Mill Slab Problem (cbls), 411
- Steel Mill Slab Problem (cp), 180
- stretch (cp), 133
- string, 19–23
- string comparison, 21
- string equality, 20
- sub-tour elimination, 167
- substring, 19
- suffix, 19
- sum constraint (cp)
 - sum, 121
- switch, 48
- symmetry, 163
- symmetry breaking, 174, 225
- symmetry breaking constraint, 160
- synchronization, 85
- table, 174
- table constraint, 186
- table constraints, 124
- tabu search, 329, 385, 429
 - aspiration criteria, 387, 438
 - dynamic list, 387
 - intensification, 391
 - iterated tabu search, 433
 - randomized tabu list, 438
 - restarts, 393
 - using neighbors, 397
- tabu search (cbls)
 - diversification, 429
- tabu search generic, 330
- tan, 16
- target variable, 447
- text table, 481
- this, 63
- thread, 85
- ThreadPool, 85
- Time Tabling Problem, 509
- Time Tabling Problem (cbls), 373
- time-tabling, 247
- toFloat, 23
- toInt, 23
- toString, 73
- tracking mode, 5
- trail, 269
- trailable set, 263
- trigonometric, 16
- trolley problem, 292
- try, 74, 197
- tryall, 203
- tryPost, 271
- tuple, 63
- unary resource, 277
- UnaryResource<CP>, 279
- union, 38–39
- user-defined constraint (cbls), 404
 - AllDistinct, 404
 - SocialTournament, 438
- user-defined function (cbls), 416, 417
 - SteelObjective, 417
- user-defined invariant (cbls), 447
 - Invariant<LS>, 447
 - Meet, 448
 - propagation, 447

- UserConstraint, [269](#)
- UserConstraint<LS>, [404](#)
- using, [195](#)

- valRemove, [269](#)
- value heuristic, [113](#), [213](#)
- var<CP>{int}, [102](#)
- var<LP>{float}, [456](#)
- var<MIP>{int}, [465](#)
- var{int}, [352](#)
- variable heuristic, [213](#)
- viewpoint, [216](#)
- violations (cbls), [363](#)
 - decomposition-based, [364](#)
 - value-based, [364](#)
 - variable-based, [364](#)
- VisualDrawingBoard, [484](#)
- visualization
 - color, [480](#)
 - font, [480](#)
 - of models, [505](#)
 - qtvisual, [471](#)
 - using events, [489](#)
- Visualizer, [472](#)
- VisualToolButton, [490](#)

- Warehouse Location Problem (cbls), [422](#)
 - WarehouseLocation Class, [423](#)
- weighted constraint (cbls), [372](#)
- when, [82](#)
- whenever, [80](#), [511](#)
- while, [48–49](#)
- with
 - atomic, [433](#)
 - delay, [395](#)

- XML, [529](#)

