OptaPlanner User Guide

Version 6.0.0-redhat-4

by *The OptaPlanner team* [http://www.optaplanner.org/community/team.html]

..... ix
1. Planner introduction 1
1.1. What is OptaPlanner? 1
1.2. What is a planning problem? 3
1.2.1. A planning problem is NP-complete 3
1.2.2. A planning problem has (hard and soft) constraints 3
1.2.3. A planning problem has a huge search space 4
1.3. Download and run the examples 4
1.3.1. Get the release zip and run the examples 4
1.3.2. Run the examples in an IDE (IntelliJ, Eclipse, NetBeans) 6
1.3.3. Use OptaPlanner with Maven, Gradle, Ivy, Buildr or ANT 7
1.3.4. Build OptaPlanner from source 8
1.4. Status of OptaPlanner 8
1.5. Compatibility 9
1.6. Questions, issues and blog 9
2. Quick start 11
2.1. Cloud balancing tutorial 11
2.1.1. Problem statement 11
2.1.2. Problem size 12
2.1.3. Domain model diagram 13
2.1.4. Main method 14
2.1.5. Solver configuration 15
2.1.6. Domain model implementation 17
2.1.7. Score configuration 20
2.1.8. Beyond this tutorial 24
3. Use cases and examples 25
3.1. Examples overview 25
3.2. Basic examples 26
3.2.1. N queens 26
3.2.2. Cloud balancing 30
3.2.3. Traveling salesman (TSP - Traveling salesman problem) 30
3.2.4. Manners 2009 31
3.3. Real examples 31
3.3.1. Course timetabling (ITC 2007 track 3 - Curriculum course scheduling) 31
3.3.2. Machine reassignment (Google ROADEF 2012) 33
3.3.3. Vehicle routing 35
3.3.4. Project job scheduling 40
3.3.5. Hospital bed planning (PAS - Patient admission scheduling) 41
3.4. Difficult examples 43
3.4.1. Exam timetabling (ITC 2007 track 1 - Examination) 43
3.4.2. Employee rostering (INRC 2010 - Nurse rostering) 47
3.4.3. Sport scheduling (TTP - Traveling tournament problem) 52
4. Planner configuration 55
4.1. Overview 55

4.2. Solver configuration ... 56
4.2.1. Solver configuration by XML file ... 56
4.2.2. Solver configuration by Java API ... 57
4.3. Model your planning problem ... 58
4.3.1. Is this class a problem fact or planning entity? ... 58
4.3.2. Problem fact ... 59
4.3.3. Planning entity ... 61
4.3.4. Planning variable ... 63
4.3.5. Planning value and planning value ranges ... 65
4.3.6. Planning problem and planning solution ... 76
4.4. Use the Solver ... 84
4.4.1. The Solver interface ... 84
4.4.2. Solving a problem ... 85
4.4.3. Environment mode: Are there bugs in my code? ... 86
4.4.4. Logging level: What is the Solver doing? ... 88
5. Score calculation ... 91
5.1. Score terminology ... 91
5.1.1. What is a score? ... 91
5.1.2. Score constraint signum (positive or negative) ... 91
5.1.3. Score constraint weight ... 93
5.1.4. Score level ... 93
5.1.5. Pareto scoring (AKA multi-objective optimization scoring) ... 95
5.1.6. Combining score techniques ... 97
5.1.7. The Score interface ... 97
5.1.8. Avoid floating point numbers in score calculation ... 98
5.2. Choose a Score definition ... 100
5.2.1. SimpleScore ... 100
5.2.2. HardSoftScore (recommended) ... 100
5.2.3. HardMediumSoftScore ... 101
5.2.4. BendableScore ... 101
5.2.5. Implementing a custom Score ... 101
5.3. Calculate the Score ... 102
5.3.1. Score calculation types ... 102
5.3.2. Simple Java score calculation ... 102
5.3.3. Incremental Java score calculation ... 104
5.3.4. Drools score calculation ... 107
5.3.5. Detecting invalid scores ... 111
5.4. Score calculation performance tricks ... 112
5.4.1. Overview ... 112
5.4.2. Average calculation count per second ... 112
5.4.3. Incremental score calculation (with delta's) ... 113
5.4.4. Avoid calling remote services during score calculation ... 114
5.4.5. Unused constraint ... 114
5.4.6. Build-in hard constraint ... 114

5.4.7. Other performance tricks ... 114
5.4.8. Score trap ... 115
5.4.9. stepLimit benchmark ... 117
5.5. Reusing the score calculation outside the Solver ... 117
6. Optimization algorithms ... 119
6.1. Search space size in the real world ... 119
6.2. Does Planner find the optimal solution? ... 119
6.3. Architecture overview ... 120
6.4. Optimization algorithms overview ... 121
6.5. Which optimization algorithms should I use? ... 121
6.6. SolverPhase ... 122
6.7. Scope overview ... 124
6.8. Termination ... 125
6.8.1. TimeMillisSpendTermination ... 126
6.8.2. ScoreAttainedTermination ... 127
6.8.3. StepCountTermination ... 127
6.8.4. UnimprovedStepCountTermination ... 127
6.8.5. Combining multiple Terminations ... 128
6.8.6. Asynchronous termination from another thread ... 128
6.9. SolverEventListener ... 129
6.10. Custom SolverPhase ... 129
7. Move and neighborhood selection ... 133
7.1. Move and neighborhood introduction ... 133
7.1.1. What is a Move? ... 133
7.1.2. What is a MoveSelector? ... 134
7.1.3. Subselecting of entities, values and other moves ... 134
7.2. General Selector features ... 136
7.2.1. CacheType: Create moves ahead of time or Just In Time ... 136
7.2.2. SelectionOrder: original, sorted, random, shuffled or probabilistic ... 137
7.2.3. Recommended combinations of CacheType and SelectionOrder ... 139
7.2.4. Filtered selection ... 141
7.2.5. Sorted selection ... 143
7.2.6. Probabilistic selection ... 146
7.2.7. Mimic selection (record/replay) ... 146
7.3. Generic MoveSelectors ... 147
7.3.1. changeMoveSelector ... 147
7.3.2. swapMoveSelector ... 148
7.3.3. pillarSwapMoveSelector ... 150
7.3.4. subChainChangeMoveSelector ... 151
7.3.5. subChainSwapMoveSelector ... 152
7.4. Combining multiple MoveSelectors ... 153
7.4.1. unionMoveSelector ... 153
7.4.2. cartesianProductMoveSelector ... 154
7.5. EntitySelector ... 155

7.6. ValueSelector ... 155
7.7. Custom moves ... 156
7.7.1. Which move types might be missing in my implementation? ... 156
7.7.2. Custom moves introduction ... 156
7.7.3. The interface Move ... 156
7.7.4. MoveListFactory: the easy way to generate custom moves ... 160
7.7.5. MoveIteratorFactory: generate custom moves just in time ... 161
7.7.6. Move generation through DRL ... 162
8. Construction heuristics ... 163
8.1. Overview ... 163
8.2. First Fit ... 163
8.2.1. Algorithm description ... 163
8.2.2. Configuration ... 164
8.3. First Fit Decreasing ... 164
8.3.1. Algorithm description ... 164
8.3.2. Configuration ... 165
8.4. Best Fit ... 166
8.4.1. Algorithm description ... 166
8.4.2. Configuration ... 166
8.5. Best Fit Decreasing ... 166
8.5.1. Algorithm description ... 166
8.5.2. Configuration ... 167
8.6. Cheapest insertion ... 167
8.6.1. Algorithm description ... 167
8.6.2. Configuration ... 167
8.7. Regret insertion ... 167
8.7.1. Algorithm description ... 167
8.7.2. Configuration ... 167
9. Local search ... 169
9.1. Overview ... 169
9.2. Hill Climbing (Simple Local Search) ... 169
9.2.1. Algorithm description ... 169
9.3. Tabu Search ... 171
9.3.1. Algorithm description ... 171
9.4. Simulated Annealing ... 172
9.4.1. Algorithm description ... 172
9.5. Late Acceptance ... 173
9.5.1. Algorithm description ... 173
9.6. Late Simulated Annealing (experimental) ... 174
9.6.1. Algorithm description ... 174
9.7. About neighborhoods, moves and steps ... 175
9.7.1. Move generation tips ... 175
9.7.2. A step ... 175
9.7.3. Getting stuck in local optima ... 178

9.8. Deciding the next step ... 179
9.8.1. Acceptor ... 180
9.8.2. Forager ... 183
9.9. Using a custom Termination, MoveSelector, EntitySelector, ValueSelector or Acceptor ... 185
10. Evolutionary algorithms ... 187
10.1. Overview ... 187
10.2. Evolutionary Strategies ... 187
10.3. Genetic algorithms ... 187
11. Hyperheuristics ... 189
11.1. Overview ... 189
12. Exact methods ... 191
12.1. Overview ... 191
12.2. Brute Force ... 191
12.2.1. Algorithm description ... 191
12.2.2. Configuration ... 193
12.3. Depth-first search ... 193
12.3.1. Algorithm description ... 193
12.3.2. Configuration ... 194
13. Benchmarking and tweaking ... 195
13.1. Finding the best Solver configuration ... 195
13.2. Doing a benchmark ... 196
13.2.1. Adding a dependency on optaplanner-benchmark ... 196
13.2.2. Building and running a PlannerBenchmark ... 196
13.2.3. ProblemIO: input and output of Solution files ... 199
13.2.4. Writing the output solution of the benchmark runs ... 200
13.2.5. Warming up the HotSpot compiler ... 201
13.3. Benchmark report ... 201
13.3.1. HTML report ... 201
13.3.2. Summary statistics ... 201
13.3.3. Statistic per data set (graph and CSV) ... 203
13.3.4. Ranking the Solvers ... 211
13.4. Advanced benchmarking ... 212
13.4.1. Benchmarking performance tricks ... 212
13.4.2. Template based benchmarking and matrix benchmarking ... 213
14. Repeated planning ... 215
14.1. Introduction to repeated planning ... 215
14.2. Backup planning ... 215
14.3. Continuous planning (windowed planning) ... 215
14.3.1. Immovable planning entities ... 216
14.4. Real-time planning (event based planning) ... 217

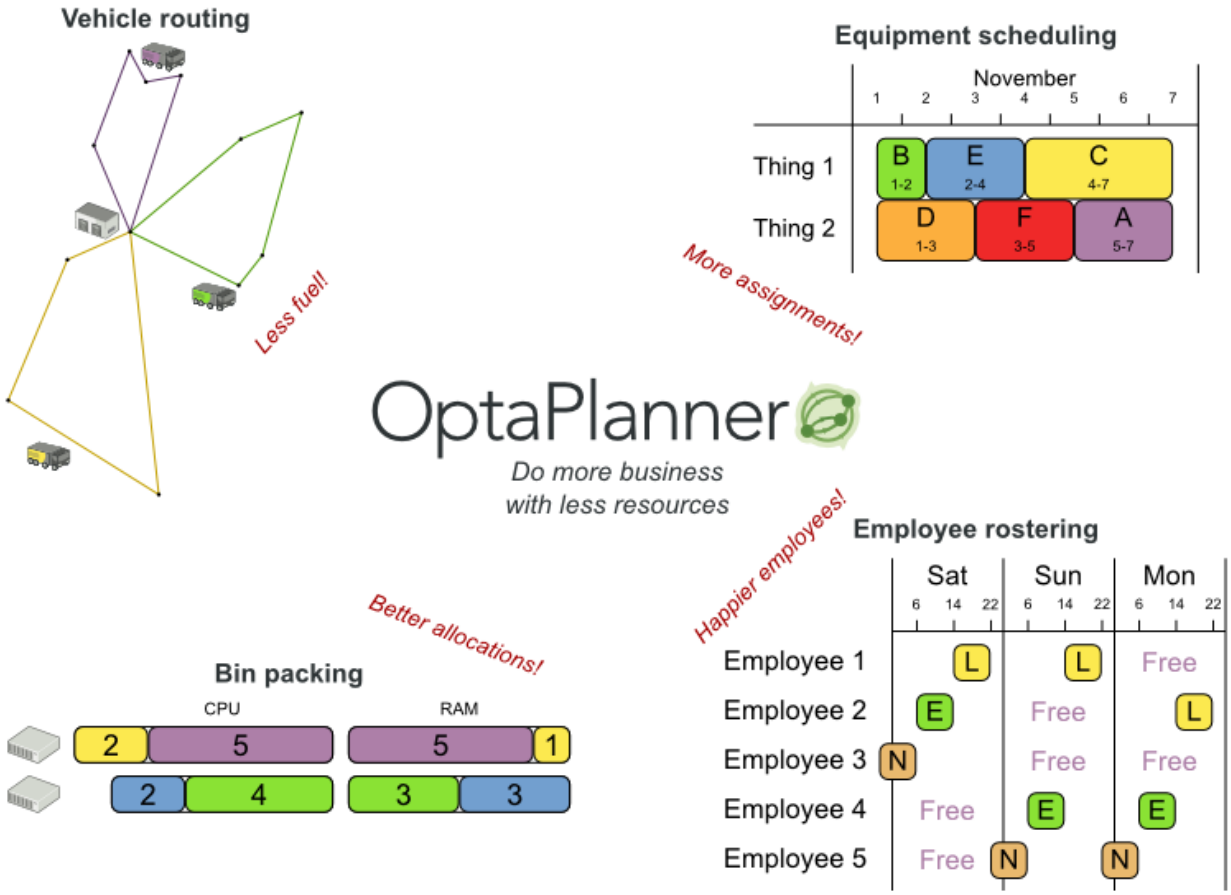
OptaPlanner

Chapter 1. Planner introduction

1.1. What is OptaPlanner?

OptaPlanner **[http://www.optaplanner.org] is a lightweight, embeddable planning engine that optimizes planning problems.** It solves use cases, such as:

- **Employee shift rostering**: timetabling nurses, repairmen, ...
- **Agenda scheduling**: scheduling meetings, appointments, maintenance jobs, advertisements, ...
- **Educational timetabling**: scheduling lessons, courses, exams, conference presentations, ...
- **Vehicle routing**: planning vehicles (trucks, trains, boats, airplanes, ...) with freight and/or people
- **Bin packing**: filling containers, trucks, ships and storage warehouses, but also cloud computers nodes, ...
- **Job shop scheduling**: planning car assembly lines, machine queue planning, workforce task planning, ...
- **Cutting stock**: minimizing waste while cutting paper, steel, carpet, ...
- **Sport scheduling**: planning football leagues, baseball leagues, ...
- **Financial optimization**: investment portfolio optimization, risk spreading, ...



Every organization faces planning problems: provide products or services with a limited set of *constrained* resources (employees, assets, time and money). OptaPlanner optimizes such planning to do more business with less resources. This is known as *Constraint Satisfaction Programming* (which is part of the discipline *Operations Research*).

OptaPlanner helps normal Java™ programmers solve constraint satisfaction problems efficiently. Under the hood, it combines optimization heuristics and metaheuristics with very efficient score calculation.

OptaPlanner is *open source* software, released under *the Apache Software License 2.0* [http://www.apache.org/licenses/LICENSE-2.0.html]. This license is very liberal and allows reuse for commercial purposes. Read *the layman's explanation* [http://www.apache.org/foundation/licence-FAQ.html#WhatDoesItMEAN]. OptaPlanner is 100% pure Java™, runs on *any JVM* [compatibility] and is available in *the Maven Central Repository* too.

1.2. What is a planning problem?

1.2.1. A planning problem is NP-complete

All the use cases above are *probably NP-complete* [http://en.wikipedia.org/wiki/NP-complete]. In layman's terms, this means:

- It's easy to verify a given solution to a problem in reasonable time.
- There is no silver bullet to find the optimal solution of a problem in reasonable time (*).

> **Note**
>
> (*) At least, none of the smartest computer scientists in the world have found such a silver bullet yet. But if they find one for 1 NP-complete problem, it will work for every NP-complete problem.
>
> In fact, there's a $ 1,000,000 reward for anyone that proves if *such a silver bullet actually exists or not* [http://en.wikipedia.org/wiki/P_%3D_NP_problem].

The implication of this is pretty dire: solving your problem is probably harder than you anticipated, because the 2 common techniques won't suffice:

- A brute force algorithm (even a smarter variant) will take too long.
- A quick algorithm, for example in bin packing, *putting in the largest items first*, will return a solution that is usually far from optimal.

By using advanced optimization algorithms, **Planner does find a good solution in reasonable time for such planning problems.**

1.2.2. A planning problem has (hard and soft) constraints

Usually, a planning problem has at least 2 levels of constraints:

- A *(negative) hard constraint* must not be broken. For example: *1 teacher can not teach 2 different lessons at the same time*.
- A *(negative) soft constraint* should not be broken if it can be avoided. For example: *Teacher A does not like to teach on Friday afternoon*.

Some problems have positive constraints too:

- A *positive soft constraint (or reward)* should be fulfilled if possible. For example: *Teacher B likes to teach on Monday morning*.

Some basic problems (such as N Queens) only have hard constraints. Some problems have 3 or more levels of constraints, for example hard, medium and soft constraints.

These constraints define the *score calculation* (AKA *fitness function*) of a planning problem. Each solution of a planning problem can be graded with a score. **With Planner, score constraints are written in an Object Orientated language, such as Java code or Drools rules**. Such code is easy, flexible and scalable.

1.2.3. A planning problem has a huge search space

A planning problem has a number of *solutions*. There are several categories of solutions:

- A *possible solution* is any solution, whether or not it breaks any number of constraints. Planning problems tend to have an incredibly large number of possible solutions. Many of those solutions are worthless.
- A *feasible solution* is a solution that does not break any (negative) hard constraints. The number of feasible solutions tends to be relative to the number of possible solutions. Sometimes there are no feasible solutions. Every feasible solution is a possible solution.
- An *optimal solution* is a solution with the highest score. Planning problems tend to have 1 or a few optimal solutions. There is always at least 1 optimal solution, even in the case that there are no feasible solutions and the optimal solution isn't feasible.
- The *best solution found* is the solution with the highest score found by an implementation in a given amount of time. The best solution found is likely to be feasible and, given enough time, it's an optimal solution.

Counterintuitively, the number of possible solutions is huge (if calculated correctly), even with a small dataset. As you can see in the examples, most instances have a lot more possible solutions than the minimal number of atoms in the known universe (10^80). Because there is no silver bullet to find the optimal solution, any implementation is forced to evaluate at least a subset of all those possible solutions.

OptaPlanner supports several optimization algorithms to efficiently wade through that incredibly large number of possible solutions. Depending on the use case, some optimization algorithms perform better than others, but it's impossible to tell in advance. **With Planner, it is easy to switch the optimization algorithm**, by changing the solver configuration in a few lines of XML or code.

1.3. Download and run the examples

1.3.1. Get the release zip and run the examples

To try it now:

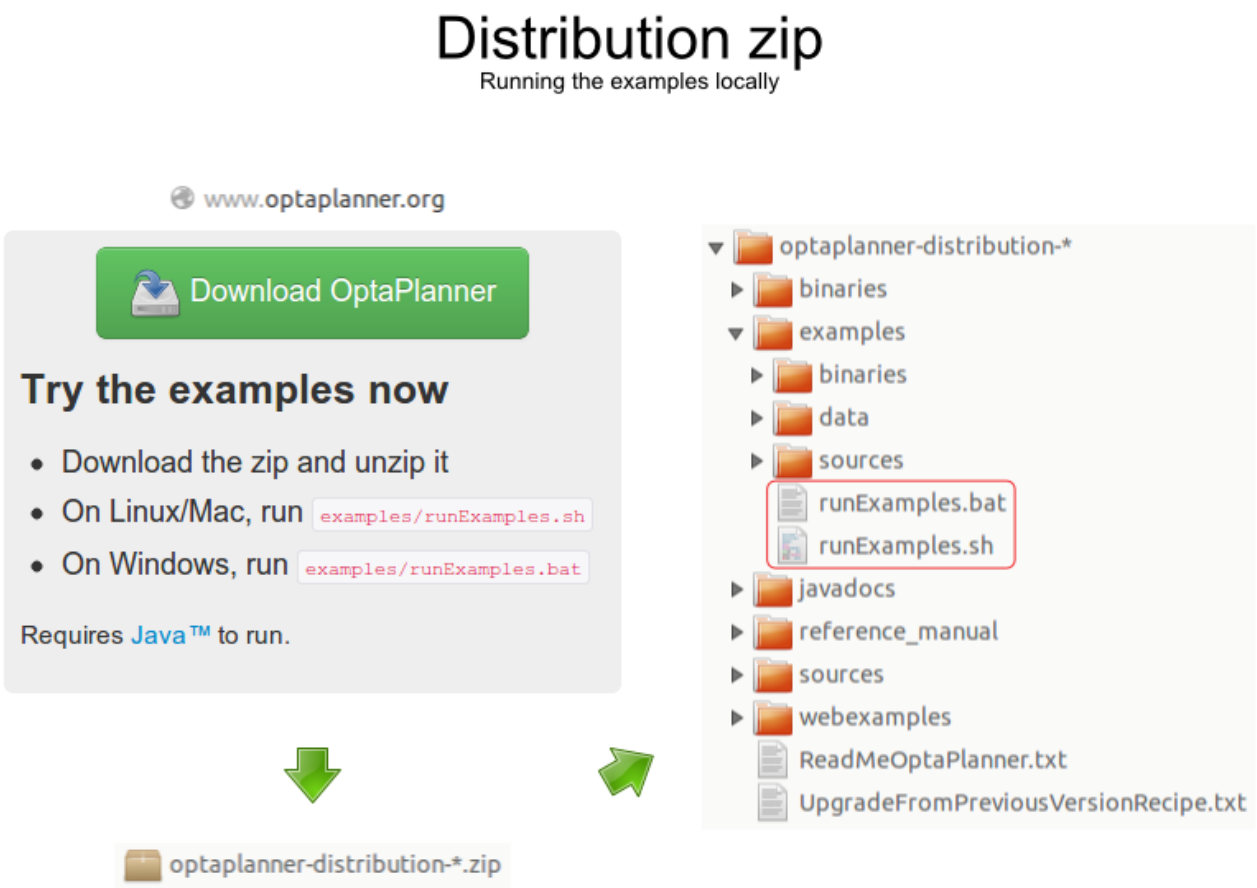
- Download a release zip of OptaPlanner from *the OptaPlanner website* [http://www.optaplanner.org].
- Unzip it.
- Open the directory `examples` and run the script.

  Linux or Mac:

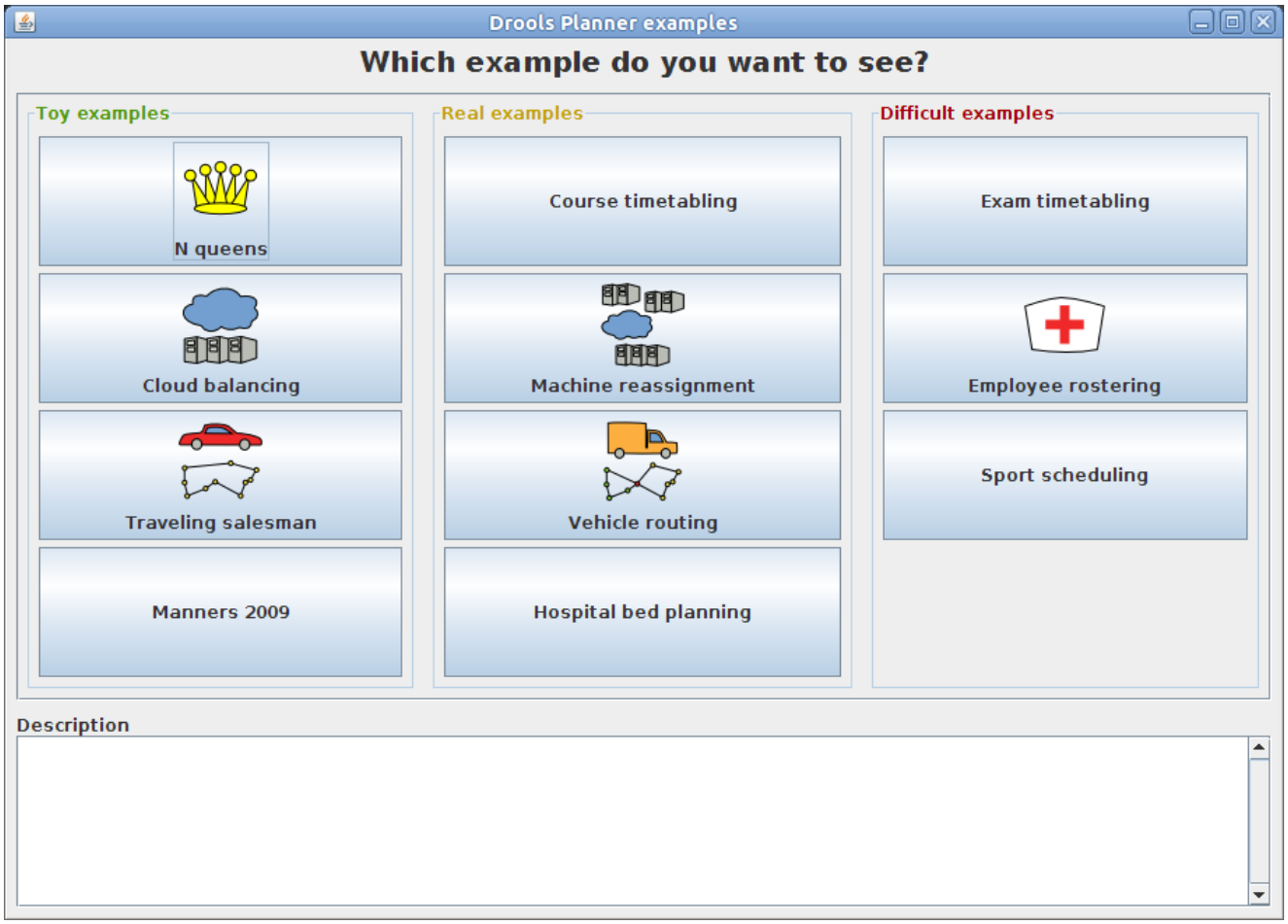
```
$ cd examples
$ ./runExamples.sh
```

  Windows:

```
$ cd examples
$ runExamples.bat
```



The Examples GUI application will open. Just pick an example:



> **Note**
>
> Planner itself has no GUI dependencies. It runs just as well on a server or a mobile JVM as it does on the desktop.

1.3.2. Run the examples in an IDE (IntelliJ, Eclipse, NetBeans)

To run the examples in your favorite IDE:

1. Configure your IDE:

   - In IntelliJ and NetBeans, just open the file `examples/sources/pom.xml` as a new project, the maven integration will take care of the rest.
   - In Eclipse, open a new project for the directory `examples/sources`.
     - Add all the jars to the classpath from the directory `binaries` and the directory `examples/binaries`, except for the file `examples/binaries/optaplanner-examples-*.jar`.

- Add the Java source directory `src/main/java` and the Java resources directory `src/main/resources`.

2. Create a run configuration:

- Main class: `org.optaplanner.examples.app.OptaPlannerExamplesApp`
- VM parameters (optional): `-Xmx512M -server`
- Working directory: `examples` (this is the directory that contains the directory `data`)

3. Run that run configuration.

1.3.3. Use OptaPlanner with Maven, Gradle, Ivy, Buildr or ANT

The OptaPlanner jars are also available in *the central maven repository* [http://search.maven.org/#search|ga|1|org.optaplanner] (and also in *the JBoss maven repository* [https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.optaplanner~~~~]).

If you use Maven, add a dependency to `optaplanner-core` in your project's `pom.xml`:

```
<dependency>
  <groupId>org.optaplanner</groupId>
  <artifactId>optaplanner-core</artifactId>
</dependency>
```

This is similar for Gradle, Ivy and Buildr. To identify the latest version, check *the central maven repository* [http://search.maven.org/#search|ga|1|org.optaplanner].

Because you might end up using other optaplanner modules too, it's recommended to import the `optaplanner-bom` in Maven's `dependencyManagement` so the optaplanner version is specified only once:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.optaplanner</groupId>
      <artifactId>optaplanner-bom</artifactId>
      <type>pom</type>
      <version>...</version>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

If you're still using ANT (without Ivy), copy all the jars from the download zip's `binaries` directory and manually verify that your classpath doesn't contain duplicate jars.

> **Note**
>
> The download zip's `binaries` directory contains far more jars then `optaplanner-core` actually uses. It also contains the jars used by other modules, such as `optaplanner-benchmark`.
>
> Check the maven repository `pom.xml` files to determine the minimal dependency set for a specific version of a specific module.

1.3.4. Build OptaPlanner from source

You can also easily build OptaPlanner from source yourself.

Set up Git [http://help.github.com/set-up-git-redirect] and clone `optaplanner` from GitHub (or alternatively, download *the zipball* [https://github.com/droolsjbpm/optaplanner/zipball/master]):

```
$ git clone git@github.com:droolsjbpm/optaplanner.git optaplanner
...
```

Then do a *Maven 3* [http://maven.apache.org/] build:

```
$ cd optaplanner
$ mvn clean install -DskipTests
...
```

After that, you can run any example directly from the command line, just run this command and pick an example:

```
$ cd optaplanner-examples
$ mvn exec:exec
...
```

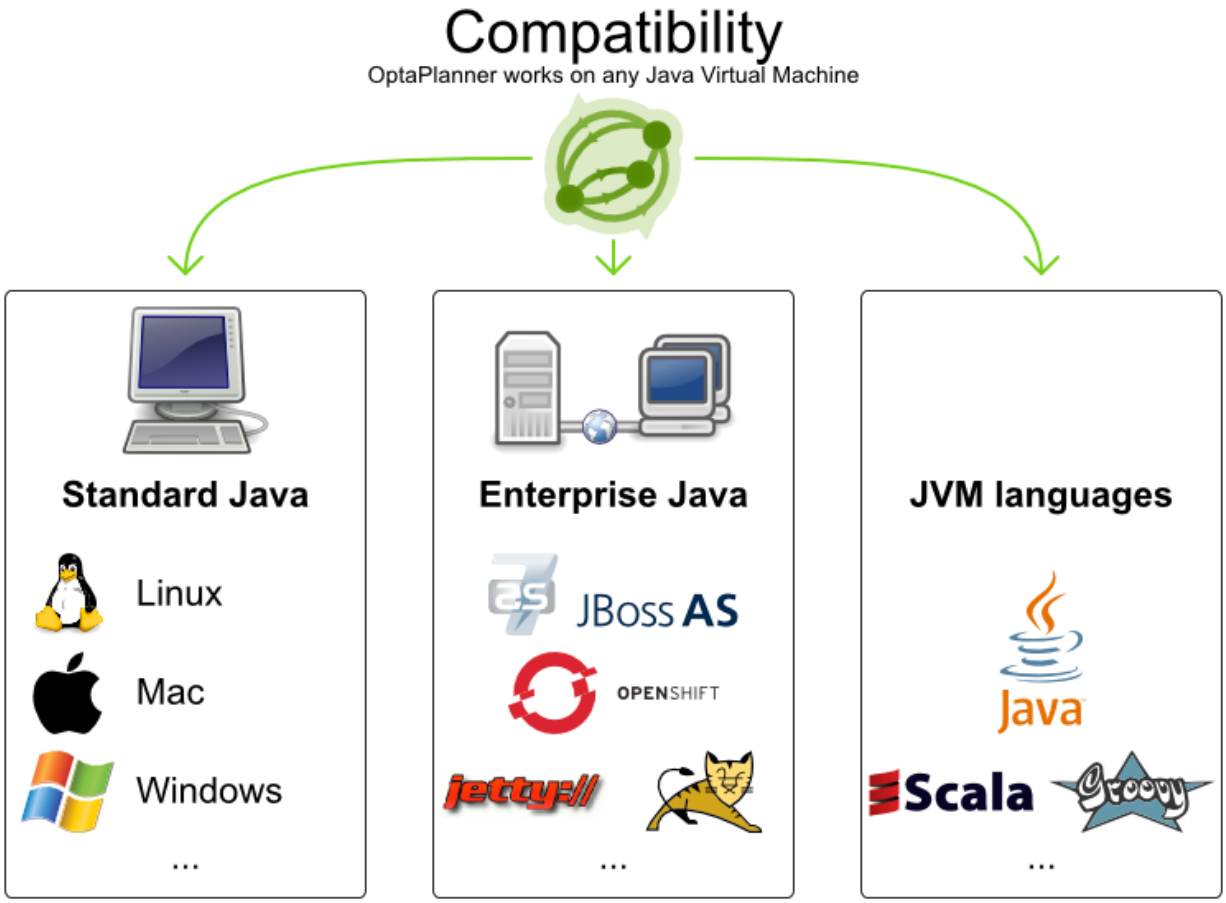
1.4. Status of OptaPlanner

OptaPlanner is production ready. The API is almost stable but backward incompatible changes can occur. With the recipe called *`UpgradeFromPreviousVersionRecipe.txt`* [https://github.com/droolsjbpm/optaplanner/blob/master/optaplanner-distribution/src/main/assembly/filtered-resources/

UpgradeFromPreviousVersionRecipe.txt] you can easily upgrade to a newer version and quickly deal with any backwards incompatible changes. That recipe file is included in every release.

1.5. Compatibility

OptaPlanner is 100% pure Java™ and runs on any JVM 1.6 or higher.



1.6. Questions, issues and blog

Your questions and comments are welcome on *the user mailing list* [http://www.jboss.org/drools/lists.html]. Start the subject of your mail with `[planner]`. You can read/write to the user mailing list without littering your mailbox through *this web forum* [http://drools.46999.n3.nabble.com/Drools-User-forum-f47000.html] or *this newsgroup* [nntp://news.gmane.org/gmane.comp.java.drools.user].

Feel free to report an issue (such as a bug, improvement or a new feature request) for the OptaPlanner code or for this manual to *our issue tracker* [https://issues.jboss.org/browse/PLANNER].

Pull requests are very welcome and get priority treatment! By open sourcing your improvements, you 'll benefit from our peer review and from our improvements made upon your improvements.

Check *our blog* [http://blog.athico.com/search/label/planner], Google+ (*OptaPlanner* [https://plus.google.com/112724449845406009021], *Geoffrey De Smet* [https://plus.google.com/112811208792575157490]) and twitter (*Geoffrey De Smet* [http://twitter.com/geoffreydesmet]) for news and articles. If OptaPlanner helps you solve your problem, don't forget to blog or tweet about it!

Chapter 2. Quick start

2.1. Cloud balancing tutorial

2.1.1. Problem statement

Suppose your company owns a number of cloud computers and needs to run a number of processes on those computers. Assign each process to a computer under the following 4 constraints.

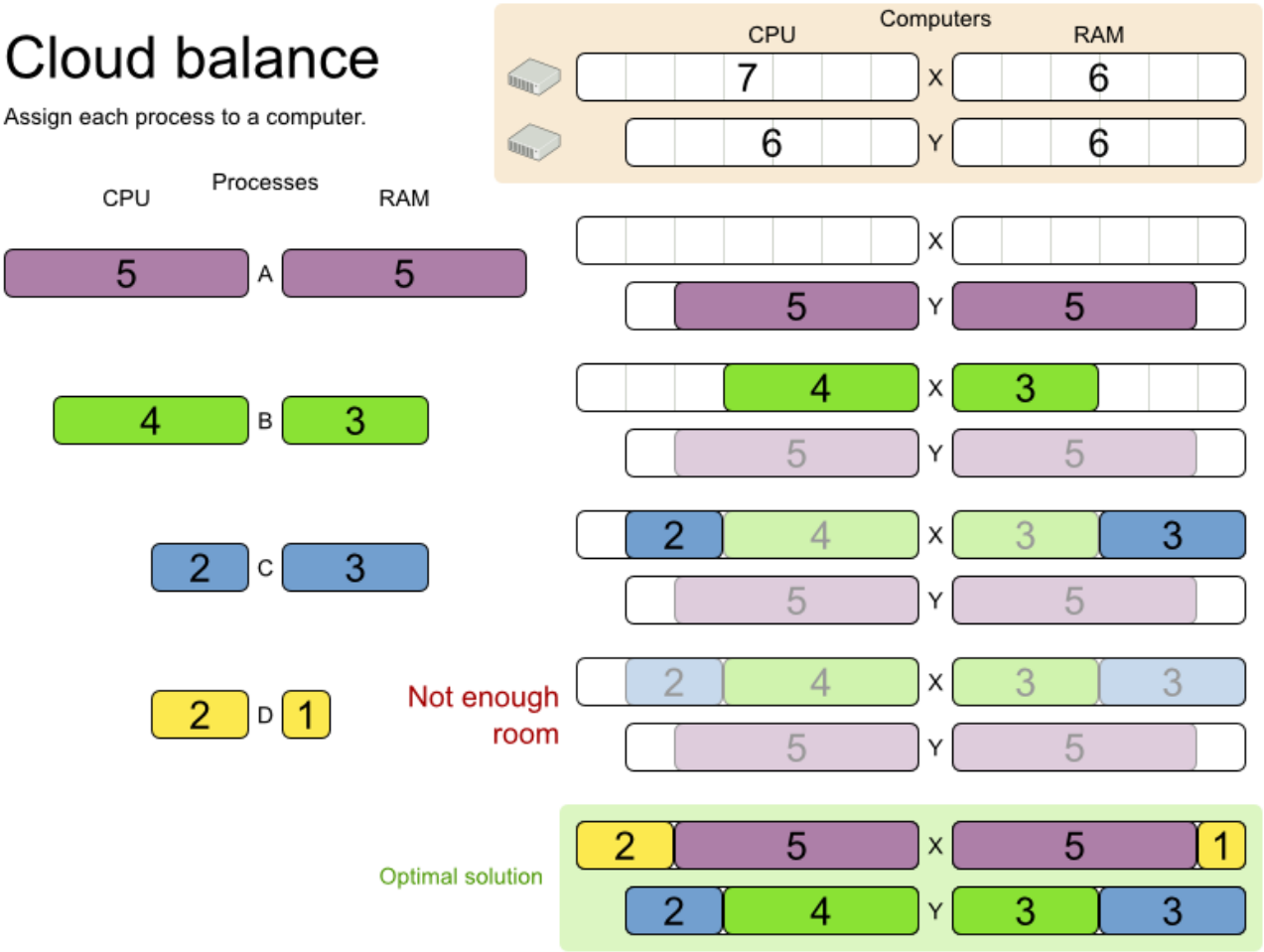
Hard constraints which must be fulfilled:

- Every computer must be able to handle the minimum hardware requirements of the sum of its processes:
  - The CPU power of a computer must be at least the sum of the CPU power required by the processes assigned to that computer.
  - The RAM memory of a computer must be at least the sum of the RAM memory required by the processes assigned to that computer.
  - The network bandwidth of a computer must be at least the sum of the network bandwidth required by the processes assigned to that computer.

Soft constraints which should be optimized:

- Each computer that has one or more processes assigned, incurs a maintenance cost (which is fixed per computer).
  - Minimize the total maintenance cost.

How would you do that? This problem is a form of *bin packing*. Here's a simplified example where we assign 4 processes to 2 computers with 2 constraints (CPU and RAM) with a simple algorithm:



The simple algorithm used here is the *First Fit Decreasing* algorithm, which assigns the bigger processes first and assigns the smaller processes to the remaining space. As you can see, it's not optimal, because it does not leave enough room to assign the yellow process D.

OptaPlanner does find the more optimal solution fast, by using additional, smarter algorithms. And it scales too: both in data (more processes, more computers) and constraints (more hardware requirements, other constraints). So let's take a look how we can use Planner for this.

2.1.2. Problem size

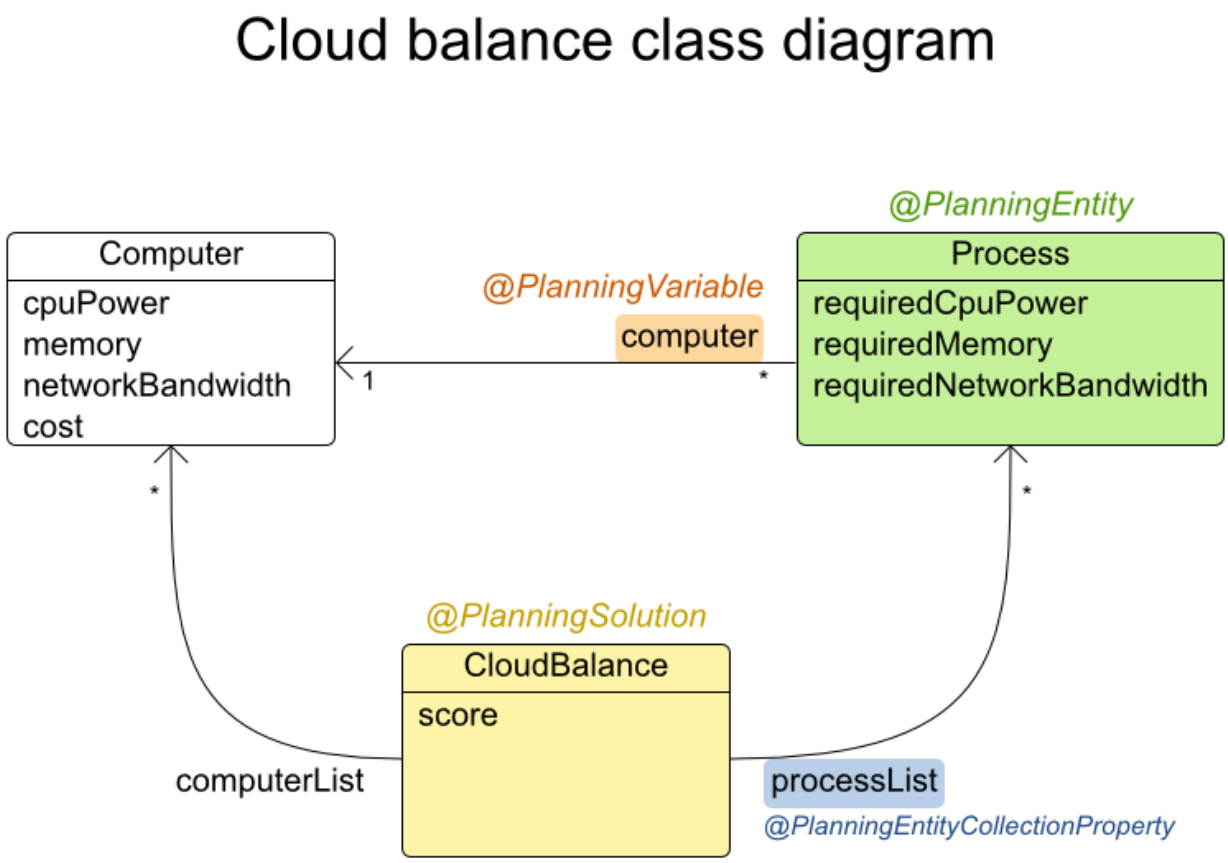
```
2computers-6processes       has   2 computers and    6 processes with a search
 space of      64.
3computers-9processes       has   3 computers and    9 processes with a search
 space of    10^4.
4computers-012processes     has   4 computers and   12 processes with a search
 space of    10^7.
100computers-300processes  has 100 computers and  300 processes with a search
 space of  10^600.
```

```
200computers-600processes  has 200 computers and  600 processes with a search
 space of 10^1380.
400computers-1200processes has 400 computers and 1200 processes with a search
 space of 10^3122.
800computers-2400processes has 800 computers and 2400 processes with a search
 space of 10^6967.
```

2.1.3. Domain model diagram

Let's start by taking a look at the domain model. It's pretty simple:

- `Computer`: represents a computer with certain hardware (CPU power, RAM memory, network bandwidth) and maintenance cost.
- `Process`: represents a process with a demand. Needs to be assigned to a `Computer` by Planner.
- `CloudBalance`: represents a problem. Contains every `Computer` and `Process` for a certain data set.



In the UML class diagram above, the Planner concepts are already annotated:

- Planning entity: the class (or classes) that changes during planning. In this example that's the class `Process`.
- Planning variable: the property (or properties) of a planning entity class that changes during planning. In this examples, that's the property `computer` on the class `Process`.
- Solution: the class that represents a data set and contains all planning entities. In this example that's the class `CloudBalance`.

2.1.4. Main method

Try it yourself. *Download and configure the examples in your favorite IDE.* Run `org.optaplanner.examples.cloudbalancing.app.CloudBalancingHelloWorld`. By default, it is configured to run for 120 seconds. It will execute this code:

Example 2.1. CloudBalancingHelloWorld.java

```
public class CloudBalancingHelloWorld {

    public static void main(String[] args) {
        // Build the Solver
        SolverFactory solverFactory = new XmlSolverFactory(
                                "/org/optaplanner/examples/cloudbalancing/solver/
cloudBalancingSolverConfig.xml");
        Solver solver = solverFactory.buildSolver();

        // Load a problem with 400 computers and 1200 processes
        CloudBalance unsolvedCloudBalance = new CloudBalancingGenerator().createCloudBalance(40

        // Solve the problem
        solver.setPlanningProblem(unsolvedCloudBalance);
        solver.solve();
      CloudBalance solvedCloudBalance = (CloudBalance) solver.getBestSolution();

        // Display the result
         System.out.println("\nSolved cloudBalance with 400 computers and 1200
 processes:\n"
                + toDisplayString(solvedCloudBalance));
    }

    ...

}
```

The code above does this:

- Build the `Solver` based on a solver configuration (in this case an XML file).

```
        SolverFactory solverFactory = new XmlSolverFactory(
                                "/org/optaplanner/examples/cloudbalancing/solver/
  cloudBalancingSolverConfig.xml");
        Solver solver = solverFactory.buildSolver();
```

- Load the problem. `CloudBalancingGenerator` generates a random problem: you'll replace this with a class that loads a real problem, for example from a database.

```
        CloudBalance unsolvedCloudBalance = new CloudBalancingGenerator().createCloudBalance(
```

- Solve the problem.

```
        solver.setPlanningProblem(unsolvedCloudBalance);
        solver.solve();
        CloudBalance solvedCloudBalance = (CloudBalance) solver.getBestSolution();
```

- Display the result.

```
        System.out.println("\nSolved cloudBalance with 400 computers and 1200
  processes:\n"
                + toDisplayString(solvedCloudBalance));
```

The only non-obvious part is building the `Solver`. Let's examine that.

2.1.5. Solver configuration

Take a look at the solver configuration:

Example 2.2. cloudBalancingSolverConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
  <!--<environmentMode>FAST_ASSERT</environmentMode>-->

  <!-- Domain model configuration -->
   <solutionClass>org.optaplanner.examples.cloudbalancing.domain.CloudBalance</
solutionClass>
<planningEntityClass>org.optaplanner.examples.cloudbalancing.domain.CloudProcess</
planningEntityClass>

  <!-- Score configuration -->
```

```
    <scoreDirectorFactory>
      <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

ng.solver.score.CloudBalancingSimpleScoreCalculator</
  simpleScoreCalculatorClass>
                  <!--<scoreDrl>/org/optaplanner/examples/cloudbalancing/solver/
  cloudBalancingScoreRules.drl</scoreDrl>-->
    </scoreDirectorFactory>

    <!-- Optimization algorithms configuration -->
    <termination>
      <maximumSecondsSpend>120</maximumSecondsSpend>
    </termination>
    <constructionHeuristic>
      <constructionHeuristicType>FIRST_FIT_DECREASING</constructionHeuristicType>
      <!--forager-->
        <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType>
      <!--/forager-->
    </constructionHeuristic>
    <localSearch>
      <acceptor>
        <entityTabuSize>7</entityTabuSize>
      </acceptor>
      <forager>
        <acceptedCountLimit>1000</acceptedCountLimit>
      </forager>
    </localSearch>
  </solver>
```

This solver configuration consists out of 3 parts:

- **Domain model configuration**: What can Planner change? We need to make Planner aware of our domain classes:

```
  <solutionClass>org.optaplanner.examples.cloudbalancing.domain.CloudBalance</
  solutionClass>
<planningEntityClass>org.optaplanner.examples.cloudbalancing.domain.CloudProcess</
  planningEntityClass>
```

- **Score configuration**: How should Planner optimize the planning variables? Since we have hard and soft constraints, we use a `HardSoftScore`. But we also need to tell Planner how to calculate such the score, depending on our business requirements. Further down, we 'll look into 2 alternatives to calculate the score: using a simple Java implementation or using Drools DRL.

```
    <scoreDirectorFactory>
      <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
```

```
g.solver.score.CloudBalancingSimpleScoreCalculator</
  simpleScoreCalculatorClass>
                <!--<scoreDrl>/org/optaplanner/examples/cloudbalancing/solver/
  cloudBalancingScoreRules.drl</scoreDrl>-->
    </scoreDirectorFactory>
```

- **Optimization algorithms configuration**: How should Planner optimize it? Don't worry about this for now: this is a good default configuration that works on most planning problems. It will already surpass human planners and most in-house implementations. Using the Planner benchmark toolkit, you can tweak it to get even better results.

```
  <termination>
    <maximumSecondsSpend>120</maximumSecondsSpend>
  </termination>
  <constructionHeuristic>
                            <constructionHeuristicType>FIRST_FIT_DECREASING</
  constructionHeuristicType>
    <!--forager-->
      <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType>
    <!--/forager-->
  </constructionHeuristic>
  <localSearch>
    <acceptor>
      <entityTabuSize>7</entityTabuSize>
    </acceptor>
    <forager>
      <acceptedCountLimit>1000</acceptedCountLimit>
    </forager>
  </localSearch>
```

Let's examine the domain model classes and the score configuration.

2.1.6. Domain model implementation

2.1.6.1. The class `Computer`

The class `Computer` is a POJO (Plain Old Java Object), nothing special. Usually, you'll have more of these kind of classes.

Example 2.3. CloudComputer.java

```
public class CloudComputer ... {

    private int cpuPower;
    private int memory;
```

```
    private int networkBandwidth;
    private int cost;

    ... // getters
}
```

2.1.6.2. The class `Process`

The class `Process` is a little bit special. We need to tell Planner that it can change the field `computer`, so we annotate the class with `@PlanningEntity` and the getter `getComputer` with `@PlanningVariable`:

Example 2.4. CloudProcess.java

```
@PlanningEntity(...)
public class CloudProcess ... {

    private int requiredCpuPower;
    private int requiredMemory;
    private int requiredNetworkBandwidth;

    private CloudComputer computer;

    ... // getters

    @PlanningVariable(valueRangeProviderRefs = {"computerRange"})
    public CloudComputer getComputer() {
        return computer;
    }

    public void setComputer(CloudComputer computer) {
        computer = computer;
    }

    // ************************************************************************
    // Complex methods
    // ************************************************************************

    ...

}
```

The values that Planner can choose from for the field `computer`, are retrieved from a method on the `Solution` implementation: `CloudBalance.getComputerList()` which returns a list of all computers in the current data set. We tell Planner about this by using the `valueRangeProviderRefs` property.

2.1.6.3. The class `CloudBalance`

The class `CloudBalance` implements the `Solution` interface. It holds a list of all computers and processes. We need to tell Planner how to retrieve the collection of process which it can change, so we need to annotate the getter `getProcessList` with `@PlanningEntityCollectionProperty`.

The `CloudBalance` class also has a property `score` which is the `Score` of that `Solution` instance in it's current state:

Example 2.5. CloudBalance.java

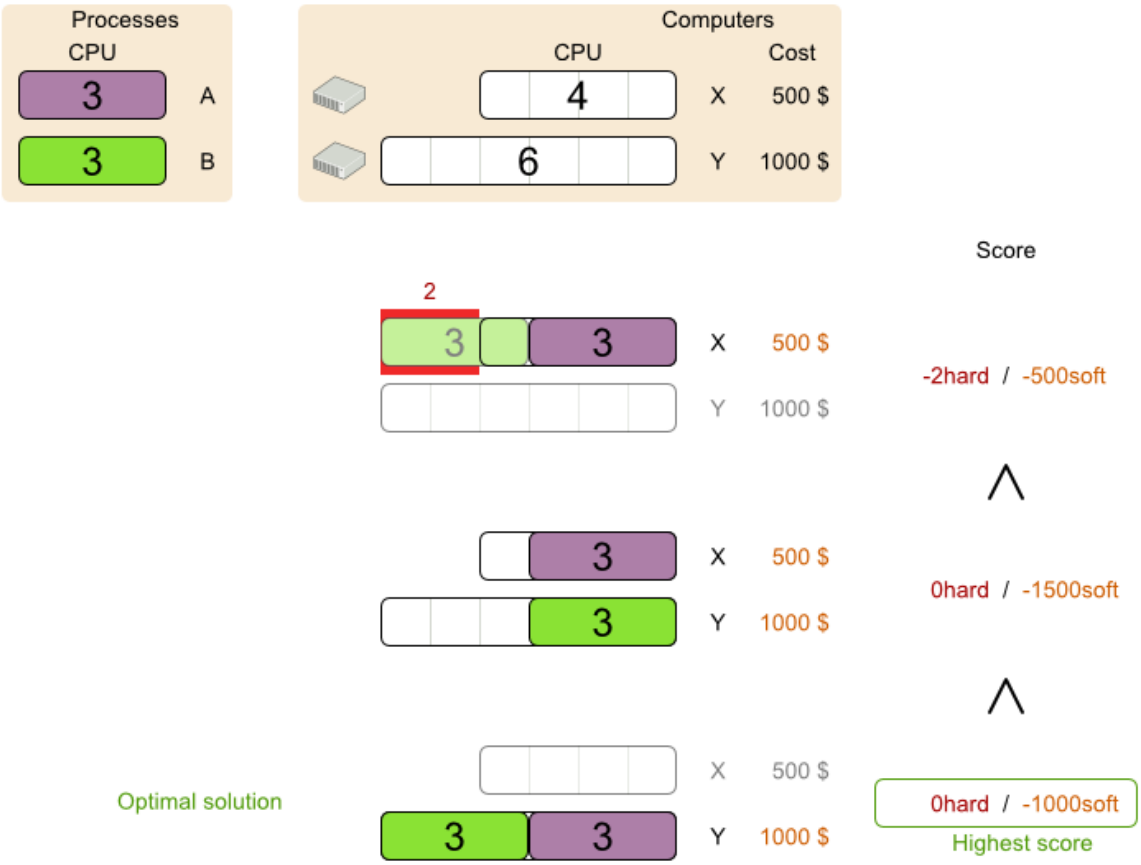
```
public class CloudBalance ... implements Solution<HardSoftScore> {

    private List<CloudComputer> computerList;

    private List<CloudProcess> processList;

    private HardSoftScore score;

    @ValueRangeProvider(id = "computerRange")
    public List<CloudComputer> getComputerList() {
        return computerList;
    }

    @PlanningEntityCollectionProperty
    public List<CloudProcess> getProcessList() {
        return processList;
    }

    ...

    public HardSoftScore getScore() {
        return score;
    }

    public void setScore(HardSoftScore score) {
        this.score = score;
    }

    // ************************************************************************
    // Complex methods
    // ************************************************************************

    public Collection<? extends Object> getProblemFacts() {
        List<Object> facts = new ArrayList<Object>();
        facts.addAll(computerList);
          // Do not add the planning entity's (processList) because that will
 be done automatically
```

```
        return facts;
    }

    ...

}
```

The method `getProblemFacts()` is only needed for score calculation with Drools. It's not needed for the other score calculation types.

2.1.7. Score configuration

Planner will search for the `Solution` with the highest `Score`. We're using a `HardSoftScore`, which means Planner will look for the solution with no hard constraints broken (fulfill hardware requirements) and as little as possible soft constraints broken (minimize maintenance cost).



Of course, Planner needs to be told about these domain-specific score constraints. There are several ways to implement such a score function:

- Simple Java

- Incremental Java
- Drools

Let's take a look at 2 different implementations:

2.1.7.1. Simple Java score configuration

One way to define a score function is to implement the interface `SimpleScoreCalculator` in plain Java.

```
  <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

ng.solver.score.CloudBalancingSimpleScoreCalculator</
simpleScoreCalculatorClass>
  </scoreDirectorFactory>
```

Just implement the method `calculateScore(Solution)` to return a `HardSoftScore` instance.

Example 2.6. CloudBalancingSimpleScoreCalculator.java

```
public class CloudBalancingSimpleScoreCalculator implements SimpleScoreCalculator<CloudBalance>

    /**
     * A very simple implementation. The double loop can easily be removed by using Maps as sho
     * {@link CloudBalancingMapBasedSimpleScoreCalculator#calculateScore(CloudBalance)}.
     */
    public HardSoftScore calculateScore(CloudBalance cloudBalance) {
        int hardScore = 0;
        int softScore = 0;
        for (CloudComputer computer : cloudBalance.getComputerList()) {
            int cpuPowerUsage = 0;
            int memoryUsage = 0;
            int networkBandwidthUsage = 0;
            boolean used = false;

            // Calculate usage
            for (CloudProcess process : cloudBalance.getProcessList()) {
                if (computer.equals(process.getComputer())) {
                    cpuPowerUsage += process.getRequiredCpuPower();
                    memoryUsage += process.getRequiredMemory();
                networkBandwidthUsage += process.getRequiredNetworkBandwidth();
                    used = true;
                }
            }
```

```
            // Hard constraints
            int cpuPowerAvailable = computer.getCpuPower() - cpuPowerUsage;
            if (cpuPowerAvailable < 0) {
                hardScore += cpuPowerAvailable;
            }
            int memoryAvailable = computer.getMemory() - memoryUsage;
            if (memoryAvailable < 0) {
                hardScore += memoryAvailable;
            }
            int networkBandwidthAvailable = computer.getNetworkBandwidth() - networkBandwidthUs
            if (networkBandwidthAvailable < 0) {
                hardScore += networkBandwidthAvailable;
            }

            // Soft constraints
            if (used) {
                softScore -= computer.getCost();
            }
        }
        return HardSoftScore.valueOf(hardScore, softScore);
    }

}
```

Even if we optimize the code above to use `Maps` to iterate through the `processList` only once, **it is still slow** because it doesn't do incremental score calculation. To fix that, either use an incremental Java score function or a Drools score function. Let's take a look at the latter.

2.1.7.2. Drools score configuration

To use the Drools rule engine as a score function, simply add a `scoreDrl` resource in the classpath:

```
  <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
                      <scoreDrl>/org/optaplanner/examples/cloudbalancing/solver/
cloudBalancingScoreRules.drl</scoreDrl>
  </scoreDirectorFactory>
```

First, we want to make sure that all computers have enough CPU, RAM and network bandwidth to support all their processes, so we make these hard constraints:

Example 2.7. cloudBalancingScoreRules.drl - hard constraints

```
...
```

```
import org.optaplanner.examples.cloudbalancing.domain.CloudBalance;
import org.optaplanner.examples.cloudbalancing.domain.CloudComputer;
import org.optaplanner.examples.cloudbalancing.domain.CloudProcess;

global HardSoftScoreHolder scoreHolder;

// ############################################################################
// Hard constraints
// ############################################################################

rule "requiredCpuPowerTotal"
    when
        $computer : CloudComputer($cpuPower : cpuPower)
        $requiredCpuPowerTotal : Number(intValue > $cpuPower) from accumulate(
            CloudProcess(
                computer == $computer,
                $requiredCpuPower : requiredCpuPower),
            sum($requiredCpuPower)
        )
    then
                    scoreHolder.addHardConstraintMatch(kcontext,   $cpuPower   -
 $requiredCpuPowerTotal.intValue());
end

rule "requiredMemoryTotal"
    ...
end

rule "requiredNetworkBandwidthTotal"
    ...
end
```

Next, if those constraints are met, we want to minimize the maintenance cost, so we add that as a soft constraint:

Example 2.8. cloudBalancingScoreRules.drl - soft constraints

```
// ############################################################################
// Soft constraints
// ############################################################################

rule "computerCost"
    when
        $computer : CloudComputer($cost : cost)
        exists CloudProcess(computer == $computer)
    then
```

```
        scoreHolder.addSoftConstraintMatch(kcontext, - $cost);
end
```

If you use the Drools rule engine for score calculation, you can integrate with other Drools technologies, such as decision tables (XLS or web based), the Guvnor rule repository, ...

2.1.8. Beyond this tutorial

Now that this simple example works, try going further. Enrich the domain model and add extra constraints such as these:

- Each `Process` belongs to a `Service`. A computer can crash, so processes running the same service should be assigned to different computers.
- Each `Computer` is located in a `Building`. A building can burn down, so processes of the same services should be assigned to computers in different buildings.

Chapter 3. Use cases and examples

3.1. Examples overview

OptaPlanner has several examples. In this manual we explain OptaPlanner mainly using the n queens example. So it's advisable to read at least the section about that example.

The source code of all these examples is available in the distribution zip under `examples/sources` and also in git under `optaplanner/optaplanner-examples`.

Table 3.1. Examples overview

Example	Domain	Entity size	Value size	Search space	Competition?
N queens	• 1 entity class • 1 variable	`<= 256`	`<= 256`	`<= 10^616`	Pointless
Cloud balancing	• 1 entity class • 1 variable	`<= 2400`	`<= 800`	`<= 10^6967`	No
Traveling salesman	• 1 entity class • 1 chained variable	`<= 980`	`<= 980`	`<= 10^2927`	Unrealistic
Manners 2009	• 1 entity class • 1 variable	`<= 144`	`<= 144`	`<= 10^310`	Unrealistic
Course timetabling	• 1 entity class • 2 variables	`<= 434`	`<= 25` and `<= 20`	`<= 10^1171`	Realistic
Machine reassignment	• 1 entity class • 1 variable	`<= 50000`	`<= 5000`	`<= 10^184948`	Nearly realistic
Vehicle routing	• 1 entity class • 1 chained variable • 1 shadow entity class • 1 automatic shadow variable	`<= 134`	`<= 141`	`<= 10^285`	Unrealistic
Vehicle routing with time windows	Extra on Vehicle Routing:	`<= 250`	`<= 1250`	`<= 10^3000`	Unrealistic

Example	Domain	Entity size	Value size	Search space	Competition?
	• 1 shadow variable				
Project job scheduling	• 1 entity class • 2 variables • 1 shadow variable	<= `640`	<= ? and <= ?	<= ?	Nearly realistic
Hospital bed planning	• 1 entity class • 1 variable	<= `2750`	<= `471`	<= `10^6851`	Realistic
Exam timetabling	• 1 entity class • 2 variables	<= `1096`	<= 80 and <= `49`	<= `10^3374`	Realistic
Employee rostering	• 1 entity class • 1 variable	<= `752`	<= `50`	<= `10^1277`	Realistic
Sport scheduling	• 1 entity class • 1 variable	<= `1560`	<= `78`	<= `10^2951`	Unrealistic

A *realistic competition* is **an official, independent competition**:

- that clearly defines a real-word use case
- with real-world constraints
- with multiple, real-world datasets
- that expects reproducible results within a specific time limit on specific hardware
- that has had serious participation from the academic and/or enterprise Operations Research community

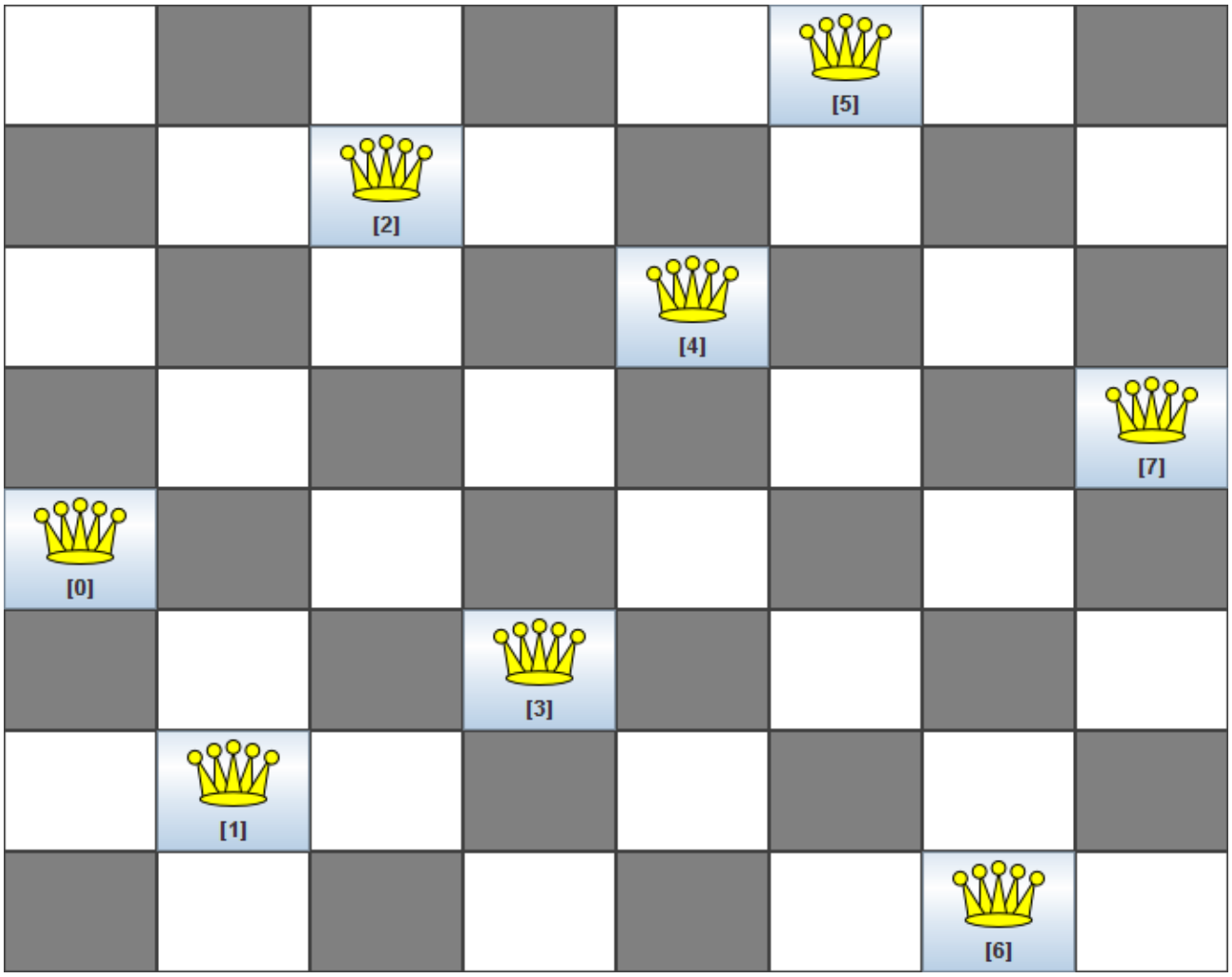
These realistic competitions provide an objective comparison of OptaPlanner with competitive software and academic research.

3.2. Basic examples

3.2.1. N queens

3.2.1.1. Problem statement

Place n queens on a n sized chessboard so no 2 queens can attach each other. The most common n queens puzzle is the 8 queens puzzle, with $n = 8$:



Constraints:

- Use a chessboard of *n* columns and *n* rows.
- Place *n* queens on the chessboard.
- No 2 queens can attack each other. A queen can attack any other queen on the same horizontal, vertical or diagonal line.

This documentation heavily uses the 4 queens puzzle as the primary example.

A proposed solution could be:

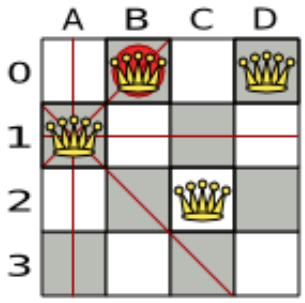


Figure 3.1. A wrong solution for the 4 queens puzzle

The above solution is wrong because queens `A1` and `B0` can attack each other (so can queens `B0` and `D0`). Removing queen `B0` would respect the "no 2 queens can attack each other" constraint, but would break the "place n queens" constraint.

Below is a correct solution:

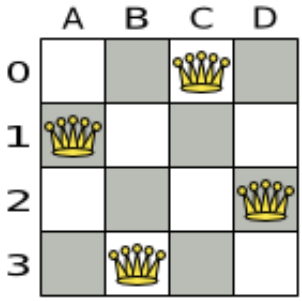


Figure 3.2. A correct solution for the 4 queens puzzle

All the constraints have been met, so the solution is correct. Note that most n queens puzzles have multiple correct solutions. We'll focus on finding a single correct solution for a given n, not on finding the number of possible correct solutions for a given n.

3.2.1.2. Problem size

```
4queens   has   4 queens with a search space of    256.
8queens   has   8 queens with a search space of   10^7.
16queens  has  16 queens with a search space of  10^19.
32queens  has  32 queens with a search space of  10^48.
64queens  has  64 queens with a search space of 10^115.
256queens has 256 queens with a search space of 10^616.
```

The implementation of the N queens example has not been optimized because it functions as a beginner example. Nevertheless, it can easily handle 64 queens. With a few changes it has been shown to easily handle 5000 queens and more.

3.2.1.3. Domain model

Use a good domain model: it will be easier to understand and solve your planning problem. This is the domain model for the n queens example:

```
public class Column {

    private int index;

    // ... getters and setters
}
```

```
public class Row {

    private int index;

    // ... getters and setters
}
```

```
public class Queen {

    private Column column;
    private Row row;

    public int getAscendingDiagonalIndex() {...}
    public int getDescendingDiagonalIndex() {...}

    // ... getters and setters
}
```

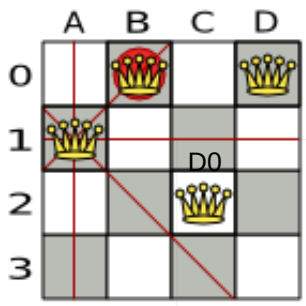
A `Queen` instance has a `Column` (for example: 0 is column A, 1 is column B, ...) and a `Row` (its row, for example: 0 is row 0, 1 is row 1, ...). Based on the column and the row, the ascending diagonal line as well as the descending diagonal line can be calculated. The column and row indexes start from the upper left corner of the chessboard.

```
public class NQueens implements Solution<SimpleScore> {

    private int n;
    private List<Column> columnList;
    private List<Row> rowList;

    private List<Queen> queenList;
```

```
    private SimpleScore score;

    // ... getters and setters
}
```

A single `NQueens` instance contains a list of all `Queen` instances. It is the `Solution` implementation which will be supplied to, solved by and retrieved from the Solver. Notice that in the 4 queens example, NQueens's `getN()` method will always return 4.

Table 3.2. A solution for 4 queens shown in the domain model

A solution	Queen	columnIndex	rowIndex	ascendingDiag (columnIndex + rowIndex)	descendingDi (columnIndex - rowIndex)
		0	1	**1 (**)**	-1
		1	**0 (*)**	**1 (**)**	1
		2	2	4	0
		3	**0 (*)**	3	3

When 2 queens share the same column, row or diagonal line, such as (*) and (**), they can attack each other.

3.2.2. Cloud balancing

This example is explained in *a tutorial*.

3.2.3. Traveling salesman (TSP - Traveling salesman problem)

3.2.3.1. Problem statement

Given a list of cities, find the shortest tour for a salesman that visits each city exactly once.

The problem is defined by *Wikipedia* [http://en.wikipedia.org/wiki/Travelling_salesman_problem]. It is *one of the most intensively studied problems* [http://www.tsp.gatech.edu/] in computational mathematics. Yet, in the real world, it's often only part of a planning problem, along with other constraints, such as employee shift rostering constraints.

3.2.3.2. Problem size

```
dj38     has  38 cities with a search space of   10^58.
```

```
europe40 has  40 cities with a search space of   10^62.
st70     has  70 cities with a search space of  10^126.
pcb442   has 442 cities with a search space of 10^1166.
lu980    has 980 cities with a search space of 10^2927.
```

3.2.4. Manners 2009

3.2.4.1. Problem statement

In Manners 2009, miss Manners is throwing a party again.

- This time she invited 144 guests and prepared 12 round tables with 12 seats each.
- Every guest should sit next to someone (left and right) of the opposite gender.
- And that neighbour should have at least one hobby in common with the guest.
- Also, this time there should be 2 politicians, 2 doctors, 2 socialites, 2 sports stars, 2 teachers and 2 programmers at each table.
- And the 2 politicians, 2 doctors, 2 sports stars and 2 programmers shouldn't be the same kind.

Drools Expert also has the normal miss Manners examples (which is much smaller) and employs a brute force heuristic to solve it. OptaPlanner's implementation is far more scalable because it uses heuristics to find the best solution and Drools Expert to calculate the score of each solution.

3.2.4.2. Problem size

```
geoffrey1             has 18 jobs, 144 guests, 288 hobby practicians, 12 tables
 and 144 seats with a search space of 10^310.
james1WithGenderChange has 18 jobs, 144 guests, 432 hobby practicians, 12 tables
 and 144 seats with a search space of 10^310.
```

3.3. Real examples

3.3.1. Course timetabling (ITC 2007 track 3 - Curriculum course scheduling)

3.3.1.1. Problem statement

Schedule each lecture into a timeslot and into a room.

Hard constraints:

- Teacher conflict: A teacher must not have 2 lectures in the same period.

- Curriculum conflict: A curriculum must not have 2 lectures in the same period.
- Room occupancy: 2 lectures must not be in the same room in the same period.
- Unavailable period (specified per dataset): A specific lecture must not be assigned to a specific period.

Soft constraints:

- Room capacity: A room's capacity should not be less than the number of students in its lecture.
- Minimum working days: Lectures of the same course should be spread into a minimum number of days.
- Curriculum compactness: Lectures belonging to the same curriculum should be adjacent to each other (so in consecutive periods).
- Room stability: Lectures of the same course should be assigned the same room.

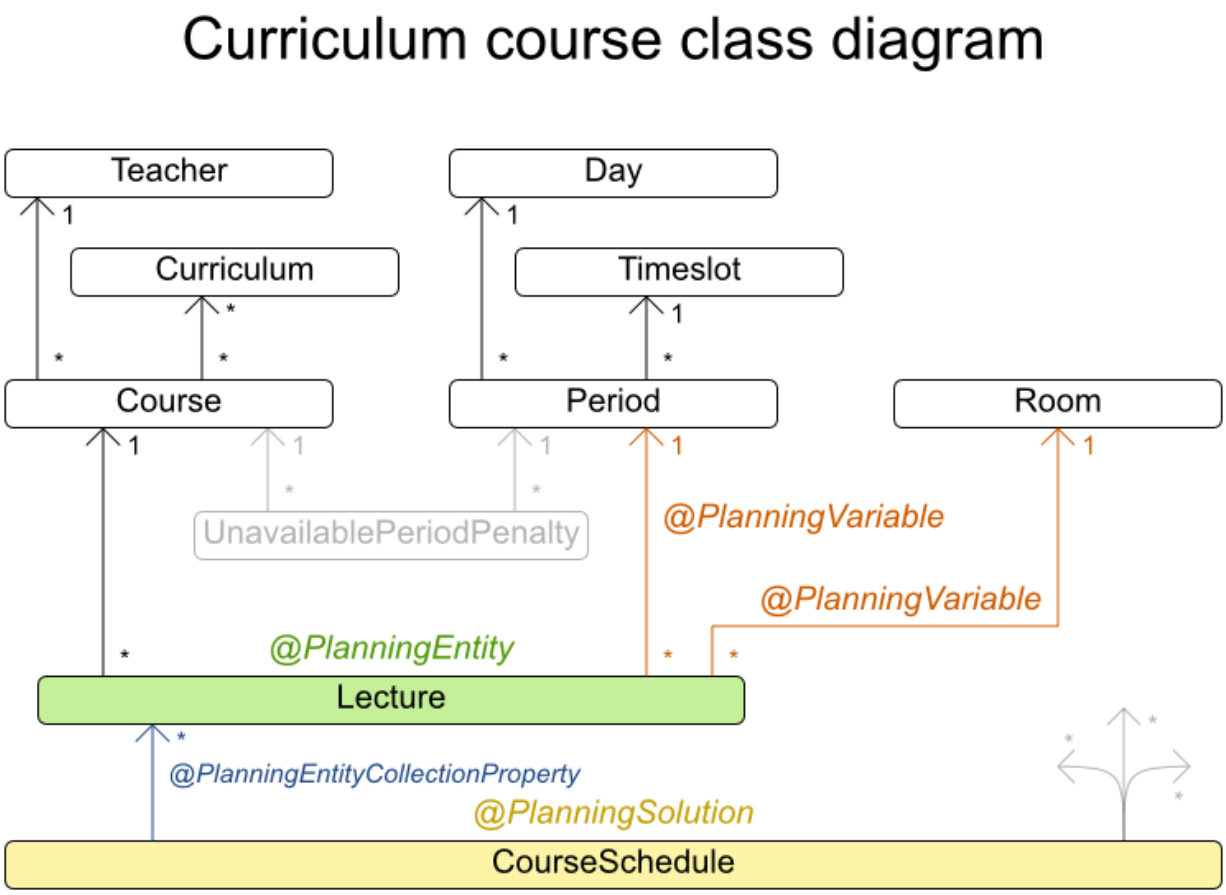
The problem is defined by *the International Timetabling Competition 2007 track 3* [http://www.cs.qub.ac.uk/itc2007/curriculmcourse/course_curriculm_index.htm].

3.3.1.2. Problem size

```
comp01 has 24 teachers,  14 curricula,  30 courses, 160 lectures, 30 periods,  6
 rooms and   53 unavailable period constraints with a search space of  10^360.
comp02 has 71 teachers,  70 curricula,  82 courses, 283 lectures, 25 periods, 16
 rooms and  513 unavailable period constraints with a search space of  10^736.
comp03 has 61 teachers,  68 curricula,  72 courses, 251 lectures, 25 periods, 16
 rooms and  382 unavailable period constraints with a search space of  10^653.
comp04 has 70 teachers,  57 curricula,  79 courses, 286 lectures, 25 periods, 18
 rooms and  396 unavailable period constraints with a search space of  10^758.
comp05 has 47 teachers, 139 curricula,  54 courses, 152 lectures, 36 periods,  9
 rooms and  771 unavailable period constraints with a search space of  10^381.
comp06 has 87 teachers,  70 curricula, 108 courses, 361 lectures, 25 periods, 18
 rooms and  632 unavailable period constraints with a search space of  10^957.
comp07 has 99 teachers,  77 curricula, 131 courses, 434 lectures, 25 periods, 20
 rooms and  667 unavailable period constraints with a search space of 10^1171.
comp08 has 76 teachers,  61 curricula,  86 courses, 324 lectures, 25 periods, 18
 rooms and  478 unavailable period constraints with a search space of  10^859.
comp09 has 68 teachers,  75 curricula,  76 courses, 279 lectures, 25 periods, 18
 rooms and  405 unavailable period constraints with a search space of  10^740.
comp10 has 88 teachers,  67 curricula, 115 courses, 370 lectures, 25 periods, 18
 rooms and  694 unavailable period constraints with a search space of  10^981.
comp11 has 24 teachers,  13 curricula,  30 courses, 162 lectures, 45 periods,  5
 rooms and   94 unavailable period constraints with a search space of  10^381.
comp12 has 74 teachers, 150 curricula,  88 courses, 218 lectures, 36 periods, 11
 rooms and 1368 unavailable period constraints with a search space of  10^566.
```

```
comp13 has 77 teachers,  66 curricula,  82 courses, 308 lectures, 25 periods, 19
 rooms and  468 unavailable period constraints with a search space of  10^824.
comp14 has 68 teachers,  60 curricula,  85 courses, 275 lectures, 25 periods, 17
 rooms and  486 unavailable period constraints with a search space of  10^722.
```

3.3.1.3. Domain model



3.3.2. Machine reassignment (Google ROADEF 2012)

3.3.2.1. Problem statement

Assign each process to a machine. All processes already have an original (unoptimized) assignment. Each process requires an amount of each resource (such as CPU, RAM, ...). This is more complex version of the Cloud Balancing example.

Hard constraints:

- Maximum capacity: The maximum capacity for each resource for each machine must not be exceeded.

- Conflict: Processes of the same service must run on distinct machines.
- Spread: Processes of the same service must be spread across locations.
- Dependency: The processes of a service depending on another service must run in the neighborhood of a process of the other service.
- Transient usage: Some resources are transient and count towards the maximum capacity of both the original machine as the newly assigned machine.

Soft constraints:

- Load: The safety capacity for each resource for each machine should not be exceeded.
- Balance: Leave room for future assignments by balancing the available resources on each machine.
- Process move cost: A process has a move cost.
- Service move cost: A service has a move cost.
- Machine move cost: Moving a process from machine A to machine B has another A-B specific move cost.

The problem is defined by *the Google ROADEF/EURO Challenge 2012* [http://challenge.roadef.org/2012/en/].

3.3.2.2. Problem size

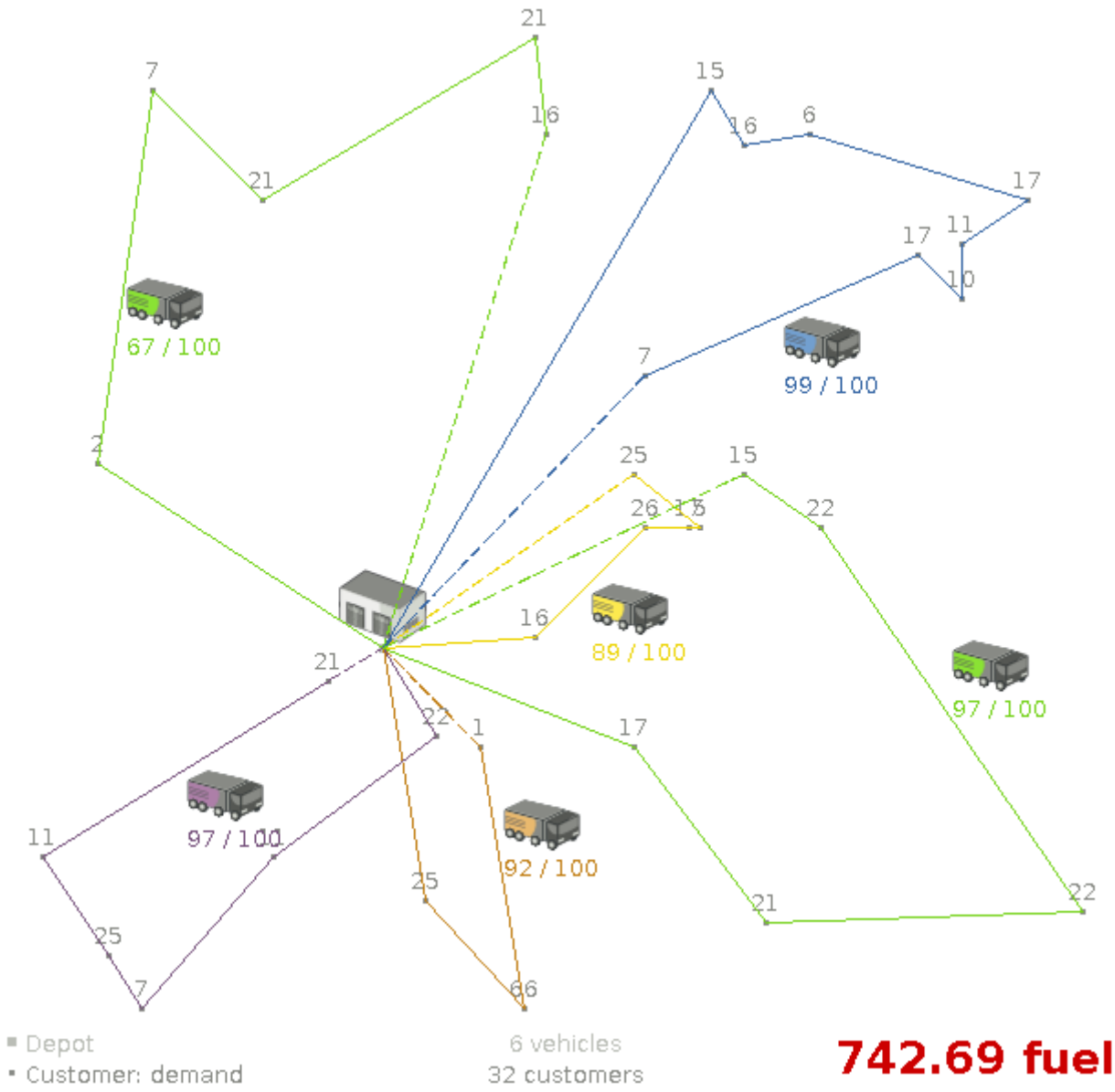
```
model_a1_1 has  2 resources,  1 neighborhoods,   4 locations,    4 machines,    79
 services,   100 processes and 1 balancePenalties with a search space of    10^60.
model_a1_2 has   4 resources,   2 neighborhoods,    4 locations,   100 machines,
    980 services,   1000 processes and 0 balancePenalties with a search space
 of   10^2000.
model_a1_3 has   3 resources,   5 neighborhoods,   25 locations,   100 machines,
    216 services,   1000 processes and 0 balancePenalties with a search space
 of   10^2000.
model_a1_4 has   3 resources, 50 neighborhoods,   50 locations,    50 machines,
    142 services,   1000 processes and 1 balancePenalties with a search space
 of   10^1698.
model_a1_5 has   4 resources,   2 neighborhoods,    4 locations,    12 machines,
    981 services,   1000 processes and 1 balancePenalties with a search space
 of   10^1079.
model_a2_1 has   3 resources,   1 neighborhoods,    1 locations,   100 machines,
   1000 services,   1000 processes and 0 balancePenalties with a search space
 of   10^2000.
model_a2_2 has 12 resources,   5 neighborhoods,   25 locations,   100 machines,
    170 services,   1000 processes and 0 balancePenalties with a search space
 of   10^2000.
```

```
model_a2_3 has 12 resources,  5 neighborhoods,  25 locations,  100 machines,
    129 services,  1000 processes and 0 balancePenalties with a search space
 of   10^2000.
model_a2_4 has 12 resources,  5 neighborhoods,  25 locations,   50 machines,
    180 services,  1000 processes and 1 balancePenalties with a search space
 of   10^1698.
model_a2_5 has 12 resources,  5 neighborhoods,  25 locations,   50 machines,
    153 services,  1000 processes and 0 balancePenalties with a search space
 of   10^1698.
model_b_1  has 12 resources,  5 neighborhoods,  10 locations,  100 machines,
   2512 services,  5000 processes and 0 balancePenalties with a search space
 of  10^10000.
model_b_2  has 12 resources,  5 neighborhoods,  10 locations,  100 machines,
   2462 services,  5000 processes and 1 balancePenalties with a search space
 of  10^10000.
model_b_3  has  6 resources,  5 neighborhoods,  10 locations,  100 machines,
 15025 services, 20000 processes and 0 balancePenalties with a search space of
  10^40000.
model_b_4  has  6 resources,  5 neighborhoods,  50 locations,  500 machines,
 1732 services, 20000 processes and 1 balancePenalties with a search space of
  10^53979.
model_b_5  has  6 resources,  5 neighborhoods,  10 locations,  100 machines,
 35082 services, 40000 processes and 0 balancePenalties with a search space of
  10^80000.
model_b_6  has  6 resources,  5 neighborhoods,  50 locations,  200 machines,
 14680 services, 40000 processes and 1 balancePenalties with a search space of
  10^92041.
model_b_7  has  6 resources,  5 neighborhoods,  50 locations, 4000 machines,
 15050 services, 40000 processes and 1 balancePenalties with a search space of
 10^144082.
model_b_8  has  3 resources,  5 neighborhoods,  10 locations,  100 machines,
 45030 services, 50000 processes and 0 balancePenalties with a search space of
 10^100000.
model_b_9  has  3 resources,  5 neighborhoods, 100 locations, 1000 machines,
 4609 services, 50000 processes and 1 balancePenalties with a search space of
 10^150000.
model_b_10 has  3 resources,  5 neighborhoods, 100 locations, 5000 machines,
 4896 services, 50000 processes and 1 balancePenalties with a search space of
 10^184948.
```

3.3.3. Vehicle routing

3.3.3.1. Problem statement

Using a fleet of vehicles, pick up the objects of each customer and bring them to the depot. Each vehicle can service multiple customers, but it has a limited capacity.



Besides the basic case (CVRP), there is also a variant with time windows (CVRPTW).

Hard constraints:

- Vehicle capacity: a vehicle cannot carry more items then its capacity.
- Time windows (only in CVRPTW):
  - Travel time: Traveling from one location to another takes time.
  - Customer service duration: a vehicle must stay at the customer for the length of the service duration.
  - Customer ready time: a vehicle may arrive before the customer's ready time, but it must wait until the ready time before servicing.

- Customer due time: a vehicle must arrive in time, before the customer's due time.

Soft constraints:

- Total distance: minimize the total distance driven (fuel consumption) of all vehicles.

The capacitated vehicle routing problem (CVRP) and it's timewindowed variant (CVRPTW) are defined by *the VRP web* [http://neo.lcc.uma.es/vrp/].

3.3.3.2. Problem size

CVRP instances (without time windows):

```
A-n32-k5  has 1 depots,  5 vehicles and  31 customers with a search space of  10^46.
A-n33-k5  has 1 depots,  5 vehicles and  32 customers with a search space of  10^48.
A-n33-k6  has 1 depots,  6 vehicles and  32 customers with a search space of  10^48.
A-n34-k5  has 1 depots,  5 vehicles and  33 customers with a search space of  10^50.
A-n36-k5  has 1 depots,  5 vehicles and  35 customers with a search space of  10^54.
A-n37-k5  has 1 depots,  5 vehicles and  36 customers with a search space of  10^56.
A-n37-k6  has 1 depots,  6 vehicles and  36 customers with a search space of  10^56.
A-n38-k5  has 1 depots,  5 vehicles and  37 customers with a search space of  10^58.
A-n39-k5  has 1 depots,  5 vehicles and  38 customers with a search space of  10^60.
A-n39-k6  has 1 depots,  6 vehicles and  38 customers with a search space of  10^60.
A-n44-k7  has 1 depots,  7 vehicles and  43 customers with a search space of  10^70.
A-n45-k6  has 1 depots,  6 vehicles and  44 customers with a search space of  10^72.
A-n45-k7  has 1 depots,  7 vehicles and  44 customers with a search space of  10^72.
A-n46-k7  has 1 depots,  7 vehicles and  45 customers with a search space of  10^74.
A-n48-k7  has 1 depots,  7 vehicles and  47 customers with a search space of  10^78.
A-n53-k7  has 1 depots,  7 vehicles and  52 customers with a search space of  10^89.
A-n54-k7  has 1 depots,  7 vehicles and  53 customers with a search space of  10^91.
A-n55-k9  has 1 depots,  9 vehicles and  54 customers with a search space of  10^93.
A-n60-k9  has 1 depots,  9 vehicles and  59 customers with a search space of
 10^104.
A-n61-k9  has 1 depots,  9 vehicles and  60 customers with a search space of
 10^106.
A-n62-k8  has 1 depots,  8 vehicles and  61 customers with a search space of
 10^108.
A-n63-k10 has 1 depots, 10 vehicles and  62 customers with a search space of
 10^111.
A-n63-k9  has 1 depots,  9 vehicles and  62 customers with a search space of
 10^111.
A-n64-k9  has 1 depots,  9 vehicles and  63 customers with a search space of
 10^113.
A-n65-k9  has 1 depots,  9 vehicles and  64 customers with a search space of
 10^115.
A-n69-k9  has 1 depots,  9 vehicles and  68 customers with a search space of
 10^124.
```

```
A-n80-k10 has 1 depots, 10 vehicles and  79 customers with a search space of
 10^149.
F-n135-k7 has 1 depots,  7 vehicles and 134 customers with a search space of
 10^285.
F-n45-k4  has 1 depots,  4 vehicles and  44 customers with a search space of  10^72.
F-n72-k4  has 1 depots,  4 vehicles and  71 customers with a search space of
 10^131.
```

CVRPTW instances (with time windows):

```
Solomon_025_C101        has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_025_C201        has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_025_R101        has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_025_R201        has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_025_RC101       has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_025_RC201       has 1 depots,  25 vehicles and   25 customers with a
 search space of   10^34.
Solomon_100_C101        has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Solomon_100_C201        has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Solomon_100_R101        has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Solomon_100_R201        has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Solomon_100_RC101       has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Solomon_100_RC201       has 1 depots,  25 vehicles and  100 customers with a
 search space of  10^200.
Homberger_0200_C1_2_1  has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
Homberger_0200_C2_2_1  has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
Homberger_0200_R1_2_1  has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
Homberger_0200_R2_2_1  has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
Homberger_0200_RC1_2_1 has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
Homberger_0200_RC2_2_1 has 1 depots,  50 vehicles and  200 customers with a
 search space of  10^460.
```

```
Homberger_0400_C1_4_1   has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0400_C2_4_1   has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0400_R1_4_1   has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0400_R2_4_1   has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0400_RC1_4_1 has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0400_RC2_4_1 has 1 depots, 100 vehicles and   400 customers with a
 search space of 10^1040.
Homberger_0600_C1_6_1   has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0600_C2_6_1   has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0600_R1_6_1   has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0600_R2_6_1   has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0600_RC1_6_1 has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0600_RC2_6_1 has 1 depots, 150 vehicles and   600 customers with a
 search space of 10^1666.
Homberger_0800_C1_8_1   has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_0800_C2_8_1   has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_0800_R1_8_1   has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_0800_R2_8_1   has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_0800_RC1_8_1 has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_0800_RC2_8_1 has 1 depots, 200 vehicles and   800 customers with a
 search space of 10^2322.
Homberger_1000_C110_1   has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
Homberger_1000_C210_1   has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
Homberger_1000_R110_1   has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
Homberger_1000_R210_1   has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
Homberger_1000_RC110_1 has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
Homberger_1000_RC210_1 has 1 depots, 250 vehicles and 1000 customers with a
 search space of 10^3000.
```

3.3.4. Project job scheduling

3.3.4.1. Problem statement

Schedule all jobs in time and execution mode to minimize project delays. Each job is part of a project. A job can be executed in different ways: each way is an execution mode that implies a different duration but also different resource usages. This is a form of flexible *job shop scheduling*.

Hard constraints:

- Job precedence: a job can only start when all its predecessor jobs are finished.
- Resource capacity: do not use more resources then available.
  - Resources are local (shared between jobs of the same project) or global (shared between all jobs)
  - Resource are renewable (capacity available per day) or nonrenewable (capacity available for all days)

Medium constraints:

- Total project delay: minimize the duration (makespan) of each project.

Soft constraints:

- Total makespan: minimize the duration of the whole multi-project schedule.

The problem is defined by *the MISTA 2013 challenge* [http://allserv.kahosl.be/mista2013challenge/].

3.3.4.2. Problem size

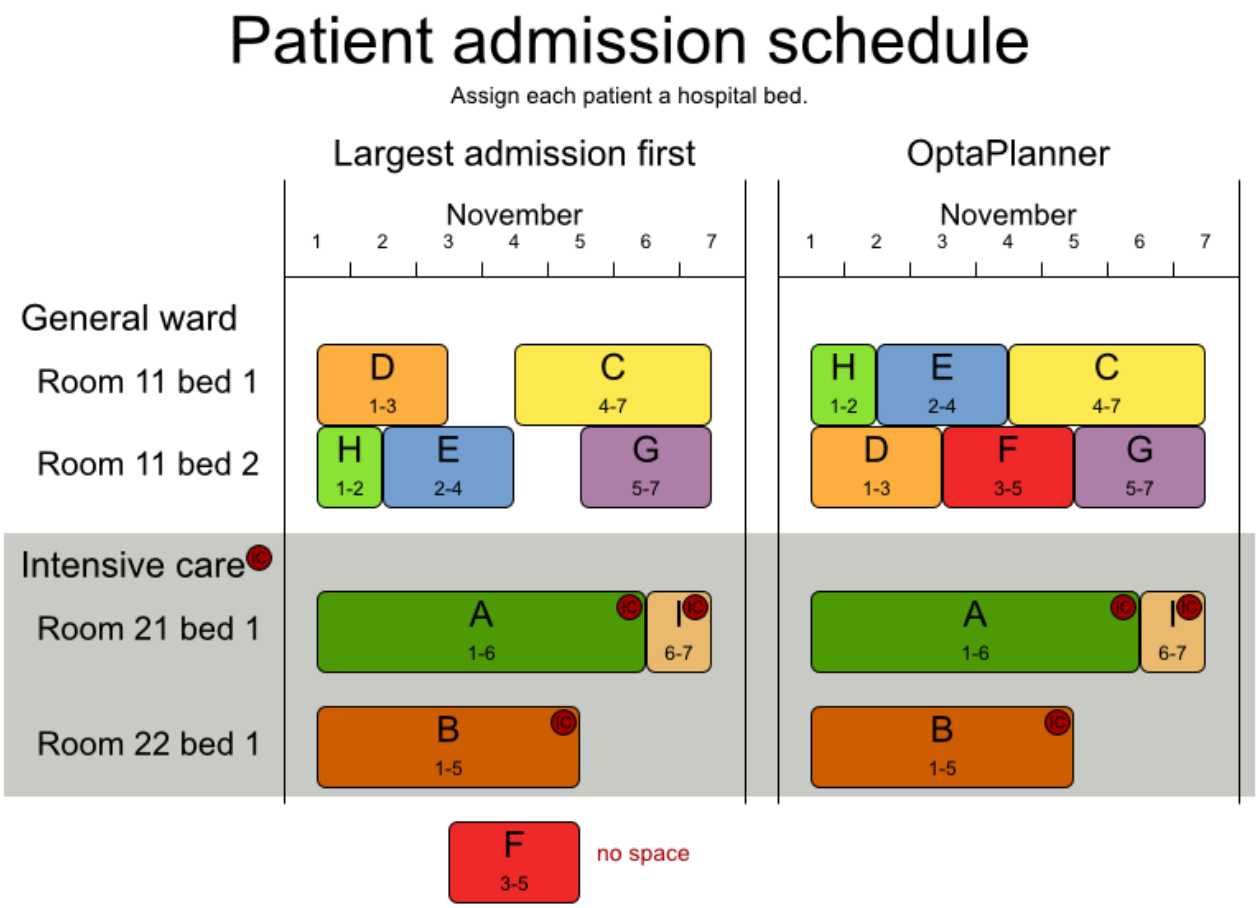
```
Schedule A-1  has  2 projects,  24 jobs,   64 execution modes,  7 resources
 and  150 resource requirements.
Schedule A-2  has  2 projects,  44 jobs,  124 execution modes,  7 resources
 and  420 resource requirements.
Schedule A-3  has  2 projects,  64 jobs,  184 execution modes,  7 resources
 and  630 resource requirements.
Schedule A-4  has  5 projects,  60 jobs,  160 execution modes, 16 resources
 and  390 resource requirements.
Schedule A-5  has  5 projects, 110 jobs,  310 execution modes, 16 resources
 and  900 resource requirements.
Schedule A-6  has  5 projects, 160 jobs,  460 execution modes, 16 resources
 and 1440 resource requirements.
```

```
Schedule A-7  has 10 projects, 120 jobs,  320 execution modes, 22 resources and
  900 resource requirements.
Schedule A-8  has 10 projects, 220 jobs,  620 execution modes, 22 resources and
 1860 resource requirements.
Schedule A-9  has 10 projects, 320 jobs,  920 execution modes, 31 resources and
 2880 resource requirements.
Schedule A-10 has 10 projects, 320 jobs,  920 execution modes, 31 resources and
 2970 resource requirements.
Schedule B-1  has 10 projects, 120 jobs,  320 execution modes, 31 resources and
  900 resource requirements.
Schedule B-2  has 10 projects, 220 jobs,  620 execution modes, 22 resources and
 1740 resource requirements.
Schedule B-3  has 10 projects, 320 jobs,  920 execution modes, 31 resources and
 3060 resource requirements.
Schedule B-4  has 15 projects, 180 jobs,  480 execution modes, 46 resources and
 1530 resource requirements.
Schedule B-5  has 15 projects, 330 jobs,  930 execution modes, 46 resources and
 2760 resource requirements.
Schedule B-6  has 15 projects, 480 jobs, 1380 execution modes, 46 resources and
 4500 resource requirements.
Schedule B-7  has 20 projects, 240 jobs,  640 execution modes, 61 resources and
 1710 resource requirements.
Schedule B-8  has 20 projects, 440 jobs, 1240 execution modes, 42 resources and
 3180 resource requirements.
Schedule B-9  has 20 projects, 640 jobs, 1840 execution modes, 61 resources and
 5940 resource requirements.
Schedule B-10 has 20 projects, 460 jobs, 1300 execution modes, 42 resources and
 4260 resource requirements.
```

3.3.5. Hospital bed planning (PAS - Patient admission scheduling)

3.3.5.1. Problem statement

Assign each patient (that will come to the hospital) into a bed for each night that the patient will stay in the hospital. Each bed belongs to a room and each room belongs to a department. The arrival and departure dates of the patients is fixed: only a bed needs to be assigned for each night.



Hard constraints:

- 2 patients must not be assigned to the same bed in the same night.
- A room can have a gender limitation: only females, only males, the same gender in the same night or no gender limitation at all.
- A department can have a minimum or maximum age.
- A patient can require a room with specific equipment(s).

Soft constraints:

- A patient can prefer a maximum room size, for example if he/she want a single room.
- A patient is best assigned to a department that specializes in his/her problem.
- A patient is best assigned to a room that specializes in his/her problem.
- A patient can prefer a room with specific equipment(s).

The problem is defined on *this webpage* [http://allserv.kahosl.be/~peter/pas/] and the test data comes from real world hospitals.

3.3.5.2. Problem size

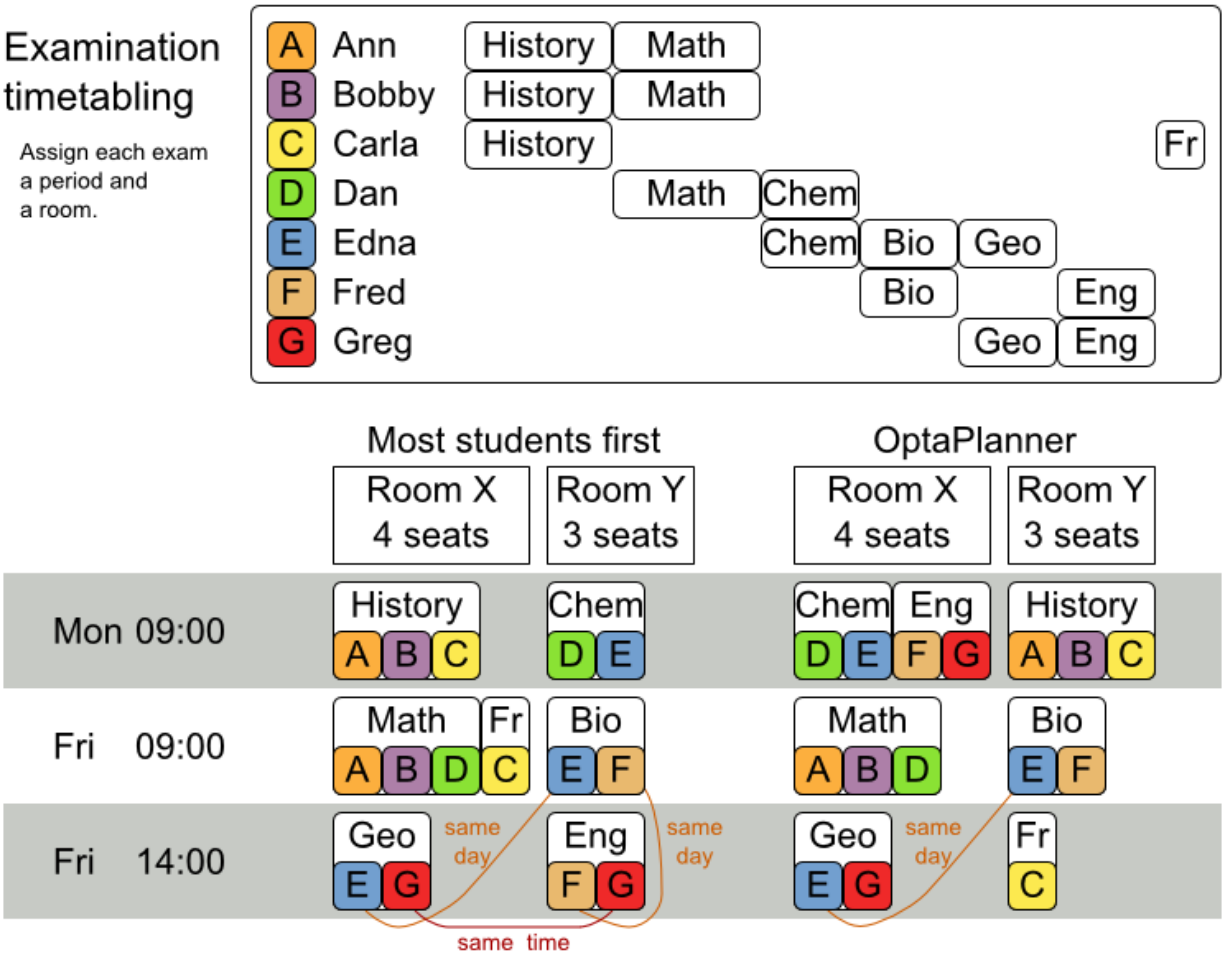
```
testdata01 has 4 specialisms, 2 equipments, 4 departments,  98 rooms, 286 beds,
 14 nights,  652 patients and  652 admissions with a search space of 10^1601.
testdata02 has 6 specialisms, 2 equipments, 6 departments, 151 rooms, 465 beds,
 14 nights,  755 patients and  755 admissions with a search space of 10^2013.
testdata03 has 5 specialisms, 2 equipments, 5 departments, 131 rooms, 395 beds,
 14 nights,  708 patients and  708 admissions with a search space of 10^1838.
testdata04 has 6 specialisms, 2 equipments, 6 departments, 155 rooms, 471 beds,
 14 nights,  746 patients and  746 admissions with a search space of 10^1994.
testdata05 has 4 specialisms, 2 equipments, 4 departments, 102 rooms, 325 beds,
 14 nights,  587 patients and  587 admissions with a search space of 10^1474.
testdata06 has 4 specialisms, 2 equipments, 4 departments, 104 rooms, 313 beds,
 14 nights,  685 patients and  685 admissions with a search space of 10^1709.
testdata07 has 6 specialisms, 4 equipments, 6 departments, 162 rooms, 472 beds,
 14 nights,  519 patients and  519 admissions with a search space of 10^1387.
testdata08 has 6 specialisms, 4 equipments, 6 departments, 148 rooms, 441 beds,
 21 nights,  895 patients and  895 admissions with a search space of 10^2366.
testdata09 has 4 specialisms, 4 equipments, 4 departments, 105 rooms, 310 beds,
 28 nights, 1400 patients and 1400 admissions with a search space of 10^3487.
testdata10 has 4 specialisms, 4 equipments, 4 departments, 104 rooms, 308 beds,
 56 nights, 1575 patients and 1575 admissions with a search space of 10^3919.
testdata11 has 4 specialisms, 4 equipments, 4 departments, 107 rooms, 318 beds,
 91 nights, 2514 patients and 2514 admissions with a search space of 10^6291.
testdata12 has 4 specialisms, 4 equipments, 4 departments, 105 rooms, 310 beds,
 84 nights, 2750 patients and 2750 admissions with a search space of 10^6851.
testdata13 has 5 specialisms, 4 equipments, 5 departments, 125 rooms, 368 beds,
 28 nights,  907 patients and 1109 admissions with a search space of 10^2845.
```

3.4. Difficult examples

3.4.1. Exam timetabling (ITC 2007 track 1 - Examination)

3.4.1.1. Problem statement

Schedule each exam into a period and into a room. Multiple exams can share the same room during the same period.



Hard constraints:

- Exam conflict: 2 exams that share students must not occur in the same period.
- Room capacity: A room's seating capacity must suffice at all times.
- Period duration: A period's duration must suffice for all of its exams.
- Period related hard constraints (specified per dataset):
  - Coincidence: 2 specified exams must use the same period (but possibly another room).
  - Exclusion: 2 specified exams must not use the same period.
  - After: A specified exam must occur in a period after another specified exam's period.
- Room related hard constraints (specified per dataset):
  - Exclusive: 1 specified exam should not have to share its room with any other exam.

Soft constraints (each of which has a parametrized penalty):

- The same student should not have 2 exams in a row.
- The same student should not have 2 exams on the same day.
- Period spread: 2 exams that share students should be a number of periods apart.
- Mixed durations: 2 exams that share a room should not have different durations.
- Front load: Large exams should be scheduled earlier in the schedule.
- Period penalty (specified per dataset): Some periods have a penalty when used.
- Room penalty (specified per dataset): Some rooms have a penalty when used.

It uses large test data sets of real-life universities.

The problem is defined by *the International Timetabling Competition 2007 track 1* [http://www.cs.qub.ac.uk/itc2007/examtrack/exam_track_index.htm]. Geoffrey De Smet finished 4th in that competition with a very early version of OptaPlanner. Many improvements have been made since then.

3.4.1.2. Problem size

```
exam_comp_set1 has  7883 students,  607 exams, 54 periods,  7 rooms,  12 period
 constraints and  0 room constraints with a search space of 10^1564.
exam_comp_set2 has 12484 students,  870 exams, 40 periods, 49 rooms,  12 period
 constraints and  2 room constraints with a search space of 10^2864.
exam_comp_set3 has 16365 students,  934 exams, 36 periods, 48 rooms, 168 period
 constraints and 15 room constraints with a search space of 10^3023.
exam_comp_set4 has  4421 students,  273 exams, 21 periods,  1 rooms,  40 period
 constraints and  0 room constraints with a search space of  10^360.
exam_comp_set5 has  8719 students, 1018 exams, 42 periods,  3 rooms,  27 period
 constraints and  0 room constraints with a search space of 10^2138.
exam_comp_set6 has  7909 students,  242 exams, 16 periods,  8 rooms,  22 period
 constraints and  0 room constraints with a search space of  10^509.
exam_comp_set7 has 13795 students, 1096 exams, 80 periods, 15 rooms,  28 period
 constraints and  0 room constraints with a search space of 10^3374.
exam_comp_set8 has  7718 students,  598 exams, 80 periods,  8 rooms,  20 period
 constraints and  1 room constraints with a search space of 10^1678.
```

3.4.1.3. Domain model

Below you can see the main examination domain classes:

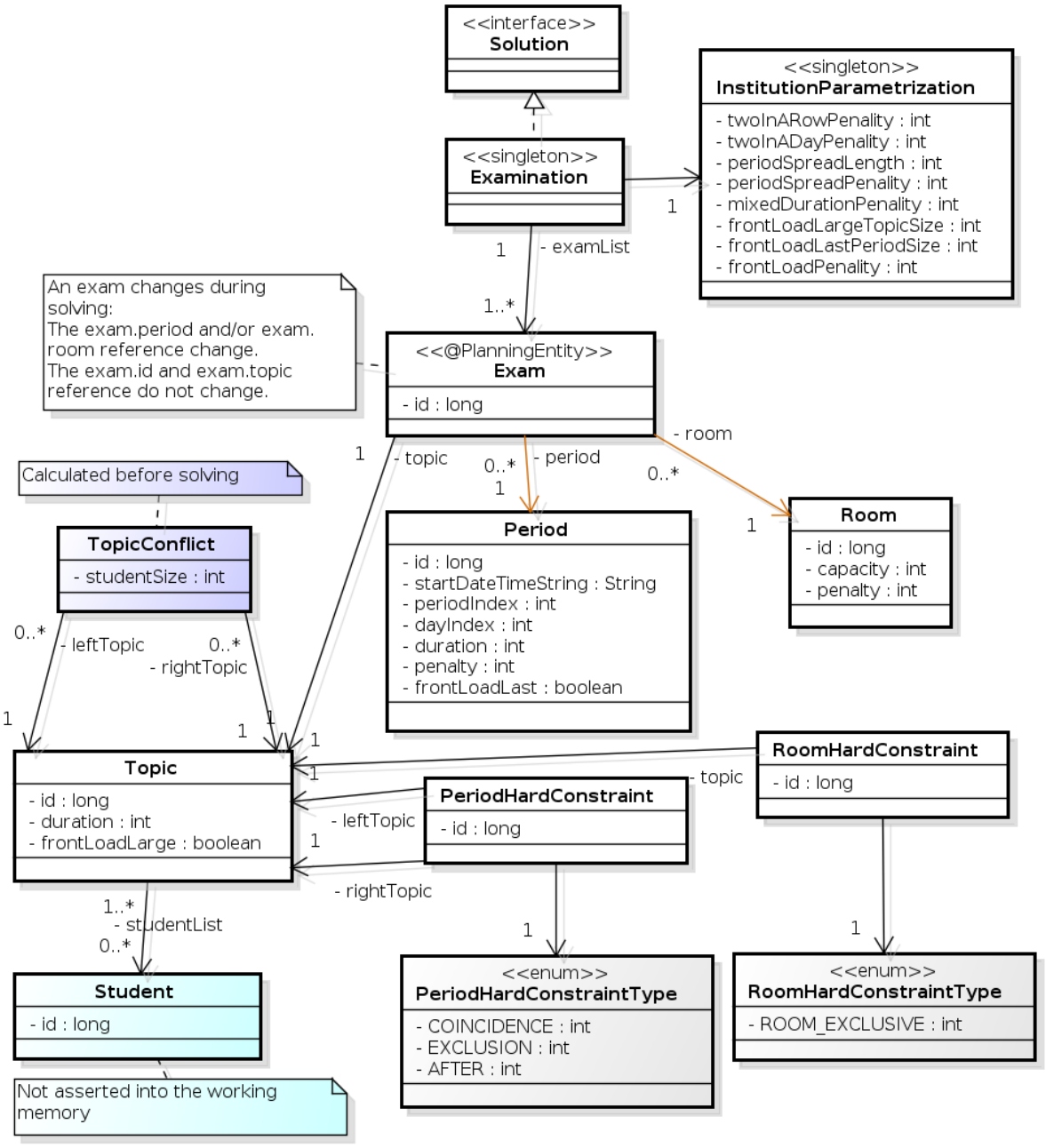


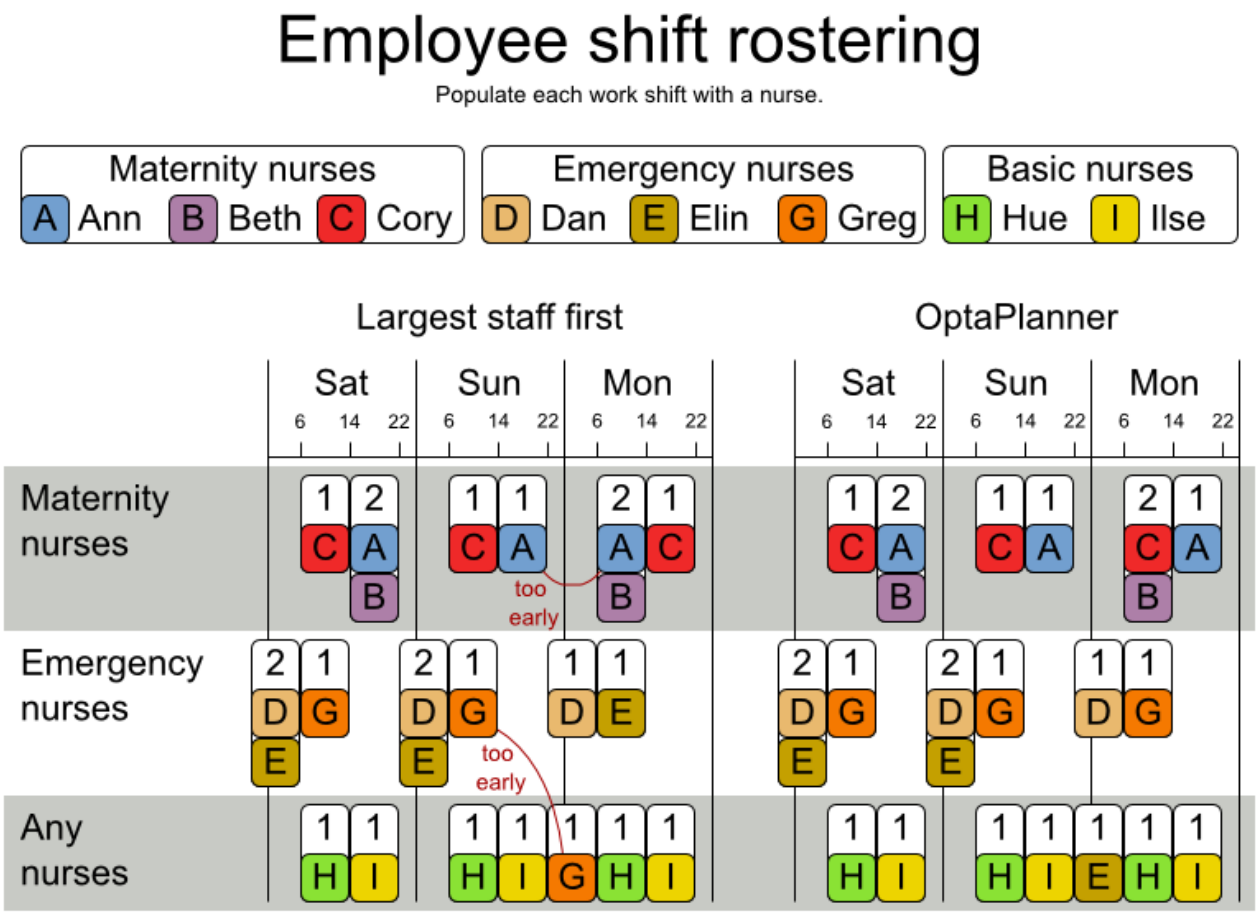
Figure 3.3. Examination domain class diagram

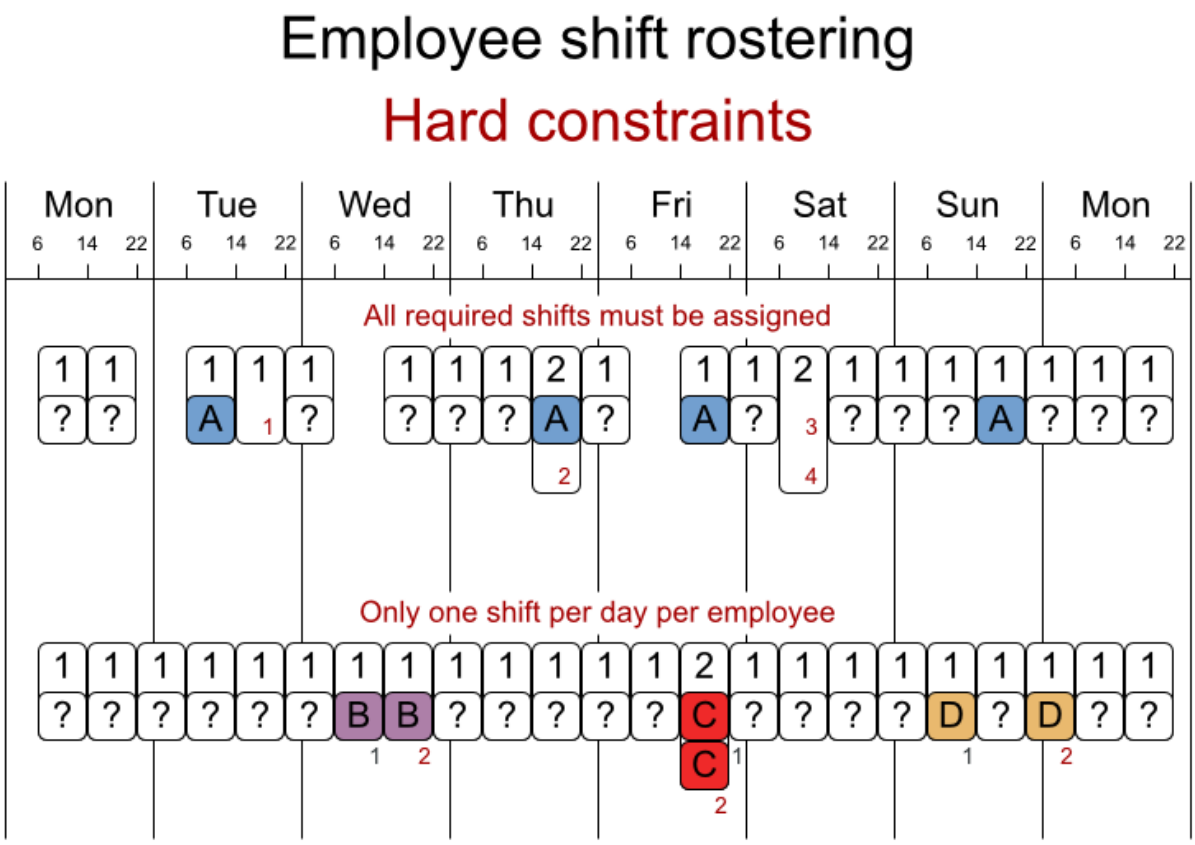
Notice that we've split up the exam concept into an `Exam` class and a `Topic` class. The `Exam` instances change during solving (this is the planning entity class), when their period or room property changes. The `Topic`, `Period` and `Room` instances never change during solving (these are problem facts, just like some other classes).

3.4.2. Employee rostering (INRC 2010 - Nurse rostering)

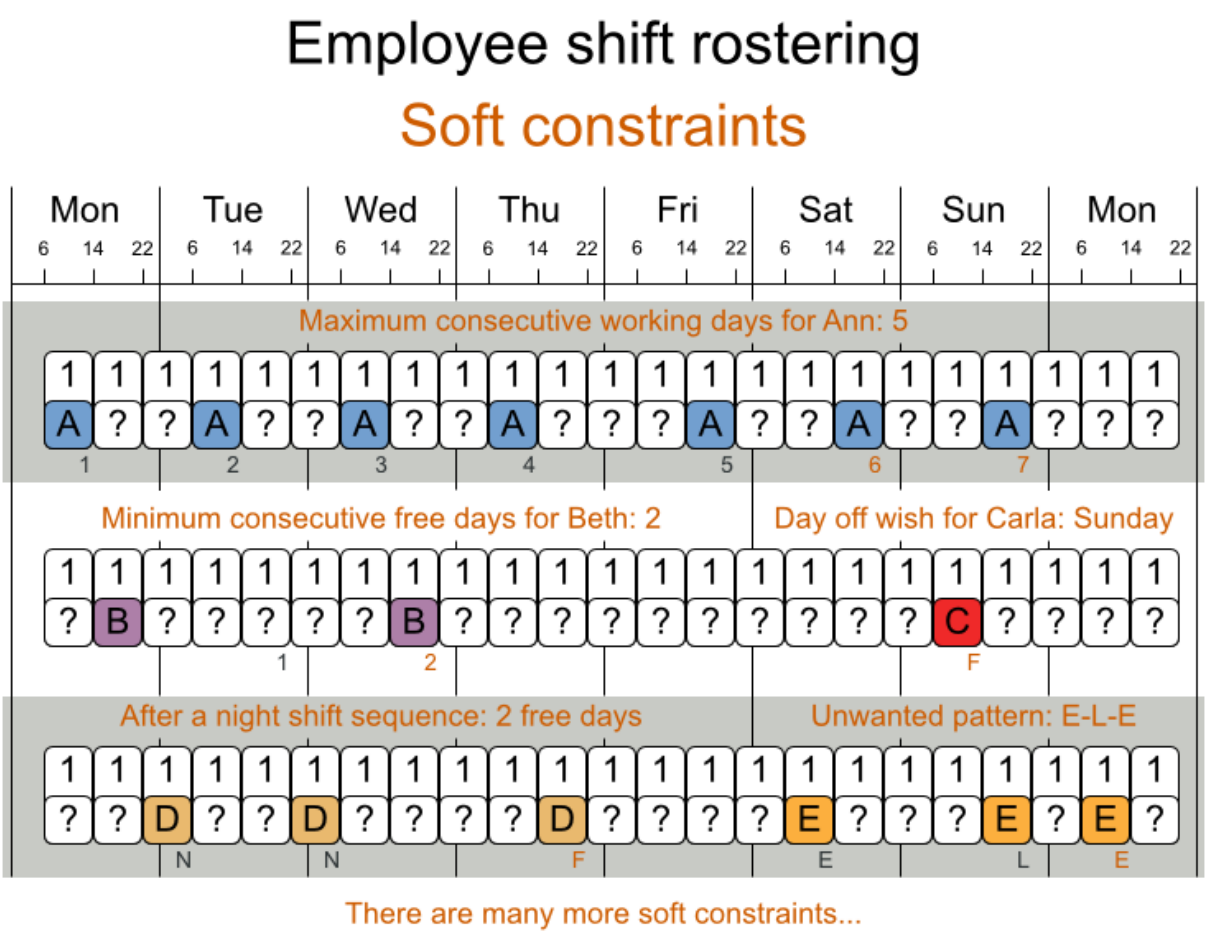
3.4.2.1. Problem statement

For each shift, assign a nurse to work that shift.





No hard constraint broken => solution is feasible



The problem is defined by *the International Nurse Rostering Competition 2010* [http://www.kuleuven-kortrijk.be/nrpcompetition].

3.4.2.2. Problem size

There are 3 dataset types:

- sprint: must be solved in seconds.
- medium: must be solved in minutes.
- long: must be solved in hours.

```
toy1         has 1 skills, 3 shiftTypes, 2 patterns, 1 contracts,  6 employees,  7
 shiftDates,  35 shiftAssignments and   0 requests with a search space of   10^27.
toy2         has 1 skills, 3 shiftTypes, 3 patterns, 2 contracts, 20 employees, 28
 shiftDates, 180 shiftAssignments and 140 requests with a search space of  10^234.

sprint01     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
```

```
sprint02     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint03     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint04     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint05     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint06     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint07     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint08     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint09     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint10     has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28
 shiftDates, 152 shiftAssignments and 150 requests with a search space of  10^152.
sprint_hint01 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_hint02 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_hint03 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_late01 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_late02 has 1 skills, 3 shiftTypes, 4 patterns, 3 contracts, 10 employees,
  28 shiftDates, 144 shiftAssignments and 139 requests with a search space of
   10^144.
sprint_late03 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 160 shiftAssignments and 150 requests with a search space of
   10^160.
sprint_late04 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 160 shiftAssignments and 150 requests with a search space of
   10^160.
sprint_late05 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_late06 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
sprint_late07 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
  28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
   10^152.
```

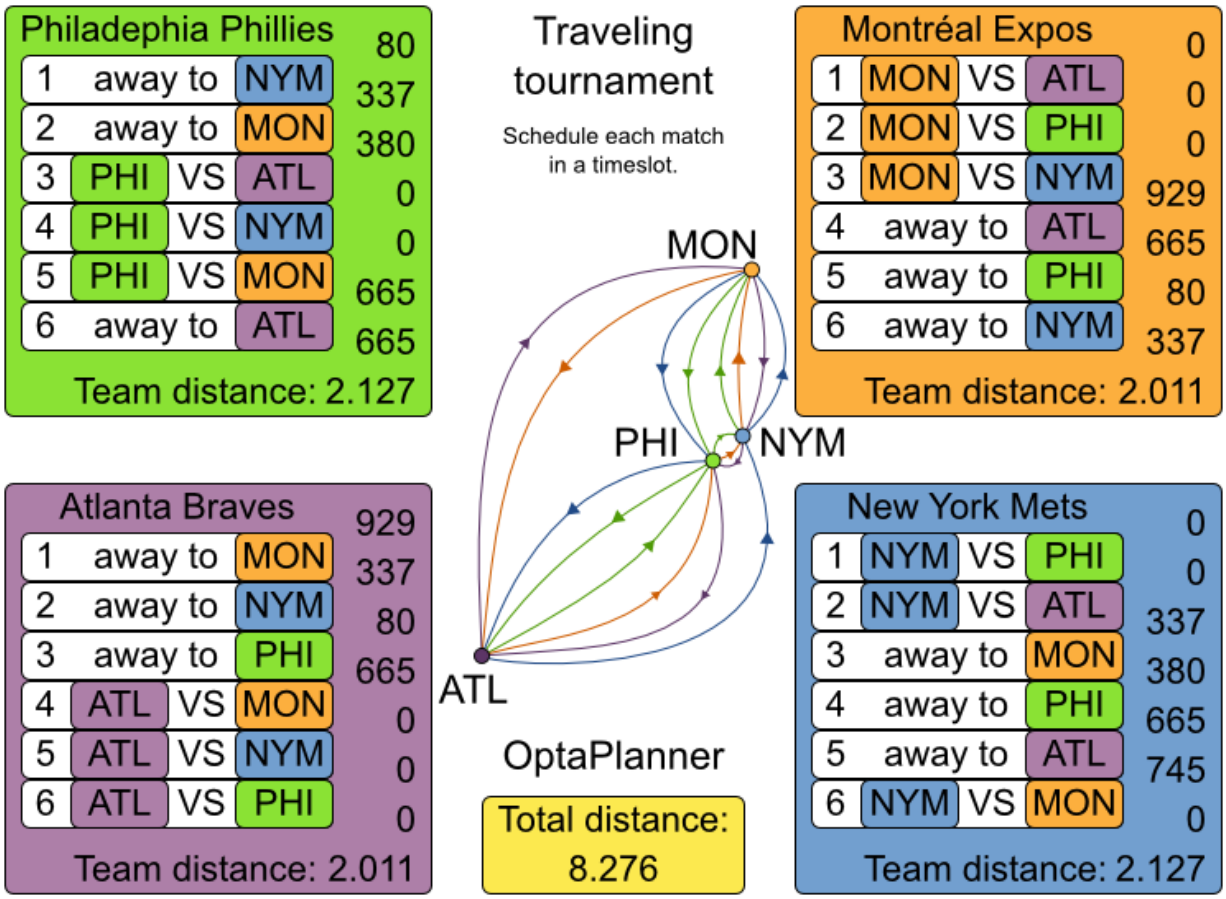
```
sprint_late08 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
 28 shiftDates, 152 shiftAssignments and   0 requests with a search space of
  10^152.
sprint_late09 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
 28 shiftDates, 152 shiftAssignments and   0 requests with a search space of
  10^152.
sprint_late10 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees,
 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of
  10^152.

medium01     has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28
 shiftDates, 608 shiftAssignments and 403 requests with a search space of  10^906.
medium02     has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28
 shiftDates, 608 shiftAssignments and 403 requests with a search space of  10^906.
medium03     has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28
 shiftDates, 608 shiftAssignments and 403 requests with a search space of  10^906.
medium04     has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28
 shiftDates, 608 shiftAssignments and 403 requests with a search space of  10^906.
medium05     has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28
 shiftDates, 608 shiftAssignments and 403 requests with a search space of  10^906.
medium_hint01 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees,
 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of
  10^632.
medium_hint02 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees,
 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of
  10^632.
medium_hint03 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees,
 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of
  10^632.
medium_late01 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees,
 28 shiftDates, 424 shiftAssignments and 390 requests with a search space of
  10^626.
medium_late02 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees,
 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of
  10^632.
medium_late03 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 30 employees,
 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of
  10^632.
medium_late04 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees,
 28 shiftDates, 416 shiftAssignments and 390 requests with a search space of
  10^614.
medium_late05 has 2 skills, 5 shiftTypes, 7 patterns, 4 contracts, 30 employees,
 28 shiftDates, 452 shiftAssignments and 390 requests with a search space of
  10^667.

long01       has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees,
 28 shiftDates, 740 shiftAssignments and 735 requests with a search space of
 10^1250.
```

```
long02       has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees,
 28 shiftDates, 740 shiftAssignments and 735 requests with a search space of
 10^1250.
long03       has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees,
 28 shiftDates, 740 shiftAssignments and 735 requests with a search space of
 10^1250.
long04       has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees,
 28 shiftDates, 740 shiftAssignments and 735 requests with a search space of
 10^1250.
long05       has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees,
 28 shiftDates, 740 shiftAssignments and 735 requests with a search space of
 10^1250.
long_hint01   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees,
 28 shiftDates, 740 shiftAssignments and   0 requests with a search space of
 10^1257.
long_hint02   has 2 skills, 5 shiftTypes, 7 patterns, 3 contracts, 50 employees,
 28 shiftDates, 740 shiftAssignments and   0 requests with a search space of
 10^1257.
long_hint03   has 2 skills, 5 shiftTypes, 7 patterns, 3 contracts, 50 employees,
 28 shiftDates, 740 shiftAssignments and   0 requests with a search space of
 10^1257.
long_late01   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees,
 28 shiftDates, 752 shiftAssignments and   0 requests with a search space of
 10^1277.
long_late02   has 2 skills, 5 shiftTypes, 9 patterns, 4 contracts, 50 employees,
 28 shiftDates, 752 shiftAssignments and   0 requests with a search space of
 10^1277.
long_late03   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees,
 28 shiftDates, 752 shiftAssignments and   0 requests with a search space of
 10^1277.
long_late04   has 2 skills, 5 shiftTypes, 9 patterns, 4 contracts, 50 employees,
 28 shiftDates, 752 shiftAssignments and   0 requests with a search space of
 10^1277.
long_late05   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees,
 28 shiftDates, 740 shiftAssignments and   0 requests with a search space of
 10^1257.
```

3.4.3. Sport scheduling (TTP - Traveling tournament problem)

3.4.3.1. Problem statement

Schedule matches between *n* teams.



Hard constraints:

- Each team plays twice against every other team: once home and once away.
- Each team has exactly 1 match on each timeslot.
- No team must have more than 3 consecutive home or 3 consecutive away matches.
- No repeaters: no 2 consecutive matches of the same 2 opposing teams.

Soft constraints:

- Minimize the total distance traveled by all teams.

The problem is defined on *Michael Trick's website (which contains the world records too)* [http://mat.gsia.cmu.edu/TOURN/].

3.4.3.2. Problem size

```
1-nl04     has  6 days,  4 teams and   12 matches with a search space of    10^9.
```

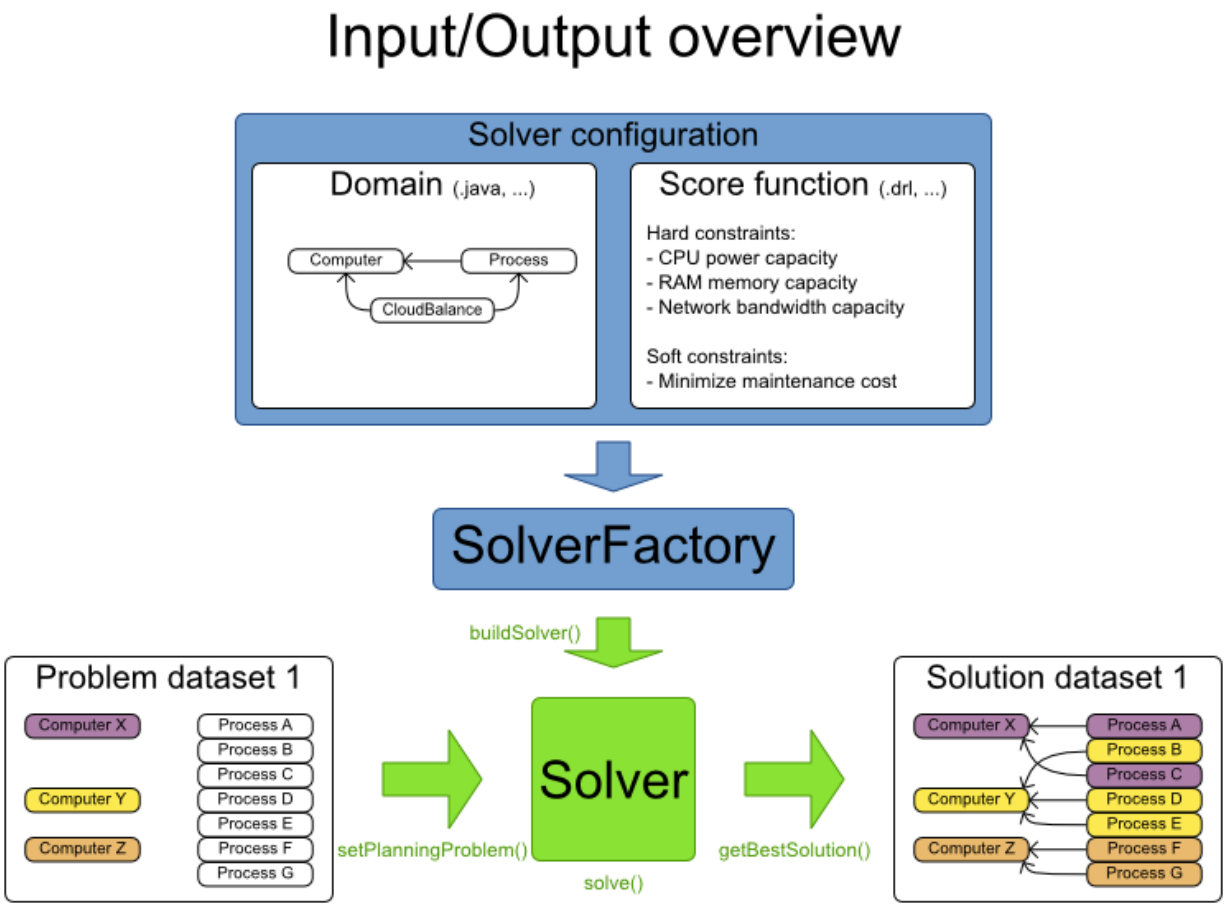
```
1-nl06     has 10 days,  6 teams and   30 matches with a search space of   10^30.
1-nl08     has 14 days,  8 teams and   56 matches with a search space of   10^64.
1-nl10     has 18 days, 10 teams and   90 matches with a search space of  10^112.
1-nl12     has 22 days, 12 teams and  132 matches with a search space of  10^177.
1-nl14     has 26 days, 14 teams and  182 matches with a search space of  10^257.
1-nl16     has 30 days, 16 teams and  240 matches with a search space of  10^354.
2-bra24    has 46 days, 24 teams and  552 matches with a search space of  10^917.
3-nfl16    has 30 days, 16 teams and  240 matches with a search space of  10^354.
3-nfl18    has 34 days, 18 teams and  306 matches with a search space of  10^468.
3-nfl20    has 38 days, 20 teams and  380 matches with a search space of  10^600.
3-nfl22    has 42 days, 22 teams and  462 matches with a search space of  10^749.
3-nfl24    has 46 days, 24 teams and  552 matches with a search space of  10^917.
3-nfl26    has 50 days, 26 teams and  650 matches with a search space of 10^1104.
3-nfl28    has 54 days, 28 teams and  756 matches with a search space of 10^1309.
3-nfl30    has 58 days, 30 teams and  870 matches with a search space of 10^1534.
3-nfl32    has 62 days, 32 teams and  992 matches with a search space of 10^1778.
4-super04  has  6 days,  4 teams and   12 matches with a search space of    10^9.
4-super06  has 10 days,  6 teams and   30 matches with a search space of   10^30.
4-super08  has 14 days,  8 teams and   56 matches with a search space of   10^64.
4-super10  has 18 days, 10 teams and   90 matches with a search space of  10^112.
4-super12  has 22 days, 12 teams and  132 matches with a search space of  10^177.
4-super14  has 26 days, 14 teams and  182 matches with a search space of  10^257.
5-galaxy04 has  6 days,  4 teams and   12 matches with a search space of    10^9.
5-galaxy06 has 10 days,  6 teams and   30 matches with a search space of   10^30.
5-galaxy08 has 14 days,  8 teams and   56 matches with a search space of   10^64.
5-galaxy10 has 18 days, 10 teams and   90 matches with a search space of  10^112.
5-galaxy12 has 22 days, 12 teams and  132 matches with a search space of  10^177.
5-galaxy14 has 26 days, 14 teams and  182 matches with a search space of  10^257.
5-galaxy16 has 30 days, 16 teams and  240 matches with a search space of  10^354.
5-galaxy18 has 34 days, 18 teams and  306 matches with a search space of  10^468.
5-galaxy20 has 38 days, 20 teams and  380 matches with a search space of  10^600.
5-galaxy22 has 42 days, 22 teams and  462 matches with a search space of  10^749.
5-galaxy24 has 46 days, 24 teams and  552 matches with a search space of  10^917.
5-galaxy26 has 50 days, 26 teams and  650 matches with a search space of 10^1104.
5-galaxy28 has 54 days, 28 teams and  756 matches with a search space of 10^1309.
5-galaxy30 has 58 days, 30 teams and  870 matches with a search space of 10^1534.
5-galaxy32 has 62 days, 32 teams and  992 matches with a search space of 10^1778.
5-galaxy34 has 66 days, 34 teams and 1122 matches with a search space of 10^2041.
5-galaxy36 has 70 days, 36 teams and 1260 matches with a search space of 10^2324.
5-galaxy38 has 74 days, 38 teams and 1406 matches with a search space of 10^2628.
5-galaxy40 has 78 days, 40 teams and 1560 matches with a search space of 10^2951.
```

Chapter 4. Planner configuration

4.1. Overview

Solving a planning problem with OptaPlanner consists out of 5 steps:

1. **Model your planning problem** as a class that implements the interface `Solution`, for example the class `NQueens`.
2. **Configure a `Solver`**, for example a first fit and tabu search solver for any `NQueens` instance.
3. **Load a problem data set** from your data layer, for example a 4 Queens instance. Set it as the planning problem on the `Solver` with `Solver.setPlanningProblem(...)`.
4. **Solve it** with `Solver.solve()`.
5. **Get the best solution found** by the `Solver` with `Solver.getBestSolution()`.



4.2. Solver configuration

4.2.1. Solver configuration by XML file

You can build a `Solver` instance with the `XmlSolverFactory`. Configure it with a solver configuration XML file:

```
XmlSolverFactory solverFactory = new XmlSolverFactory(
    "/org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml");
Solver solver = solverFactory.buildSolver();
```

A solver configuration file looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
  <!-- Define the model -->
              <solutionClass>org.optaplanner.examples.nqueens.domain.NQueens</
solutionClass>
          <planningEntityClass>org.optaplanner.examples.nqueens.domain.Queen</
planningEntityClass>

  <!-- Define the score function -->
  <scoreDirectorFactory>
    <scoreDefinitionType>SIMPLE</scoreDefinitionType>
     <scoreDrl>/org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</
scoreDrl>
  </scoreDirectorFactory>

  <!-- Configure the optimization algorithm(s) -->
  <termination>
    ...
  </termination>
  <constructionHeuristic>
    ...
  </constructionHeuristic>
  <localSearch>
    ...
  </localSearch>
</solver>
```

Notice the 3 parts in it:

- Define the model
- Define the score function

- Configure the optimization algorithm(s)

We'll explain these various parts of a configuration later in this manual.

OptaPlanner makes it relatively easy to switch optimization algorithm(s) just by changing the configuration. There's even a `Benchmark` utility which allows you to play out different configurations against each other and report the most appropriate configuration for your problem. You could for example play out tabu search versus simulated annealing, on 4 queens and 64 queens.

4.2.2. Solver configuration by Java API

As an alternative to the XML file, a solver configuration can also be configured with the `SolverConfig` API:

```
        SolverConfig solverConfig = new SolverConfig();

        solverConfig.setSolutionClass(NQueens.class);
        solverConfig.setPlanningEntityClassList(Collections.<Class<?>>singletonList(Queen.class

        ScoreDirectorFactoryConfig scoreDirectorFactoryConfig = new ScoreDirectorFactoryConfig(
        scoreDirectorFactoryConfig.setScoreDefinitionType(ScoreDirectorFactoryConfig.ScoreDefin
        scoreDirectorFactoryConfig.setScoreDrlList(
                       Arrays.asList("/org/optaplanner/examples/nqueens/solver/
nQueensScoreRules.drl"));
        solverConfig.setScoreDirectorFactoryConfig(scoreDirectorFactoryConfig);

        TerminationConfig terminationConfig = new TerminationConfig();
        // ...
        solverConfig.setTerminationConfig(terminationConfig);
        List<SolverPhaseConfig> solverPhaseConfigList = new ArrayList<SolverPhaseConfig>();
        ConstructionHeuristicSolverPhaseConfig constructionHeuristicSolverPhaseConfig
                = new ConstructionHeuristicSolverPhaseConfig();
        // ...
        solverPhaseConfigList.add(constructionHeuristicSolverPhaseConfig);
        LocalSearchSolverPhaseConfig localSearchSolverPhaseConfig = new LocalSearchSolverPhaseC
        // ...
        solverPhaseConfigList.add(localSearchSolverPhaseConfig);
        solverConfig.setSolverPhaseConfigList(solverPhaseConfigList);
        Solver solver = solverConfig.buildSolver();
```

It is highly recommended to configure by XML file instead of this API. To dynamically configure a value at runtime, use the XML file as a template and extract the `SolverConfig` class with `getSolverConfig()` to configure the dynamic value at runtime:

```
        XmlSolverFactory solverFactory = new XmlSolverFactory();
```

```
            "/org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml");

        SolverConfig solverConfig = solverFactory.getSolverConfig();
        solverConfig.getTerminationConfig().setMaximumMinutesSpend(userInput);
        Solver solver = solverConfig.buildSolver();
```

4.3. Model your planning problem

4.3.1. Is this class a problem fact or planning entity?

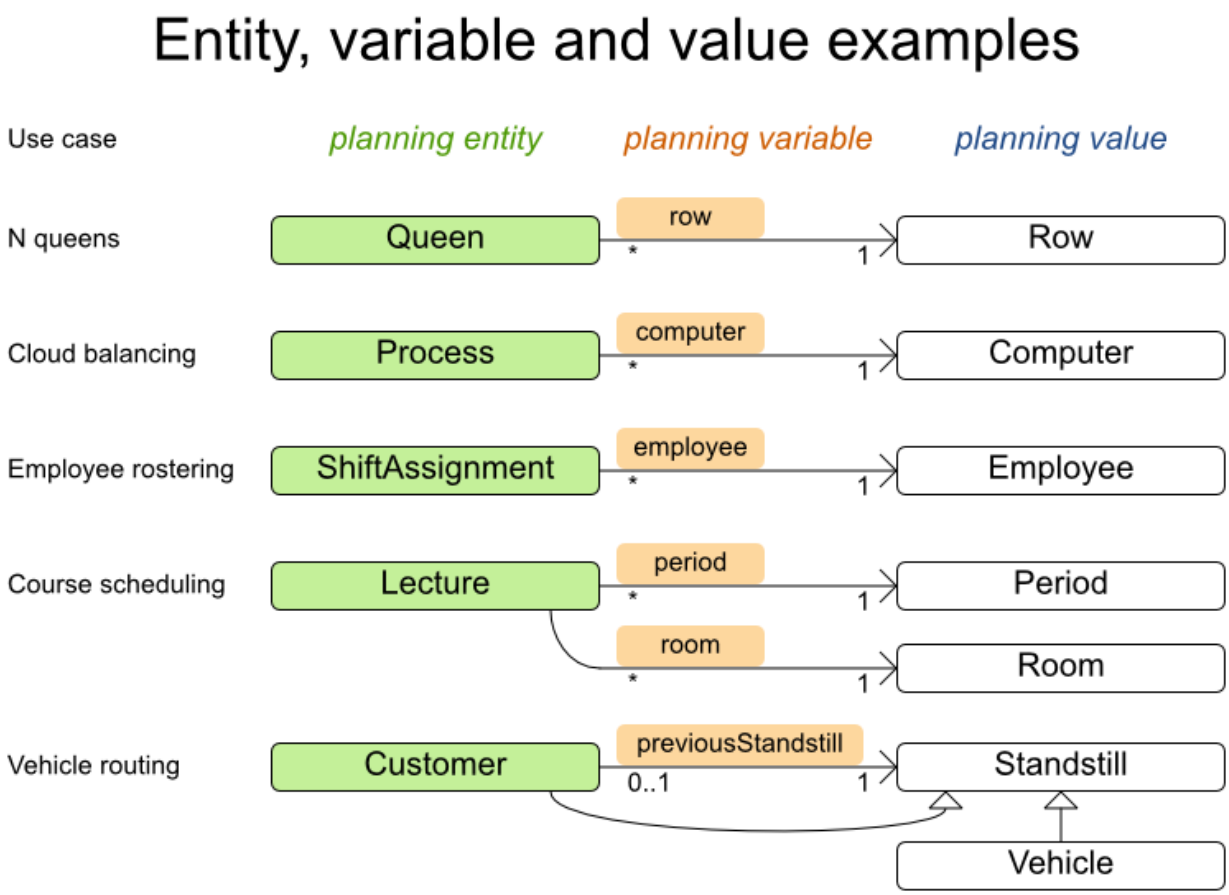
Look at a dataset of your planning problem. You 'll recognize domain classes in there, each of which is one of these:

- A unrelated class: not used by any of the score constraints. From a planning standpoint, this data is obsolete.
- A **problem fact** class: used by the score constraints, but does NOT change during planning (as long as the problem stays the same). For example: `Bed`, `Room`, `Shift`, `Employee`, `Topic`, `Period`, ...
- A **planning entity** class: used by the score constraints and changes during planning. For example: `BedDesignation`, `ShiftAssignment`, `Exam`, ...

Ask yourself: *What class changes during planning? Which class has variables that I want the `Solver` to change for me?* That class is a planning entity. Most use cases have only 1 planning entity class.

> **Note**
>
> In *real-time planning*, problem facts can change during planning, because the problem itself changes. However, that doesn't make them planning entities.

A good model can greatly improve the success of your planning implementation. For inspiration, take a look at how the examples modeled their domain:



When in doubt, it's usually the many side of a many to one relationship that is the planning entity. For example in employee rostering, the planning entity class is `ShiftAssignment`, not `Employee`. Vehicle routing is special, because it uses a *chained planning variable*.

In OptaPlanner all problems facts and planning entities are plain old JavaBeans (POJO's). You can load them from a database (JDBC/JPA/JDO), an XML file, a data repository, a noSQL cloud, ...: OptaPlanner doesn't care.

4.3.2. Problem fact

A problem fact is any JavaBean (POJO) with getters that does not change during planning. Implementing the interface `Serializable` is recommended (but not required). For example in n queens, the columns and rows are problem facts:

```
public class Column implements Serializable {

    private int index;

    // ... getters
```

```
}
```

```
public class Row implements Serializable {

    private int index;

    // ... getters
}
```

A problem fact can reference other problem facts of course:

```
public class Course implements Serializable {

    private String code;

    private Teacher teacher; // Other problem fact
    private int lectureSize;
    private int minWorkingDaySize;

    private List<Curriculum> curriculumList; // Other problem facts
    private int studentSize;

    // ... getters
}
```

A problem fact class does *not* require any Planner specific code. For example, you can reuse your domain classes, which might have JPA annotations.

> **Note**
>
> Generally, better designed domain classes lead to simpler and more efficient score constraints. Therefore, when dealing with a messy legacy system, it can sometimes be worth it to convert the messy domain set into a planner specific POJO set first. For example: if your domain model has 2 `Teacher` instances for the same teacher that teaches at 2 different departments, it's hard to write a correct score constraint that constrains a teacher's spare time.
>
> Alternatively, you can sometimes also introduce *a cached problem fact* to enrich the domain model for planning only.

4.3.3. Planning entity

4.3.3.1. Planning entity annotation

A planning entity is a JavaBean (POJO) that changes during solving, for example a `Queen` that changes to another row. A planning problem has multiple planning entities, for example for a single n queens problem, each `Queen` is a planning entity. But there's usually only 1 planning entity class, for example the `Queen` class.

A planning entity class needs to be annotated with the `@PlanningEntity` annotation.

Each planning entity class has 1 or more *planning variables*. It usually also has 1 or more *defining* properties. For example in n queens, a `Queen` is defined by its `Column` and has a planning variable `Row`. This means that a Queen's column never changes during solving, while its row does change.

```
@PlanningEntity
public class Queen {

    private Column column;

    // Planning variables: changes during planning, between score calculations.
    private Row row;

    // ... getters and setters
}
```

A planning entity class can have multiple planning variables. For example, a `Lecture` is defined by its `Course` and its index in that course (because 1 course has multiple lectures). Each `Lecture` needs to be scheduled into a `Period` and a `Room` so it has 2 planning variables (period and room). For example: the course Mathematics has 8 lectures per week, of which the first lecture is Monday morning at 08:00 in room 212.

```
@PlanningEntity
public class Lecture {

    private Course course;
    private int lectureIndexInCourse;

    // Planning variables: changes during planning, between score calculations.
    private Period period;
    private Room room;

    // ...
}
```

The solver configuration also needs to be made aware of each planning entity class:

```
<solver>
  ...
          <planningEntityClass>org.optaplanner.examples.nqueens.domain.Queen</
planningEntityClass>
  ...
</solver>
```

Some uses cases have multiple planning entity classes. For example: route freight and trains into railway network arcs, where each freight can use multiple trains over its journey and each train can carry multiple freights per arc. Having multiple planning entity classes directly raises the implementation complexity of your use case.

> **Note**
>
> *Do not create unnecessary planning entity classes.* This leads to difficult `Move` implementations and slower score calculation.
>
> For example, do not create a planning entity class to hold the total free time of a teacher, which needs to be kept up to date as the `Lecture` planning entities change. Instead, calculate the free time in the score constraints and put the result per teacher into a logically inserted score object.
>
> If historic data needs to be considered too, then create problem fact to hold the historic data up to, but *not including*, the planning window (so it doesn't change when a planning entity changes) and let the score constraints take it into account.

4.3.3.2. Planning entity difficulty

Some optimization algorithms work more efficiently if they have an estimation of which planning entities are more difficult to plan. For example: in bin packing bigger items are harder to fit, in course scheduling lectures with more students are more difficult to schedule and in n queens the middle queens are more difficult to fit on the board.

Therefore, you can set a `difficultyComparatorClass` to the `@PlanningEntity` annotation:

```
@PlanningEntity(difficultyComparatorClass = CloudProcessDifficultyComparator.class)
public class CloudProcess {
    // ...
}
```

```
public class CloudProcessDifficultyComparator implements Comparator<CloudProcess> {
```

```
    public int compare(CloudProcess a, CloudProcess b) {
        return new CompareToBuilder()
              .append(a.getRequiredMultiplicand(), b.getRequiredMultiplicand())
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```

Alternatively, you can also set a `difficultyWeightFactoryClass` to the `@PlanningEntity` annotation, so you have access to the rest of the problem facts from the `Solution` too:

```
@PlanningEntity(difficultyWeightFactoryClass = QueenDifficultyWeightFactory.class)
public class Queen {
    // ...
}
```

See *Sorted Selection* for more information.

Important

Difficulty should be implemented ascending: easy entities are lower, difficult entities are higher. For example in bin packing: small item < medium item < big item.

Even though some algorithms start with the more difficult entities first, they just reverse the ordering.

None of the current planning variable state should be used to compare planning entity difficult. During construction heuristics, those variables are likely to be `null` anyway. For example, a `Queen`'s `row` variable should not be used.

4.3.4. Planning variable

4.3.4.1. Planning variable annotation

A planning variable is a property (including getter and setter) on a planning entity. It points to a planning value, which changes during planning. For example, a `Queen`'s `row` property is a planning variable. Note that even though a `Queen`'s `row` property changes to another `Row` during planning, no `Row` instance itself is changed.

A planning variable getter needs to be annotated with the `@PlanningVariable` annotation, which needs a non-empty `valueRangeProviderRefs` property.

```
@PlanningEntity
public class Queen {

    private Row row;

    // ...

    @PlanningVariable(valueRangeProviderRefs = {"rowRange"})
    public Row getRow() {
        return row;
    }

    public void setRow(Row row) {
        this.row = row;
    }

}
```

The `valueRangeProviderRefs` property defines what are the possible planning values for this planning variable. It references 1 or more `@ValueRangeProvider` `id`'s.

4.3.4.2. Nullable planning variable

By default, an initialized planning variable cannot be `null`, so an initialized solution will never use `null` for any of its planning variables. In an over-constrained use case, this can be contra productive. For example: in task assignment with too many tasks for the workforce, we would rather leave low priority tasks unassigned instead of assigning them to an overloaded worker.

To allow an initialized planning variable to be `null`, set `nullable` to `true`:

```
    @PlanningVariable(..., nullable = true)
    public Worker getWorker() {
        return worker;
    }
```

Important

Planner will automatically add the value `null` to the value range. There is no need to add `null` in a collection used by a `ValueRangeProvider`.

> **Note**
>
> Using a nullable planning variable implies that your score calculation is responsible for punishing (or even rewarding) variables with a null value.

Repeated planning (especially *real-time planning*) does not mix well with a nullable planning variable: every time the Solver starts or a problem fact change is made, the construction heuristics will try to initialize all the null variables again, which can be a huge waste of time. One way to deal with this, is to change when a planning entity should be reinitialized with an `reinitializeVariableEntityFilter`:

```
@PlanningVariable(..., nullable = true, reinitializeVariableEntityFilter = ReinitializeTask
public Worker getWorker() {
    return worker;
}
```

4.3.4.3. When is a planning variable considered initialized?

A planning variable is considered initialized if its value is not `null` or if the variable is `nullable`. So a nullable variable is always considered initialized, even when a custom `reinitializeVariableEntityFilter` triggers a reinitialization during construction heuristics.

A planning entity is initialized if all of its planning variables are initialized.

A `Solution` is initialized if all of its planning entities are initialized.

4.3.5. Planning value and planning value ranges

4.3.5.1. Planning value

A planning value is a possible value for a planning variable. Usually, a planning value is a problem fact, but it can also be any object, for example a double. It can even be another planning entity or even a interface implemented by a planning entity and a problem fact.

A planning value range is the set of possible planning values for a planning variable. This set can be a discrete (for example row `1`, `2`, `3` or `4`) or continuous (for example any `double` between `0.0` and `1.0`).

There are several ways to define the value range of a planning variable with the `@ValueRangeProvider` annotation. A `@ValueRangeProvider` annotation always has a property `id`, which is referenced by the `@PlanningVariable`'s property `valueRangeProviderRefs`.

4.3.5.2. Planning value range provider

4.3.5.2.1. `ValueRangeProvider` on a `Solution`

All instances of the same planning entity class share the same set of possible planning values for that planning variable. This is the most common way to configure a value range.

The `Solution` implementation has method which returns a `Collection`. Any value from that `Collection` is a possible planning value for this planning variable.

```
    @PlanningVariable(valueRangeProviderRefs = {"rowRange"})
    public Row getRow() {
        return row;
    }
```

```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {

    // ...

    @ValueRangeProvider(id = "rowRange")
    public List<Row> getRowList() {
        return rowList;
    }

}
```

That `Collection` must not contain the value `null`, even for a *nullable planning variable*.

4.3.5.2.2. `ValueRangeProvider` on a planning entity

Each planning entity has its own set of possible planning values for a planning variable. For example, if a teacher can **never** teach in a room that does not belong to his department, lectures of that teacher can limit their room value range to the rooms of his department.

```
    @PlanningVariable(valueRangeProviderRefs = {"possibleRoomRange"})
    public Room getRoom() {
        return room;
    }

    @ValueRangeProvider(id = "possibleRoomRange")
    public List<Room> getPossibleRoomList() {
        return getCourse().getTeacher().getPossibleRoomList();
    }
```

Never use this to enforce a soft constraint (or even a hard constraint when the problem might not have a feasible solution). For example: *Unless there is no other way*, a teacher can not teach in a room that does not belong to his department. In this case, the teacher should *not* be limited in his room value range (because sometimes there is no other way).

> **Note**
>
> By limiting the value range specifically of 1 planning entity, you are effectively creating a *build-in hard constraint*. This can be a very good thing, as the number of possible solutions is severely lowered. But this can also be a bad thing because it takes away the freedom of the optimization algorithms to temporarily break that constraint in order to escape a local optima.

A planning entity should *not* use other planning entities to determinate its value range. That would only try to make it solve the planning problem itself and interfere with the optimization algorithms.

This value range is not (yet) compatible with a *chained* variable.

4.3.5.2.3. Combining ValueRangeProviders

Value range providers can be combined, for example:

```
@PlanningVariable(valueRangeProviderRefs = {"companyCarRange", "personalCarRange"})
public Car getCar() {
    return car;
}
```

```
@ValueRangeProvider(id = "companyCarRange")
public List<CompanyCar> getCompanyCarList() {
    return companyCarList;
}

@ValueRangeProvider(id = "personalCarRange")
public List<PersonalCar> getPersonalCarList() {
    return personalCarList;
}
```

4.3.5.3. Planning value strength

Some optimization algorithms work more efficiently if they have an estimation of which planning values are stronger, which means they are more likely to satisfy a planning entity. For example: in bin packing bigger containers are more likely to fit an item and in course scheduling bigger rooms are less likely to break the student capacity constraint.

Therefore, you can set a `strengthComparatorClass` to the `@PlanningVariable` annotation:

```
    @PlanningVariable(..., strengthComparatorClass = CloudComputerStrengthComparator.class)
    public CloudComputer getComputer() {
        // ...
    }
```

```
public class CloudComputerStrengthComparator implements Comparator<CloudComputer> {

    public int compare(CloudComputer a, CloudComputer b) {
        return new CompareToBuilder()
                .append(a.getMultiplicand(), b.getMultiplicand())
                    .append(b.getCost(), a.getCost()) // Descending (but this
 is debatable)
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```

> **Note**
>
> If you have multiple planning value classes in the *same* value range, the `strengthComparatorClass` needs to implement a `Comparator` of a common superclass (for example `Comparator<Object>`) and be able to handle comparing instances of those different classes.

Alternatively, you can also set a `strengthWeightFactoryClass` to the `@PlanningVariable` annotation, so you have access to the rest of the problem facts from the solution too:

```
    @PlanningVariable(..., strengthWeightFactoryClass = RowStrengthWeightFactory.class)
    public Row getRow() {
        // ...
    }
```

See *Sorted Selection* for more information.

Important

Strength should be implemented ascending: weaker values are lower, stronger values are higher. For example in bin packing: small container < medium container < big container.

None of the current planning variable state in any of the planning entities should be used to compare planning values. During construction heuristics, those variables are likely to be `null` anyway. For example, none of the `row` variables of any `Queen` may be used to determine the strength of a `Row`.

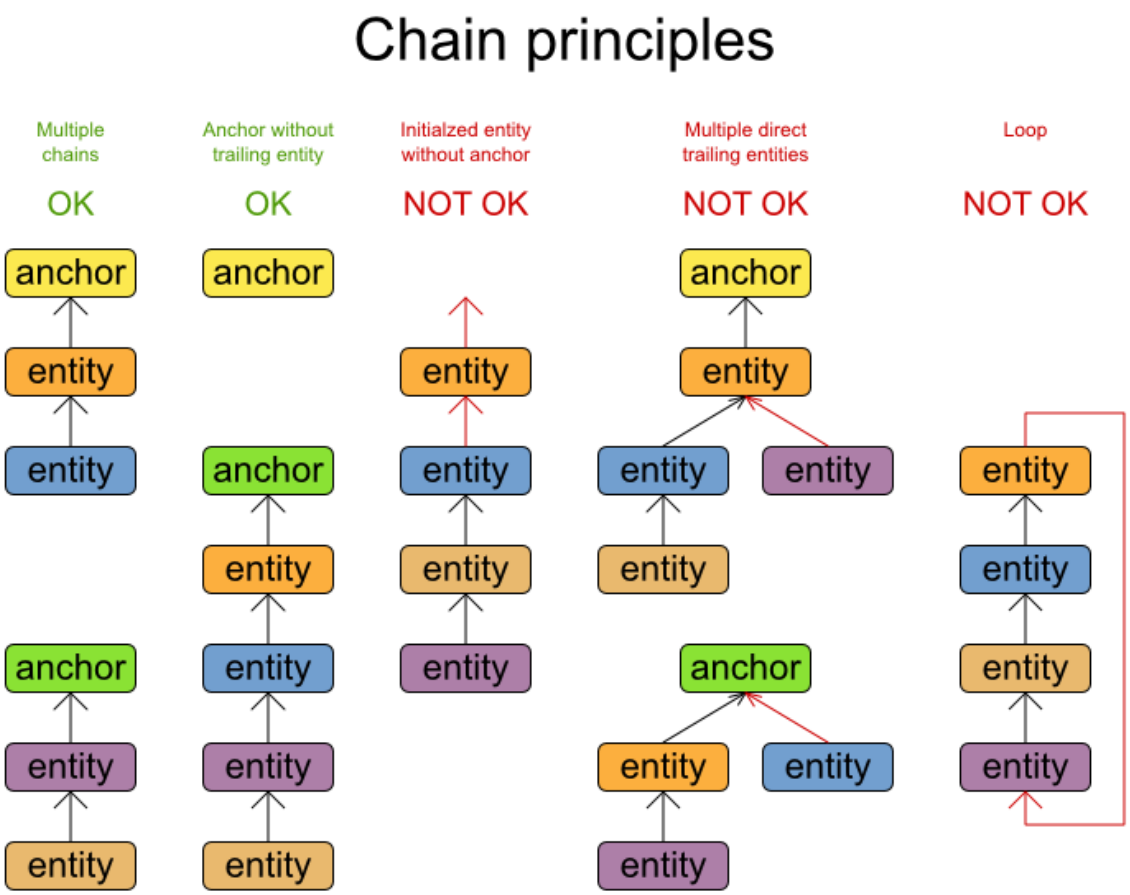
4.3.5.4. Chained planning variable

Some use cases, such as TSP and Vehicle Routing, require *chaining*. This means the planning entities point to each other and form a chain.

A planning variable that is chained either:

- Directly points to a planning fact, which is called an *anchor*.
- Points to another planning entity with the same planning variable, which recursively points to an anchor.

Here are some example of valid and invalid chains:



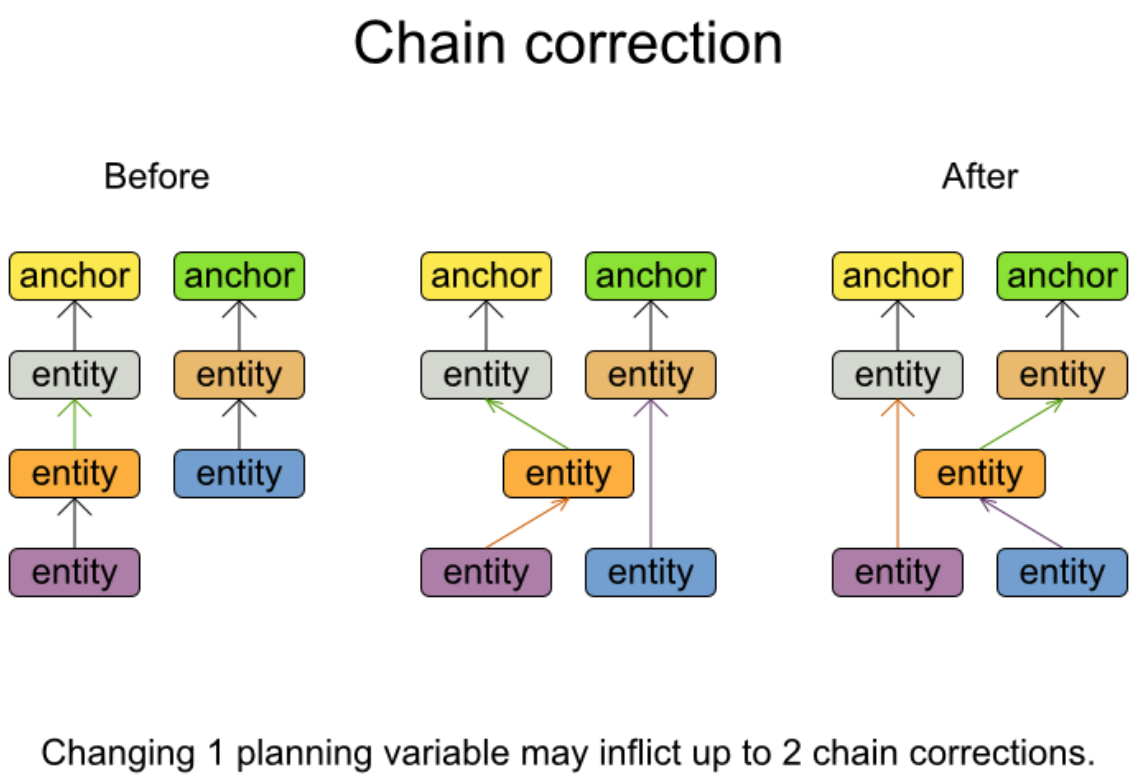
Every initialized planning entity is part of an open-ended chain that begins from an anchor. A valid model means that:

- A chain is never a loop. The tail is always open.
- Every chain always has exactly 1 anchor. The anchor is a problem fact, never a planning entity.
- A chain is never a tree, it is always a line. Every anchor or planning entity has at most 1 trailing planning entity.
- Every initialized planning entity is part of a chain.
- An anchor with no planning entities pointing to it, is also considered a chain.

> **Warning**
>
> A planning problem instance given to the `Solver` must be valid.

> **Note**
>
> If your constraints dictate a closed chain, model it as an open-ended chain (which is easier to persist in a database) and implement a score constraint for the last entity back to the anchor.

The optimization algorithms and build-in `Move`'s do chain correction to guarantee that the model stays valid:



> **Warning**
>
> A custom `Move` implementation must leave the model in a valid state.

For example, in TSP the anchor is a `Domicile` (in vehicle routing it is `Vehicle`):

```
public class Domicile ... implements Standstill {
```

```
    ...

    public City getCity() {...}

}
```

The anchor (which is a problem fact) and the planning entity implement a common interface, for example TSP's `Standstill`:

```
public interface Standstill {

    City getCity();

}
```

That interface is the return type of the planning variable. Furthermore, the planning variable is chained. For example TSP's `Visit` (in vehicle routing it is `Customer`):

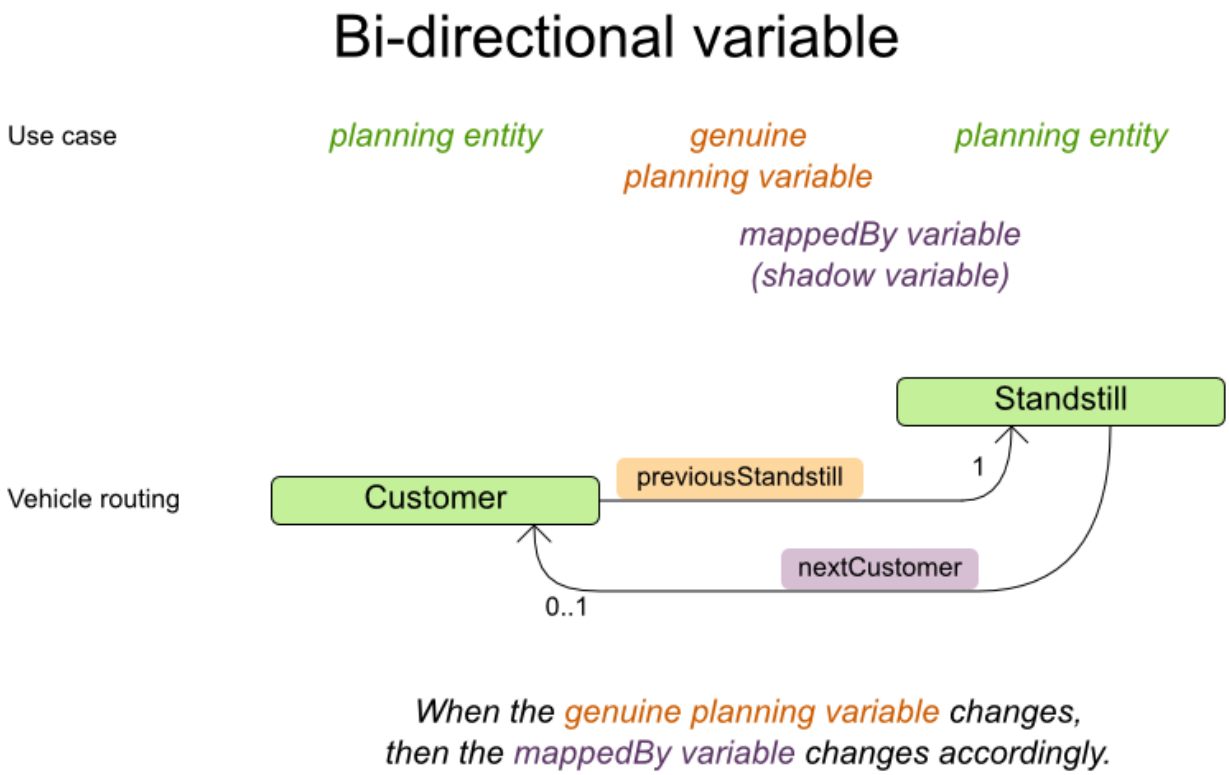
```
@PlanningEntity
public class Visit ... implements Standstill {

    ...

    public City getCity() {...}

    @PlanningVariable(chained = true, valueRangeProviderRefs = {"domicileRange", "visitRange"})
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }

    public void setPreviousStandstill(Standstill previousStandstill) {
        this.previousStandstill = previousStandstill;
    }

}
```

Notice how 2 value range providers are usually combined:

- The value range provider which holds the anchors, for example `domicileList`.
- The value range provider which holds the initialized planning entities, for example `visitList`.

4.3.5.5. Bi-directional variable

2 variables are bi-directional if their instances always point to each other (unless they point to null). So if A references B, then B references A.



To map a bi-directional relationship between 2 planning variables, annotate the master side as a normal (= genuine) planning variable:

```
@PlanningEntity
public class Customer ... {

    @PlanningVariable(chained = true, ...)
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }

    public void setPreviousStandstill(Standstill previousStandstill) {...}

}
```

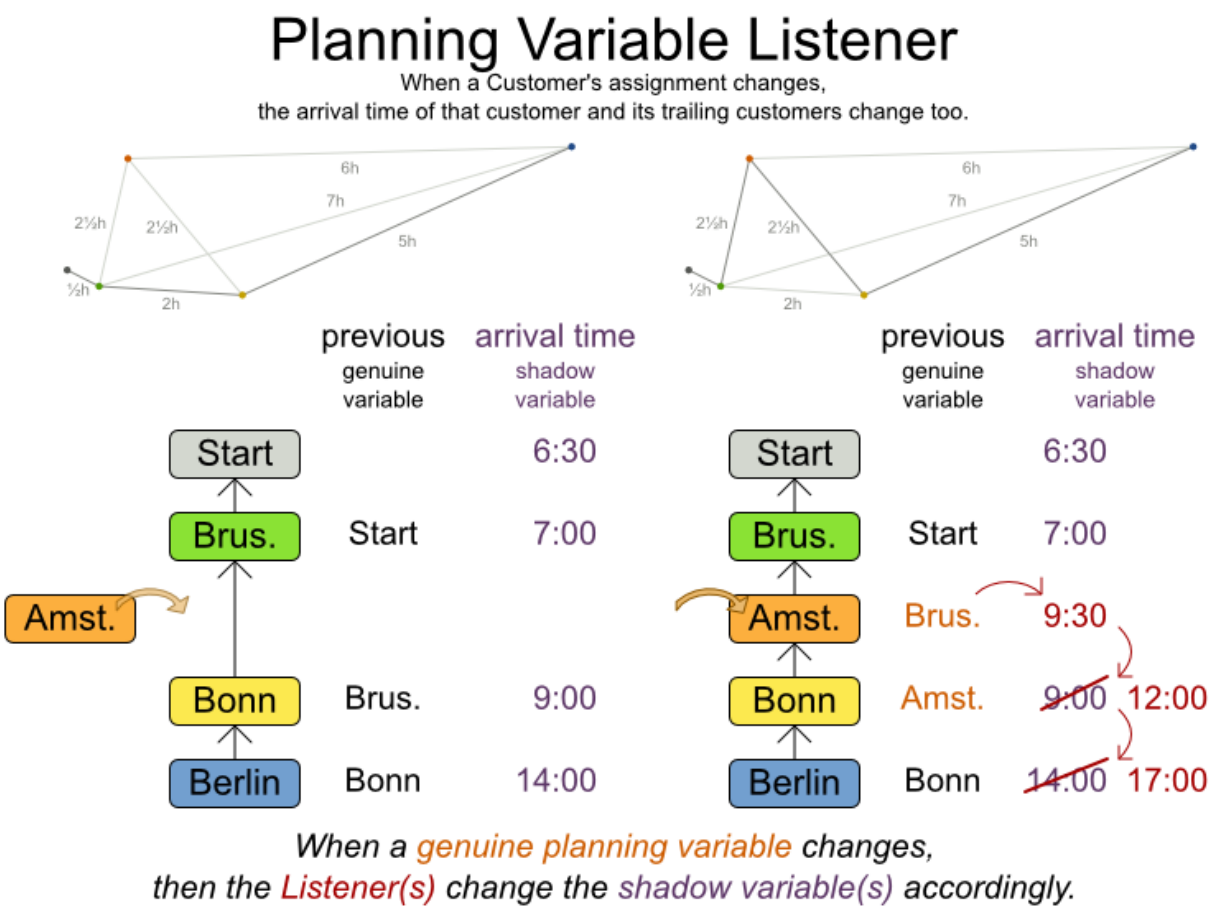
And then annotate the other side as a `@PlanningVariable` with only a `mappedBy` annotation (and no `valueRangeProviderRefs`).

```
@PlanningEntity
public interface Standstill {

    @PlanningVariable(mappedBy = "previousStandstill")
    Customer getNextCustomer();
    void setNextCustomer(Customer nextCustomer);

}
```

The `mappedBy` variable is a form of a shadow variable: Planner uses a build-in `PlanningVariableListener` to update its state.

4.3.5.6. Planning variable listener that updates shadow variables

A shadow variable is a variables who's correct value can be deduced from the state of the genuine planning variables. Even though such a variable violates the principle of normalization by definition, in some use cases it can be very practical to use a shadow variable. For example in vehicle routing with time windows: the arrival time at a customer for a vehicle can be calculated based on the previously visited customers of that vehicle (and the known travel times between 2 locations).



To use custom `PlanningVariableListener`, implement the interface and annotate it on the genuine planning variable(s) that trigger changes in the shadow variable(s):

```
    @PlanningVariable(..., variableListenerClasses = {VehicleUpdatingVariableListener.class, An
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }
```

For example, the VehicleUpdatingVariableListener assures that every `Customer` in a chain has the same `Vehicle`, namely the chain's anchor.

```
public class VehicleUpdatingVariableListener implements PlanningVariableListener<Customer> {

   public void afterEntityAdded(ScoreDirector scoreDirector, Customer customer) {
        updateVehicle(scoreDirector, customer);
    }

    public void afterVariableChanged(ScoreDirector scoreDirector, Customer customer) {
```

```
        updateVehicle(scoreDirector, customer);
    }

    ...

    protected void updateVehicle(ScoreDirector scoreDirector, Customer sourceCustomer) {
        Standstill previousStandstill = sourceCustomer.getPreviousStandstill();
        Vehicle vehicle = previousStandstill == null ? null : previousStandstill.getVehicle();
        Customer shadowCustomer = sourceCustomer;
      while (shadowCustomer != null && shadowCustomer.getVehicle() != vehicle) {
            scoreDirector.beforeVariableChanged(shadowCustomer, "vehicle");
            shadowCustomer.setVehicle(vehicle);
            scoreDirector.afterVariableChanged(shadowCustomer, "vehicle");
            shadowCustomer = shadowCustomer.getNextCustomer();
        }
    }

}
```

Any class that has a shadow variable, is a planning entity class, even it has no genuine planning variables.

Warning

A PlanningVariableListener can only change shadow variables. It must never change a genuine planning variable or a problem fact.

Warning

Any change of a shadow variable must be told to the `ScoreDirector`.

From a score calculation perspective, a shadow variable is like any other planning variable. From an optimization perspective, Planner effectively only optimizes the genuine variables (and mostly ignores the shadow variables): it just assures that when a genuine variable changes, it changes any dependent shadow variables accordingly.

4.3.6. Planning problem and planning solution

4.3.6.1. Planning problem instance

A dataset for a planning problem needs to be wrapped in a class for the `Solver` to solve. You must implement this class. For example in n queens, this in the `NQueens` class which contains a `Column` list, a `Row` list and a `Queen` list.

A planning problem is actually a unsolved planning solution or - stated differently - an uninitialized `Solution`. Therefor, that wrapping class must implement the `Solution` interface. For example in n queens, that `NQueens` class implements `Solution`, yet every `Queen` in a fresh `NQueens` class is not yet assigned to a `Row` (their `row` property is `null`). So it's not a feasible solution. It's not even a possible solution. It's an uninitialized solution.

4.3.6.2. The `Solution` interface

You need to present the problem as a `Solution` instance to the `Solver`. So you need to have a class that implements the `Solution` interface:

```
public interface Solution<S extends Score> {

    S getScore();
    void setScore(S score);

    Collection<? extends Object> getProblemFacts();

}
```

For example, an `NQueens` instance holds a list of all columns, all rows and all `Queen` instances:

```
public class NQueens implements Solution<SimpleScore> {

    private int n;

    // Problem facts
    private List<Column> columnList;
    private List<Row> rowList;

    // Planning entities
    private List<Queen> queenList;

    // ...
}
```

4.3.6.3. The `getScore()` and `setScore()` methods

A `Solution` requires a score property. The score property is `null` if the `Solution` is uninitialized or if the score has not yet been (re)calculated. The `score` property is usually typed to the specific `Score` implementation you use. For example, `NQueens` uses a `SimpleScore`:

```
public class NQueens implements Solution<SimpleScore> {
```

```
    private SimpleScore score;

    public SimpleScore getScore() {
        return score;
    }

    public void setScore(SimpleScore score) {
        this.score = score;
    }

    // ...
}
```

Most use cases use a `HardSoftScore` instead:

```
public class CourseSchedule implements Solution<HardSoftScore> {

    private HardSoftScore score;

    public HardSoftScore getScore() {
        return score;
    }

    public void setScore(HardSoftScore score) {
        this.score = score;
    }

    // ...
}
```

See the Score calculation section for more information on the `Score` implementations.

4.3.6.4. The `getProblemFacts()` method

The method is only used if Drools is used for score calculation. Other score directors do not use it.

All objects returned by the `getProblemFacts()` method will be asserted into the Drools working memory, so the score rules can access them. For example, `NQueens` just returns all `Column` and `Row` instances.

```
    public Collection<? extends Object> getProblemFacts() {
        List<Object> facts = new ArrayList<Object>();
        facts.addAll(columnList);
        facts.addAll(rowList);
         // Do not add the planning entity's (queenList) because that will be
 done automatically
```

```
        return facts;
    }
```

All planning entities are automatically inserted into the Drools working memory. Do not add them in the method `getProblemFacts()`.

> **Note**
>
> A common mistake is to use `facts.add(...)` instead of `fact.addAll(...)` for a `Collection`, which leads to score rules failing to match because the elements of that `Collection` aren't in the Drools working memory.

The method `getProblemFacts()` is not called much: at most only once per solver phase per solver thread.

4.3.6.5. Cached problem fact

A cached problem fact is a problem fact that doesn't exist in the real domain model, but is calculated before the `Solver` really starts solving. The method `getProblemFacts()` has the chance to enrich the domain model with such cached problem facts, which can lead to simpler and faster score constraints.

For example in examination, a cached problem fact `TopicConflict` is created for every 2 `Topic`'s which share at least 1 `Student`.

```
    public Collection<? extends Object> getProblemFacts() {
        List<Object> facts = new ArrayList<Object>();
        // ...
        facts.addAll(calculateTopicConflictList());
        // ...
        return facts;
    }

    private List<TopicConflict> calculateTopicConflictList() {
        List<TopicConflict> topicConflictList = new ArrayList<TopicConflict>();
        for (Topic leftTopic : topicList) {
            for (Topic rightTopic : topicList) {
                if (leftTopic.getId() < rightTopic.getId()) {
                    int studentSize = 0;
                    for (Student student : leftTopic.getStudentList()) {
                        if (rightTopic.getStudentList().contains(student)) {
                            studentSize++;
                        }
                    }
                    if (studentSize > 0) {
```

```
                        topicConflictList.add(new TopicConflict(leftTopic, rightTopic, studentS
                    }
                }
            }
        }
        return topicConflictList;
    }
```

Any score constraint that needs to check if no 2 exams have a topic which share a student are being scheduled close together (depending on the constraint: at the same time, in a row or in the same day), can simply use the `TopicConflict` instance as a problem fact, instead of having to combine every 2 `Student` instances.

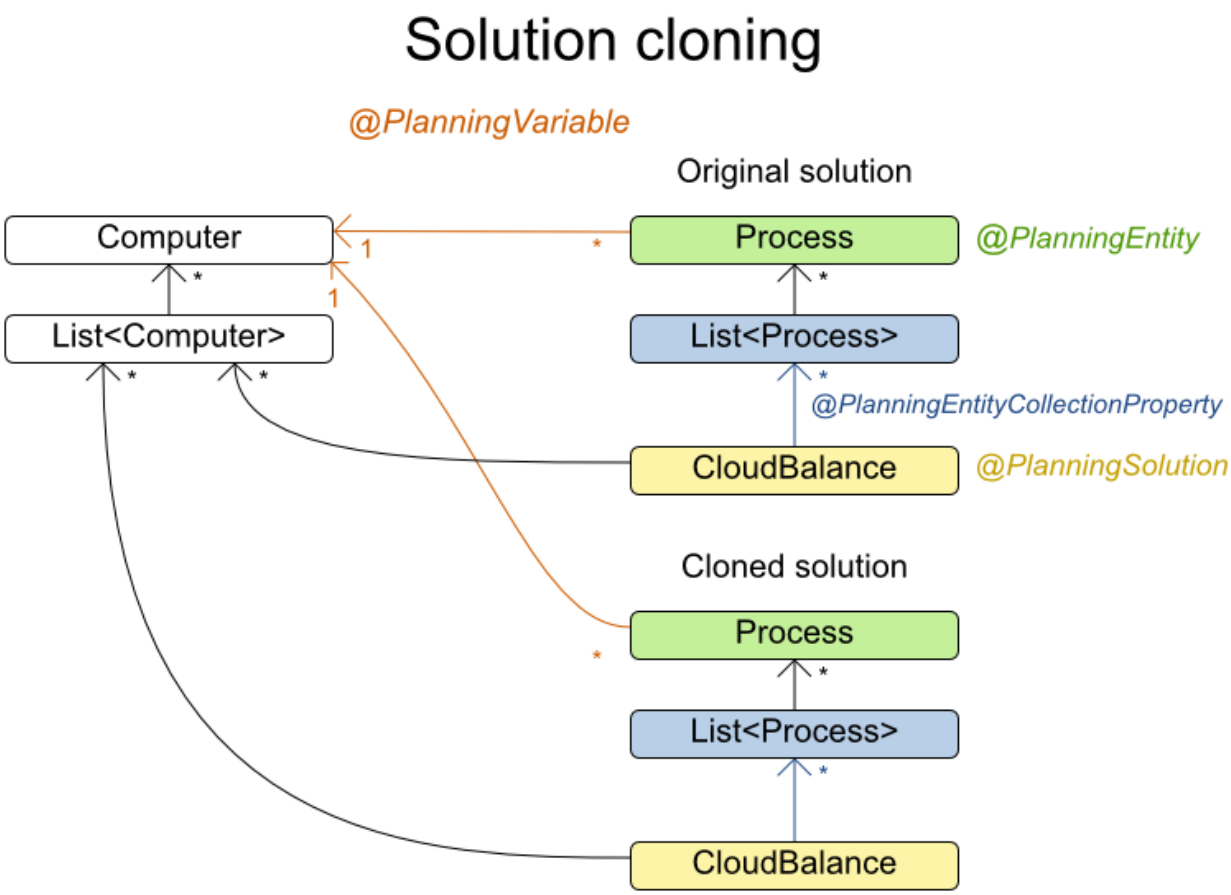
4.3.6.6. Cloning a `Solution`

Most (if not all) optimization algorithms clone the solution each time they encounter a new best solution (so they can recall it later) or to work with multiple solutions in parallel.

> **Note**
>
> There are many ways to clone, such as a shallow clone, deep clone, ... This context focuses on *a planning clone*.

A planning clone of a `Solution` must fulfill these requirements:

- The clone must represent the same planning problem. Usually it reuses the same instances of the problem facts and problem fact collections as the original.
- The clone must use different, cloned instances of the entities and entity collections. Changes to an original `Solution`'s entity's variables must not effect its clone.



Implementing a planning clone method is hard, therefore you don't need to implement it.

4.3.6.6.1. `FieldAccessingSolutionCloner`

This `SolutionCloner` is used by default. It works for the majority of use cases.

> **Warning**
>
> When the `FieldAccessingSolutionCloner` clones your entity collection, it might not recognize the implementation and replace it with `ArrayList`, `LinkedHashSet` or `TreeSet` (whichever is more applicable). It recognizes most of the common JDK `Collection` implementations.

4.3.6.6.2. Custom cloning: Make `Solution` implement `PlanningCloneable`

If your Solution implements PlanningCloneable, Planner will automatically choose to clone it by calling the method `planningClone()`.

```
public interface PlanningCloneable<T> {

    T planningClone();

}
```

For example: If `NQueens` implements `PlanningCloneable`, it would only deep clone all `Queen` instances. When the original solution is changed during planning, by changing a `Queen`, the clone stays the same.

```
public class NQueens implements Solution<...>, PlanningCloneable<NQueens> {
    ...

    /**
     * Clone will only deep copy the {@link #queenList}.
     */
    public NQueens planningClone() {
        NQueens clone = new NQueens();
        clone.id = id;
        clone.n = n;
        clone.columnList = columnList;
        clone.rowList = rowList;
        List<Queen> clonedQueenList = new ArrayList<Queen>(queenList.size());
        for (Queen queen : queenList) {
            clonedQueenList.add(queen.planningClone());
        }
        clone.queenList = clonedQueenList;
        clone.score = score;
        return clone;
    }
}
```

The `planningClone()` method should only deep clone the planning entities. Notice that the problem facts, such as `Column` and `Row` are normally *not* cloned: even their `List` instances are *not* cloned. If you were to clone the problem facts too, then you'd have to make sure that the new planning entity clones also refer to the new problem facts clones used by the solution. For example, if you would clone all `Row` instances, then each `Queen` clone and the `NQueens` clone itself should refer to those new `Row` clones.

Warning

Cloning an entity with a *chained* variable is devious: a variable of an entity A might point to another entity B. If A is cloned, then it's variable must point to the clone of B, not the original B.

4.3.6.7. Build an uninitialized solution

Build a `Solution` instance to represent your planning problem, so you can set it on the `Solver` as the planning problem to solve. For example in n queens, an `NQueens` instance is created with the required `Column` and `Row` instances and every `Queen` set to a different `column` and every `row` set to `null`.

```
private NQueens createNQueens(int n) {
    NQueens nQueens = new NQueens();
    nQueens.setId(0L);
    nQueens.setN(n);
    nQueens.setColumnList(createColumnList(nQueens));
    nQueens.setRowList(createRowList(nQueens));
    nQueens.setQueenList(createQueenList(nQueens));
    return nQueens;
}

private List<Queen> createQueenList(NQueens nQueens) {
    int n = nQueens.getN();
    List<Queen> queenList = new ArrayList<Queen>(n);
    long id = 0;
    for (Column column : nQueens.getColumnList()) {
        Queen queen = new Queen();
        queen.setId(id);
        id++;
        queen.setColumn(column);
        // Notice that we leave the PlanningVariable properties on null
        queenList.add(queen);
    }
    return queenList;
}
```

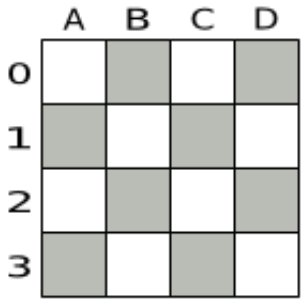


Figure 4.1. Uninitialized solution for the 4 queens puzzle

Usually, most of this data comes from your data layer, and your `Solution` implementation just aggregates that data and creates the uninitialized planning entity instances to plan:

```
        private void createLectureList(CourseSchedule schedule) {
            List<Course> courseList = schedule.getCourseList();
          List<Lecture> lectureList = new ArrayList<Lecture>(courseList.size());
            for (Course course : courseList) {
                for (int i = 0; i < course.getLectureSize(); i++) {
                    Lecture lecture = new Lecture();
                    lecture.setCourse(course);
                    lecture.setLectureIndexInCourse(i);
                       // Notice that we leave the PlanningVariable properties
 (period and room) on null
                    lectureList.add(lecture);
                }
            }
            schedule.setLectureList(lectureList);
        }
```

4.4. Use the `Solver`

4.4.1. The Solver interface

A `Solver` implementation will solve your planning problem.

```
public interface Solver {

    void setPlanningProblem(Solution planningProblem);

    void solve();

    Solution getBestSolution();
```

```
    // ...

}
```

A `Solver` can only solve 1 planning problem instance at a time. A `Solver` should only be accessed from a single thread, except for the methods that are specifically javadocced as being thread-safe. It's build with a `SolverFactory`, do not implement or build it yourself.

4.4.2. Solving a problem

Solving a problem is quite easy once you have:

- A `Solver` build from a solver configuration
- A `Solution` that represents the planning problem instance

Just set the planning problem, solve it and extract the best solution:

```
    solver.setPlanningProblem(planningProblem);
    solver.solve();
    Solution bestSolution = solver.getBestSolution();
```

For example in n queens, the method `getBestSolution()` will return an `NQueens` instance with every `Queen` assigned to a `Row`.

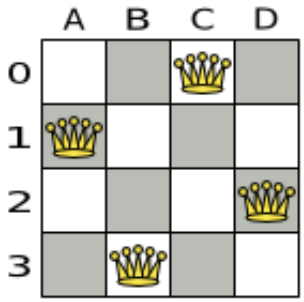


Figure 4.2. Best solution for the 4 queens puzzle in 8 ms (also an optimal solution)

The `solve()` method can take a long time (depending on the problem size and the solver configuration). The `Solver` will remember (actually clone) the best solution it encounters during its solving. Depending on a number factors (including problem size, how much time the `Solver` has, the solver configuration, ...), that best solution will be a feasible or even an optimal solution.

> **Note**
>
> The `Solution` instance given to the method `setPlanningProblem()` will be changed by the `Solver`, but it do not mistake it for the best solution.
>
> The `Solution` instance returned by the method `getBestSolution()` will most likely be a clone of the instance given to the method `setPlanningProblem()`, which means it's a different instance.

> **Note**
>
> The `Solution` instance given to the method `setPlanningProblem()` does not need to be uninitialized. It can be partially or fully initialized, which is likely to be the case in *repeated planning*.

4.4.3. Environment mode: Are there bugs in my code?

The environment mode allows you to detect common bugs in your implementation. It does not affect the logging level.

You can set the environment mode in the solver configuration XML file:

```
<solver>
  <environmentMode>FAST_ASSERT</environmentMode>
  ...
</solver>
```

A solver has a single `Random` instance. Some solver configurations use the `Random` instance a lot more than others. For example simulated annealing depends highly on random numbers, while tabu search only depends on it to deal with score ties. The environment mode influences the seed of that `Random` instance.

There are 4 environment modes:

4.4.3.1. FULL_ASSERT

The FULL_ASSERT mode turns on all assertions (such as assert that the incremental score calculation is uncorrupted for each move) to fail-fast on a bug in a Move implementation, a score rule, the rule engine itself, ...

This mode is reproducible (see the reproducible mode). It is also intrusive because it calls the method `calculateScore()` more frequently than a non assert mode.

The FULL_ASSERT mode is horribly slow (because it doesn't rely on delta based score calculation).

4.4.3.2. NON_INTRUSIVE_FULL_ASSERT

The NON_INTRUSIVE_FULL_ASSERT turns on several assertions to fail-fast on a bug in a Move implementation, a score rule, the rule engine itself, ...

This mode is reproducible (see the reproducible mode). It is non-intrusive because it does not call the method `calculateScore()` more frequently than a non assert mode.

The NON_INTRUSIVE_FULL_ASSERT mode is horribly slow (because it doesn't rely on delta based score calculation).

4.4.3.3. FAST_ASSERT

The FAST_ASSERT mode turns on most assertions (such as assert that an undo Move's score is the same as before the Move) to fail-fast on a bug in a Move implementation, a score rule, the rule engine itself, ...

This mode is reproducible (see the reproducible mode). It is also intrusive because it calls the method `calculateScore()` more frequently than a non assert mode.

The FAST_ASSERT mode is slow.

It's recommended to write a test case which does a short run of your planning problem with the FAST_ASSERT mode on.

4.4.3.4. REPRODUCIBLE (default)

The reproducible mode is the default mode because it is recommended during development. In this mode, 2 runs in the same Planner version will execute the same code in the same order. **Those 2 runs will have the same result**, except if the note below applies. This allows you to consistently reproduce bugs. It also allows you to benchmark certain refactorings (such as a score constraint optimization) fairly across runs.

> **Note**
>
> Despite the reproducible mode, your application might still not be fully reproducible because of:
>
> - Use of `HashSet` (or another `Collection` which has an inconsistent order between JVM runs) for collections of planning entities or planning values (but not normal problem facts), especially in the `Solution` implementation. Replace it with `LinkedHashSet`.
> - Combining a time gradient dependent algorithms (most notably simulated annealing) together with time spend termination. A sufficiently large difference

in allocated CPU time will influence the time gradient values. Replace simulated annealing with late acceptance. Or instead, replace time spend termination with step count termination.

The reproducible mode is slightly slower than the production mode. If your production environment requires reproducibility, use it in production too.

In practice, this mode uses the default random seed, and it also disables certain concurrency optimizations (such as work stealing).

4.4.3.5. PRODUCTION

The production mode is the fastest, but not reproducible. It is recommended for a production environment, unless reproducibility is required.

The random seed is different on every run, which makes it more robust against an unlucky random seed. An unlucky random seed gives a bad result on a certain data set with a certain solver configuration. Note that in most use cases the impact of the random seed is relatively low on the result (even with simulated annealing). An occasional bad result is far more likely to be caused by another issue (such as a *score trap*).

4.4.4. Logging level: What is the `Solver` doing?

The best way to illuminate the black box that is a `Solver`, is to play with the logging level:

- **error**: Log errors, except those that are thrown to the calling code as a `RuntimeException`.

> **Note**
>
> **If an error happens, Planner normally fails fast**: it throws a subclass of `RuntimeException` with a detailed message to the calling code. It does not log it as an error itself to avoid duplicate log messages. Except if the calling code explicitly catches and eats that `RuntimeException`, a `Thread`'s default `ExceptionHandler` will log it as an error anyway. Meanwhile, the code is disrupted from doing further harm or obfuscating the error.

- **warn**: Log suspicious circumstances.
- **info**: Log every phase and the solver itself. See *scope overview*.
- **debug**: Log every step of every phase. See *scope overview*.
- **trace**: Log every move of every step of every phase. See *scope overview*.

> **Note**
>
> Turning on `trace` logging, will slow down performance considerably: it's often 4 times slower. However, it's invaluable during development to discover a bottleneck.
>
> Even debug logging can slow down performance considerably for fast stepping algorithms (such as Late Acceptance and Simulated Annealing), but not for slow stepping algorithms (such as Tabu Search).

For example, set it to `debug` logging, to see when the phases end and how fast steps are taken:

```
INFO  Solving started: time spend (0), score (null), new best score (null),
 random seed (0).
DEBUG    Step index (0), time spend (1), score (0), initialized planning entity
 (col2@row0).
DEBUG    Step index (1), time spend (3), score (0), initialized planning entity
 (col1@row2).
DEBUG    Step index (2), time spend (4), score (0), initialized planning entity
 (col3@row3).
DEBUG      Step index (3), time spend (5), score (-1), initialized planning
 entity (col0@row1).
INFO  Phase (0) constructionHeuristic ended: step total (4), time spend (6),
 best score (-1).
DEBUG     Step index (0), time spend (10), score (-1),      best score (-1),
 accepted/selected move count (12/12) for picked step (col1@row2 => row3).
DEBUG      Step index (1), time spend (12), score (0), new best score (0),
 accepted/selected move count (12/12) for picked step (col3@row3 => row2).
INFO Phase (1) localSearch ended: step total (2), time spend (13), best score (0).
INFO  Solving ended: time spend (13), best score (0), average calculate count
 per second (4846).
```

All time spends are in milliseconds.

Everything is logged to *SLF4J* [http://www.slf4j.org/], which is a simple logging facade which delegates every log message to Logback, Apache Commons Logging, Log4j or java.util.logging. Add a dependency to the logging adaptor for your logging framework of choice.

If you're not using any logging framework yet, use Logback by adding this Maven dependency (there is no need to add an extra bridge dependency):

```
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
```

```
      <version>1.x</version>
    </dependency>
```

Configure the logging level on the package `org.optaplanner` in your `logback.xml` file:

```
<configuration>

  <logger name="org.optaplanner" level="debug"/>

  ...

<configuration>
```

If instead, you're still using Log4J (and you don't want to switch to its faster successor, Logback), add the bridge dependency:

```
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.x</version>
    </dependency>
```

And configure the logging level on the package `org.optaplanner` in your `log4j.xml` file:

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <category name="org.optaplanner">
    <priority value="debug" />
  </category>

  ...

</log4j:configuration>
```

Chapter 5. Score calculation

5.1. Score terminology

5.1.1. What is a score?

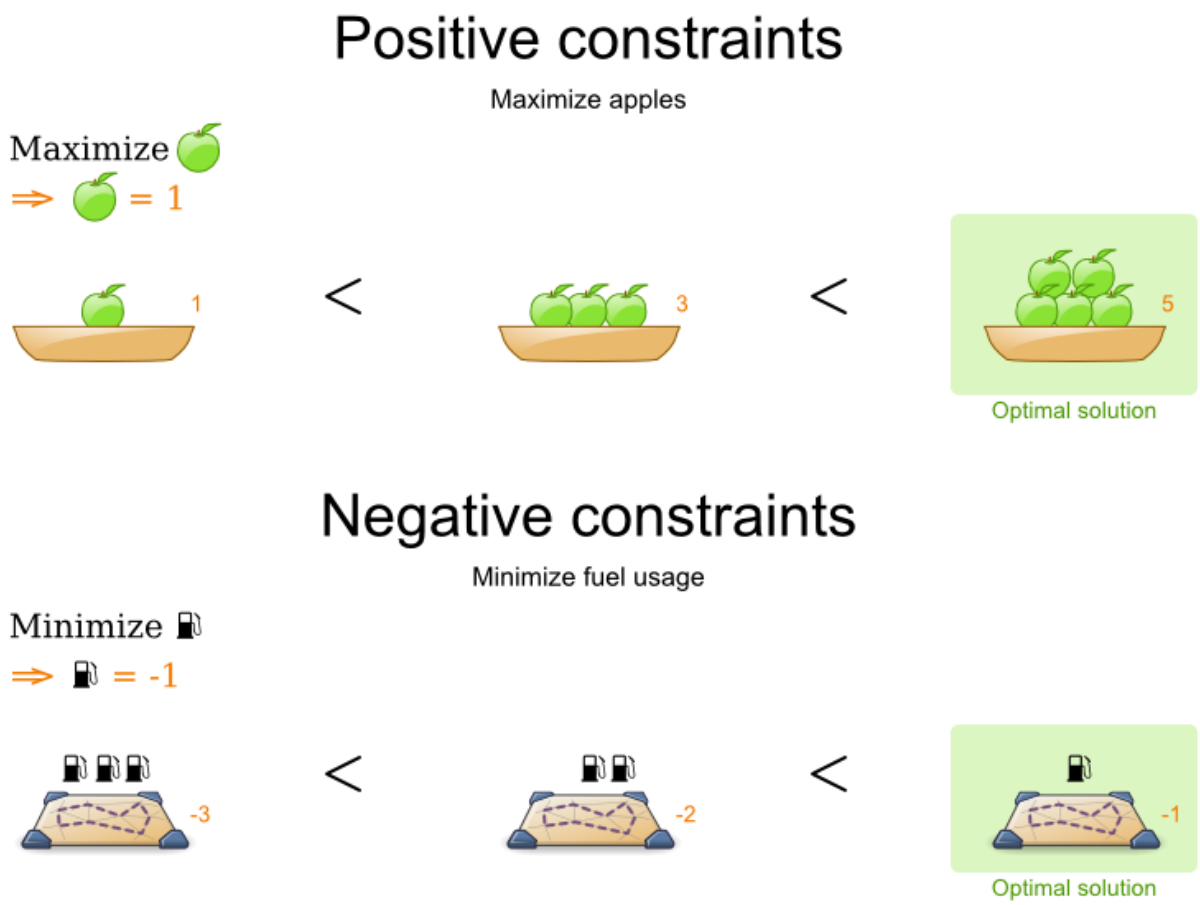
Every initialized `Solution` has a score. That score is an objective way to compare 2 solutions: the solution with the higher score is better. The `Solver` aims to find the `Solution` with the highest `Score` of all possible solutions. The *best solution* is the `Solution` with the highest `Score` that `Solver` has encountered during solving, which might be the *optimal solution*.

Planner cannot automatically know which `Solution` is best for your business, so you need to tell it how to calculate the score of a given `Solution` according to your business needs. There are multiple score techniques that you can use and combine:

- Maximize or minimize a constraint: score constraint signum (positive or negative)
- Put a cost/profit on constraints: score constraint weight
- Prioritize constraints: score level
- Pareto scoring

5.1.2. Score constraint signum (positive or negative)

All score techniques are based on constraints. Such a constraint can be a simple pattern (such as *Maximize the apple harvest in the solution*) or a more complex pattern. A positive constraint is a constraint you're trying to maximize. A negative constraint is a constraint you're trying to minimize.



Notice in the image above, that the optimal solution always has the highest score, regardless if the constraints are positive or negative.

Most planning problems have only negative constraints and therefore have a negative score. In that case, the score is usually the sum of the weight of the negative constraints being broken, with a perfect score of 0. This explains why the score of a solution of 4 queens is the negative (and not the positive!) of the number of queen couples which can attack each other.

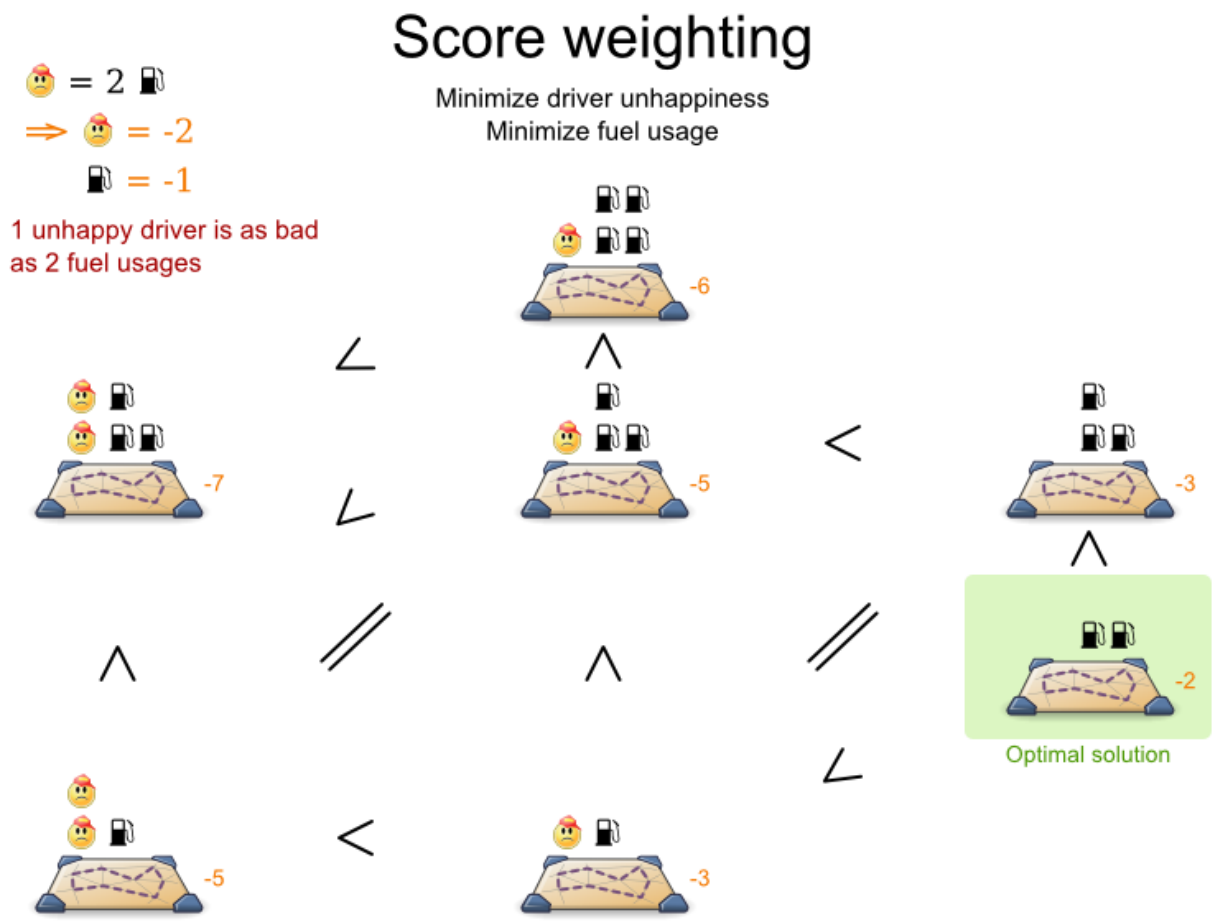
Negative and positive constraints can be combined, even in the same score level.

> **Note**
>
> Don't presume your business knows all its score constraints in advance. Expect score constraints to be added or changed after the first releases.

When a constraint activates (because the negative constraint is broken or the positive constraint is fulfilled) on a certain planning entity set, it is called a *constraint match*.

5.1.3. Score constraint weight

Not all score constraints are equally important. If breaking one constraint is equally bad as breaking another constraint x times, then those 2 constraints have a different weight (but they are in the same score level). For example in vehicle routing, you can make 2 "unhappy driver" constraint matches count as much as 1 "fuel tank usage" constraint match:



Score weighting is often used in use cases where you can put a price tag on everything. In that case, the positive constraints maximize revenue and the negative constraints minimize expenses: together they maximize profit. Alternatively, score weighting is also often used to create social fairness. For example: nurses that request a free day on New Year's eve pay a higher weight than on a normal day.

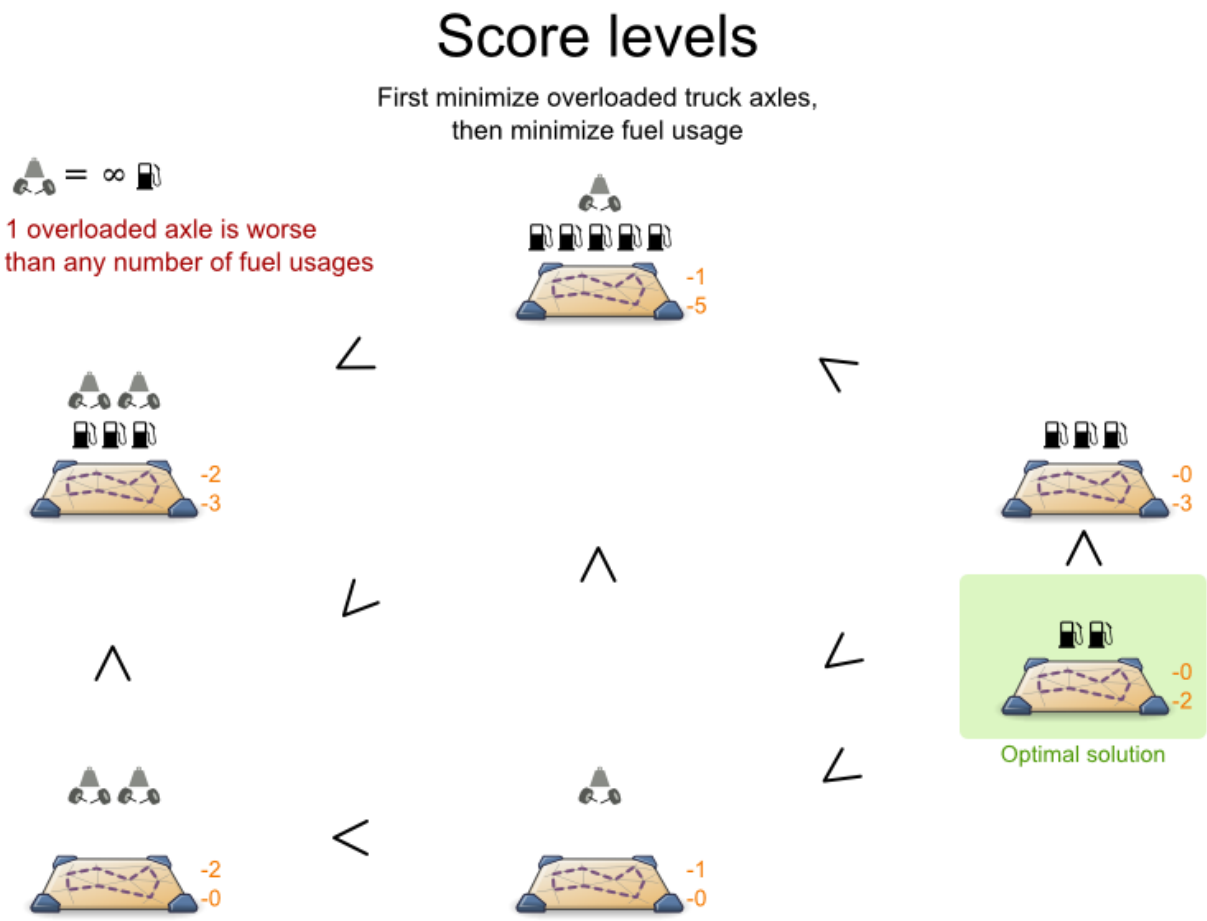
The weight of a constraint match can be dynamically based on the planning entities involved. For example in cloud balance: the weight of the soft constraint match for an active `Computer` is the `cost` of that `Computer`.

5.1.4. Score level

Sometimes a score constraint outranks another score constraint, no matter how many times the other is broken. In that case, those score constraints are in different levels. For example: a nurse

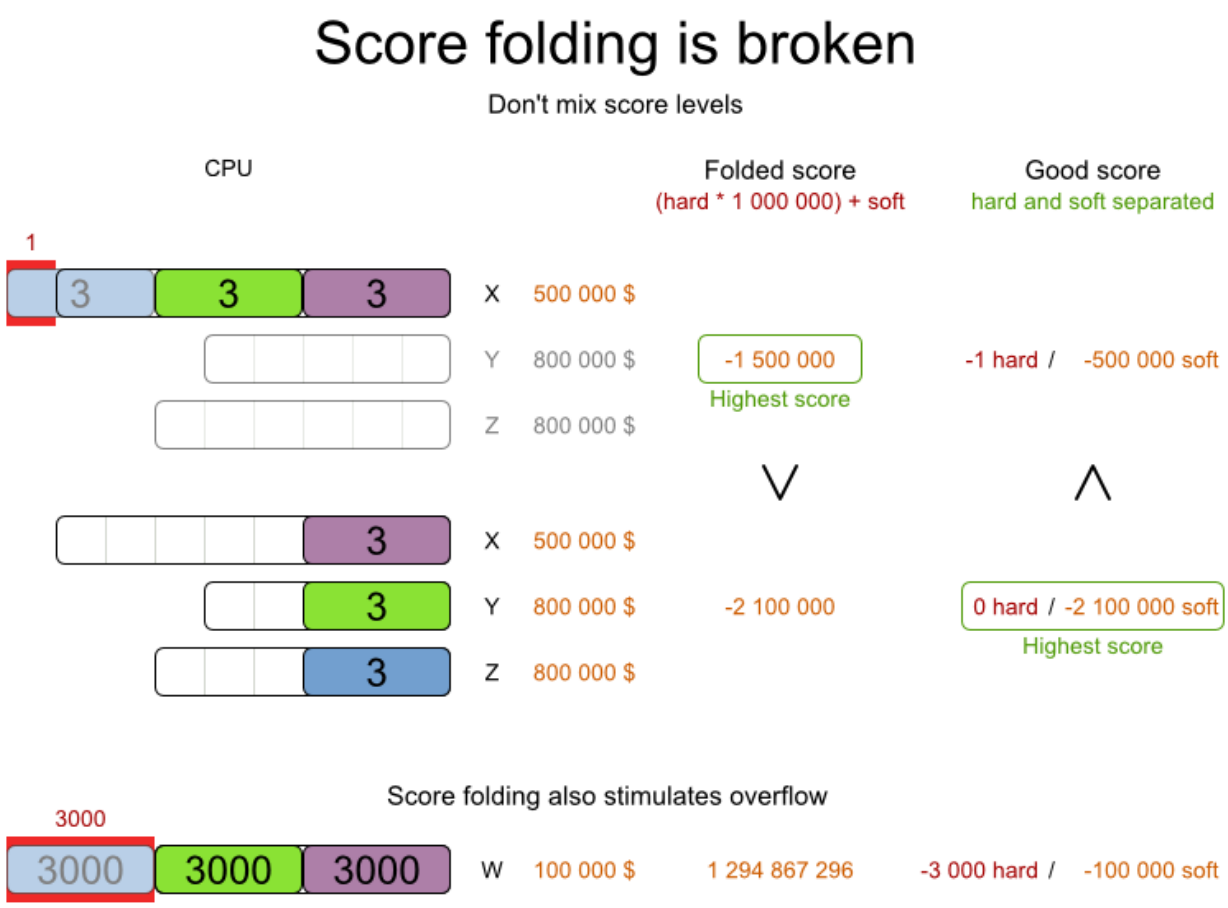
cannot do 2 shifts at the same time (due to the constraints of physical reality), this outranks all nurse happiness constraints.

Most use cases have only 2 score levels: hard and soft. When comparing 2 scores, they are compared lexicographically: the first score level gets compared first. If those differ, the others score levels are ignored. For example: a score that breaks 0 hard constraints and 1000000 soft constraints is better than a score that breaks 1 hard constraint and 0 soft constraints.



Score levels often employ score weighting per level. In such case, the hard constraint level usually makes the solution feasible and the soft constraint level maximizes profit by weighting the constraints on price.

Don't use a big constraint weight when your business actually wants different score levels. That hack, known as *score folding*, is broken:



> **Note**
>
> Your business will probably tell you that your hard constraints all have the same weight, because they cannot be broken (so their weight does not matter). This is not true and it could create a *score trap*. For example in cloud balance: if a `Computer` has 7 CPU too little for its `Processes`, then it must be weighted 7 times as much as if it had only 1 CPU too little. This way, there is an incentive to move a `Process` with 6 CPU or less away from that Computer.

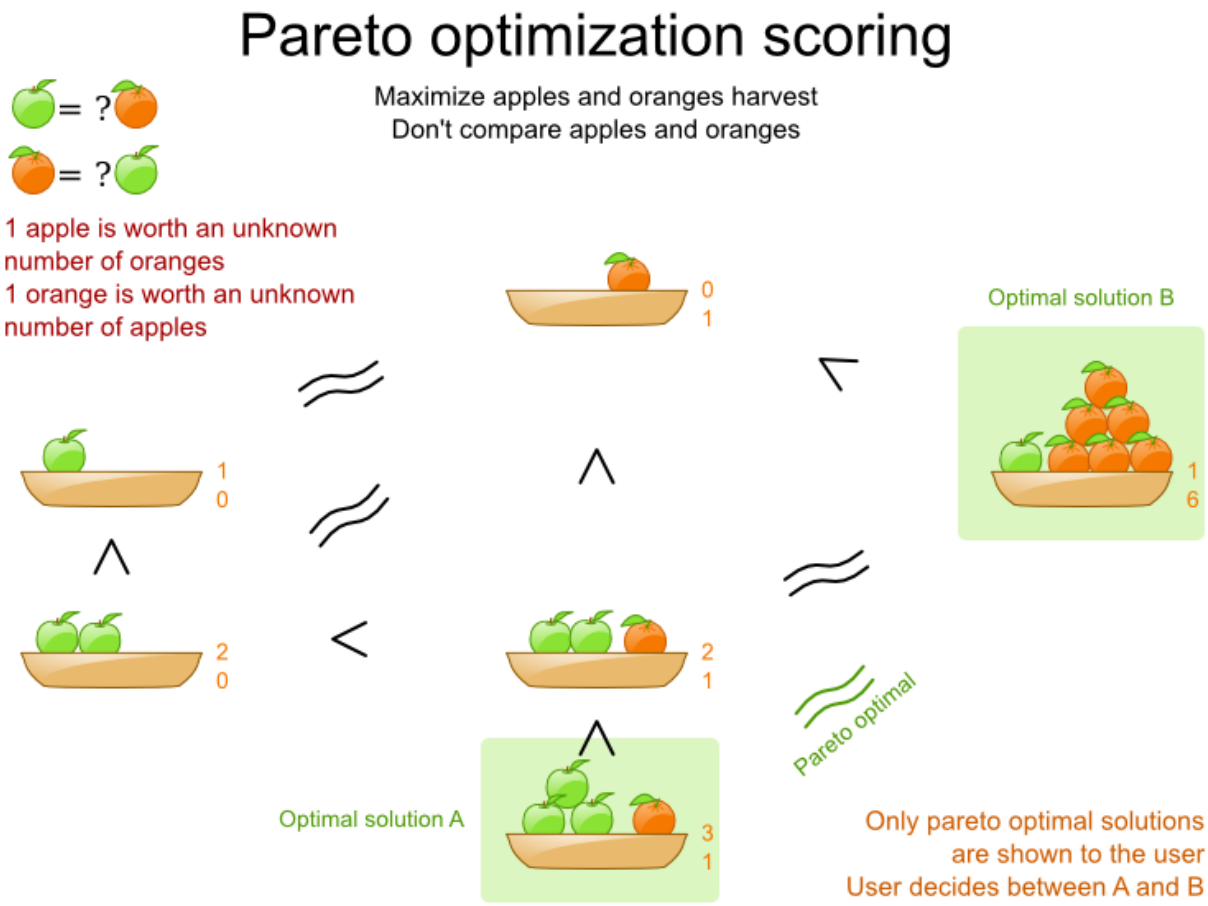
3 or more score levels is supported. For example: a company might decide that profit outranks employee satisfaction (or visa versa), while both are outranked by the constraints of physical reality.

5.1.5. Pareto scoring (AKA multi-objective optimization scoring)

Far less common is the use case of pareto optimization, which is also known under the more confusing term multi-objective optimization. In pareto scoring, score constraints are in the same score level, yet they are not weighted against each other. When 2 scores are compared, each

of the score constraints are compared individually and the score with the most dominating score constraints wins. Pareto scoring can even be combined with score levels and score constraint weighting.

Consider this example with positive constraints, where we want to get the most apples and oranges. Since it's impossible to compare apples and oranges, we can't weight them against each other. Yet, despite that we can't compare them, we can state that 2 apples are better then 1 apple. Similarly, we can state that 2 apples and 1 orange are better than just 1 orange. So despite our inability to compare some Scores conclusively (at which point we declare them equal), we can find a set of optimal scores. Those are called pareto optimal.



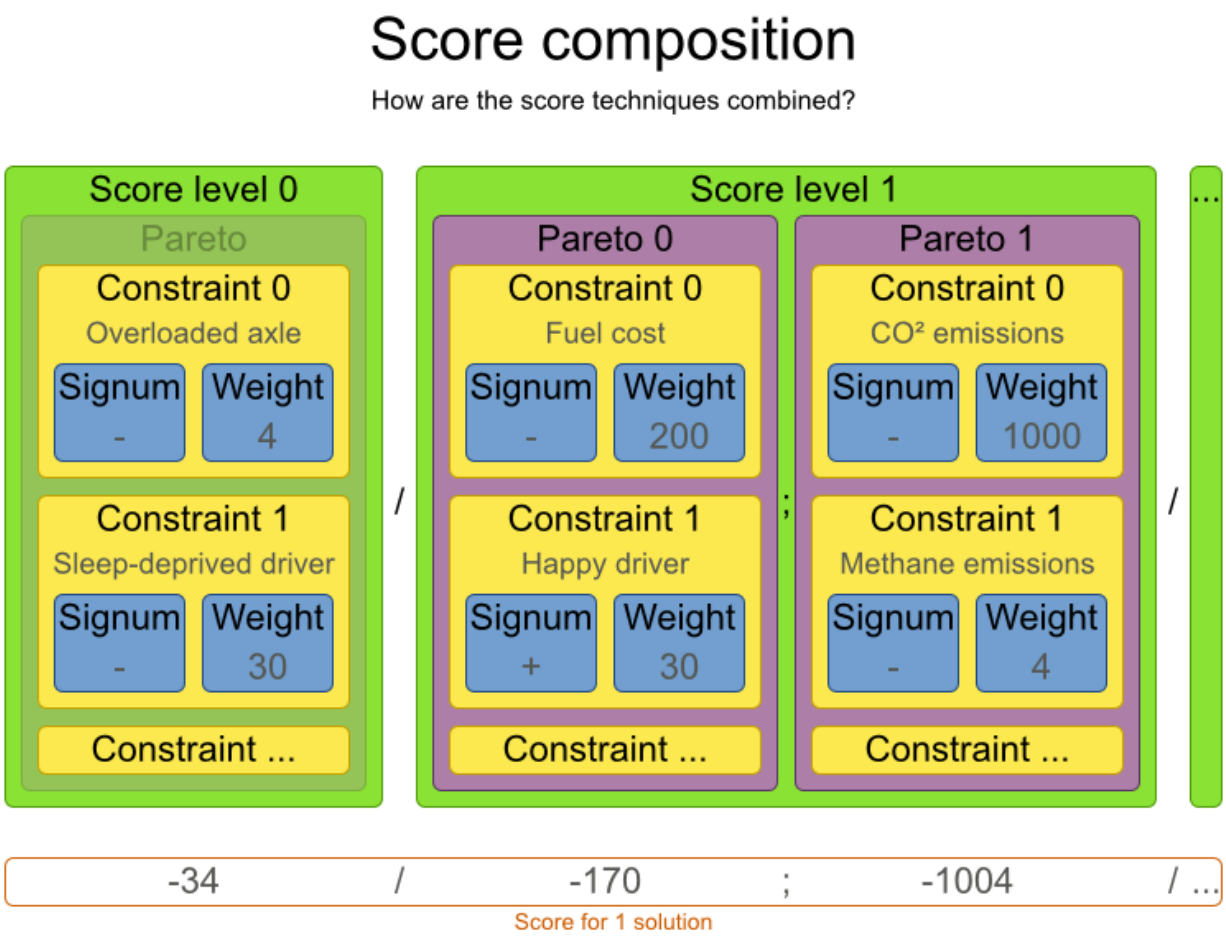
Scores are considered equal far more often. It's left up to a human to choose the better out of a set of best solutions (with equal scores) found by Planner. In the example above, the user must choose between solution A (3 apples and 1 orange) and solution B (1 apples and 6 oranges). It's guaranteed that Planner has not found another solution which has more apples or more oranges or even a better combination of both (such as 2 apples and 3 oranges).

To implement pareto scoring in Planner, *implement a custom `ScoreDefinition` and `Score`* (and replace the `BestSolutionRecaller`). Future versions will provide out-of-the-box support.

> **Note**
>
> A pareto `Score`'s method `compareTo` is not transitive because it does a pareto comparison. For example: 2 apples is greater than 1 apple. 1 apples is equal to 1 orange. Yet, 2 apples are not greater than 1 orange (but actually equal). Pareto comparison violates the contract of the interface `java.lang.Comparable`'s method `compareTo`, but Planner's systems are *pareto comparison safe*, unless explicitly stated otherwise in this documentation.

5.1.6. Combining score techniques

All the score techniques mentioned above, can be combined seamlessly:



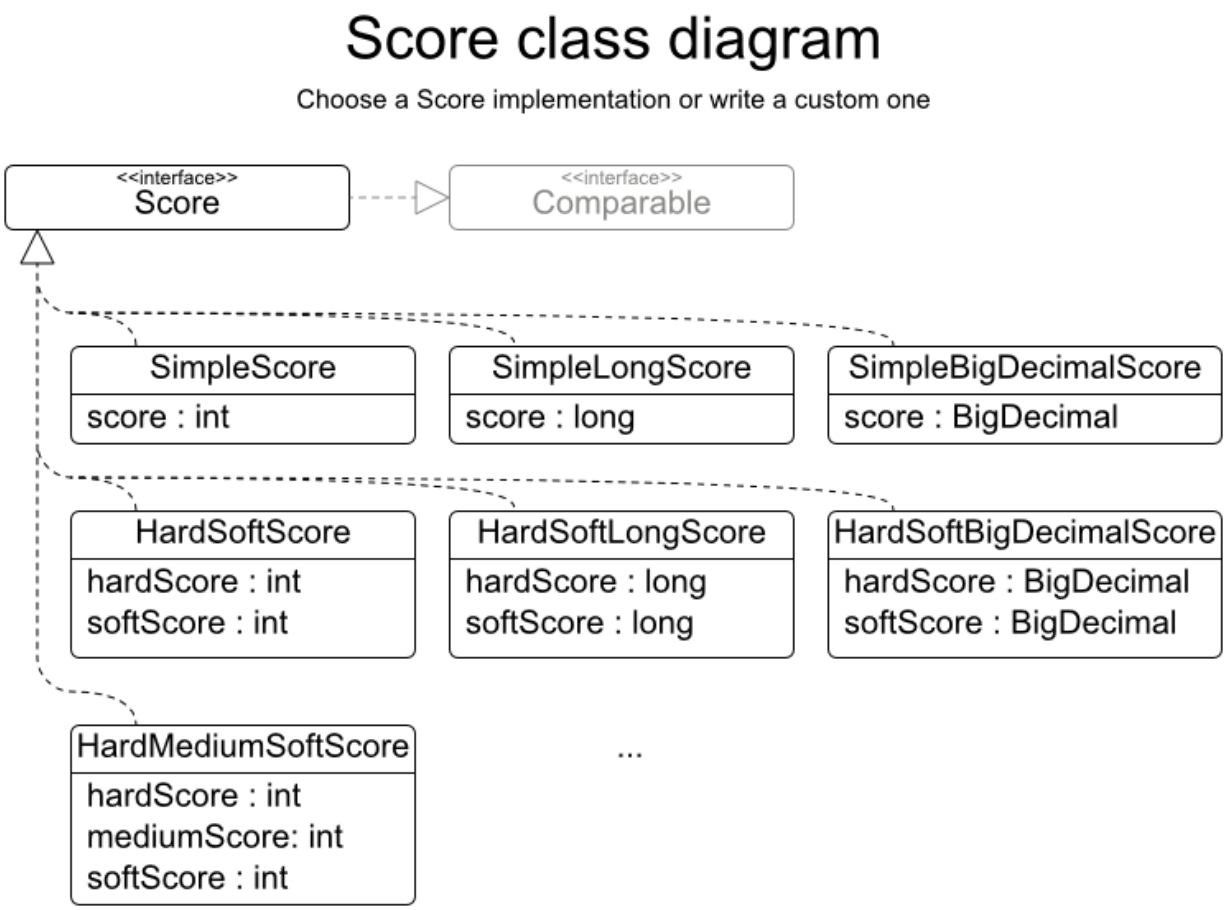
5.1.7. The `Score` interface

A score is represented by the `Score` interface, which naturally extends `Comparable`:

```
public interface Score<...> extends Comparable<...> {
```

```
    ...
}
```

The `Score` implementation to use depends on your use case. Your score might not efficiently fit in a single `long` value. Planner has several build-in `Score` implementations, but you can implement a custom `Score` too. Most use cases tend to use the build-in `HardSoftScore`.



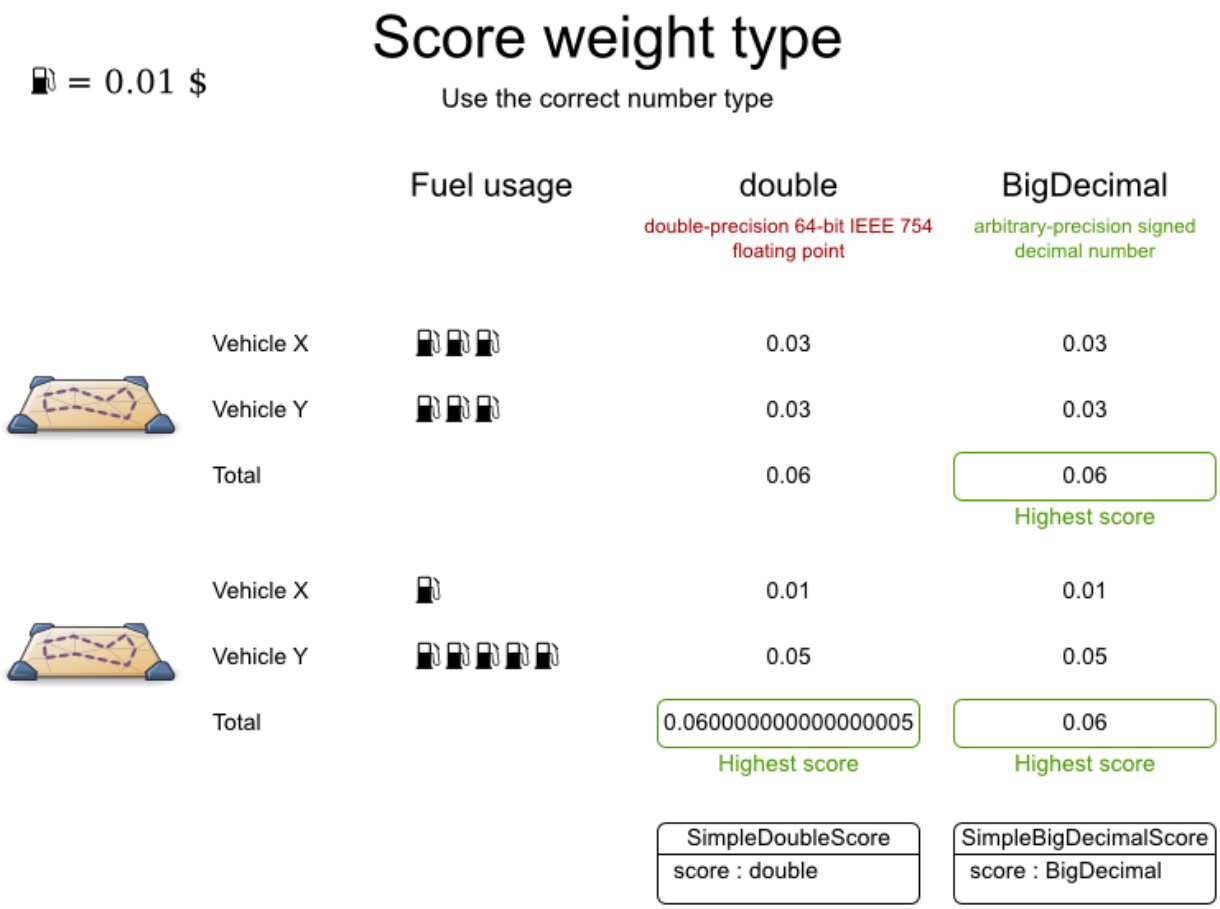
The `Score` implementation (for example `HardSoftScore`) must be the same throughout a `Solver` runtime. The `Score` implementation is configured in the solver configuration as a ScoreDefinition:

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

5.1.8. Avoid floating point numbers in score calculation

Avoid the use of `float` and `double` for score calculation. Use `BigDecimal` instead.

Floating point numbers (`float` and `double`) cannot represent a decimal number correctly. For example: a `double` cannot hold the value `0.05` correctly. Instead, it holds the nearest representable value. Arithmetic (including addition and subtraction) with floating point numbers, especially for planning problems, leads to incorrect decisions:



Additionally, floating point number addition is not associative:

```
System.out.println( ((0.01 + 0.02) + 0.03) == (0.01 + (0.02 + 0.03)) ); //
// returns false
```

This leads to *score corruption*.

Decimal numbers (`BigDecimal`) have none of these problems.

> **Note**
>
> BigDecimal arithmetic is considerably slower than `int`, `long` or `double` arithmetic. In experiments we've seen the average calculation count get divided by 5.

> Therefore, in some cases, it can be worthwhile to multiply *all* numbers for a single score weight by a plural of ten (for example `1000`), so the score weight fits in an `int` or `long`.

5.2. Choose a Score definition

Each `Score` implementation also has a `ScoreDefinition` implementation. For example: `SimpleScore` is definied by `SimpleScoreDefinition`.

5.2.1. SimpleScore

A `SimpleScore` has a single `int` value, for example `-123`. It has a single score level.

```
<scoreDirectorFactory>
  <scoreDefinitionType>SIMPLE</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

Variants of this `scoreDefinitionType`:

- `SIMPLE_LONG`: Uses `SimpleLongScore` which has a `long` value instead of an `int` value.
- `SIMPLE_DOUBLE`: Uses `SimpleDoubleScore` which has a `double` value instead of an `int` value. *Not recommended to use.*
- `SIMPLE_BIG_DECIMAL`: Uses `SimpleBigDecimalScore` which has a `BigDecimal` value instead of an `int` value.

5.2.2. HardSoftScore (recommended)

A `HardSoftScore` has a hard `int` value and a soft `int` value, for example `-123hard/-456soft`. It has 2 score levels (hard and soft).

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

Variants of this `scoreDefinitionType`:

- `HARD_SOFT_LONG`: Uses `HardSoftLongScore` which has `long` values instead of `int` values.

- `HARD_SOFT_DOUBLE`: Uses `HardSoftDoubleScore` which has `double` values instead of `int` values. *Not recommended to use.*
- `HARD_SOFT_BIG_DECIMAL`: Uses `HardSoftBigDecimalScore` which has `BigDecimal` values instead of `int` values..

5.2.3. HardMediumSoftScore

A `HardMediumSoftScore` which has a hard `int` value, a medium `int` value and a soft `int` value, for example `-123hard/-456medium/-789soft`. It has 3 score levels (hard, medium and soft).

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_MEDIUM_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

5.2.4. BendableScore

A `BendableScore` has a configurable number of score levels. It has an array of hard `int` values and an array of soft `int` value, for example 2 hard levels and 3 soft levels for a score `-123/-456/-789/-012/-345`. The number of hard and soft score levels needs to be set at configuration time, it's not flexible to change during solving.

```
<scoreDirectorFactory>
  <scoreDefinitionType>BENDABLE</scoreDefinitionType>
  <bendableHardLevelCount>2</bendableHardLevelCount>
  <bendableSoftLevelCount>3</bendableSoftLevelCount>
  ...
</scoreDirectorFactory>
```

5.2.5. Implementing a custom Score

The `ScoreDefinition` interface defines the score representation.

To implement a custom Score, you'll also need to implement a custom `ScoreDefinition`. Extend `AbstractScoreDefinition` (preferable by copy pasting `HardSoftScoreDefinition` or `SimpleScoreDefinition`) and start from there.

Then hook your custom `ScoreDefinition` in your `SolverConfig.xml`:

```
<scoreDirectorFactory>
  <scoreDefinitionClass>...MyScoreDefinition</scoreDefinitionClass>
  ...
</scoreDirectorFactory>
```

5.3. Calculate the `Score`

5.3.1. Score calculation types

There are several ways to calculate the `Score` of a `Solution`:

- **Simple Java score calculation**: implement a single Java method
- **Incremental Java score calculation**: implement multiple Java methods
- **Drools score calculation**: implement score rules

Every score calculation type can use any Score definition. For example, simple Java score calculation can output a `HardSoftScore`.

All score calculation types are Object Orientated and can reuse existing Java code.

Important

The score calculation should be read-only: it should not change the planning entities or the problem facts in any way. For example, it must not call a setter method on a planning entity in a Drools score rule's RHS. This does not apply to *logically inserted* objects, which can be changed by the score rules who logically inserted them in the first place.

OptaPlanner will not recalculate the score of a `Solution` if it can predict it (unless an *environmentMode assertion* is enabled). For example, after a winning step is done, there is no need to calculate the score because that move was done and undone earlier. As a result, there's no guarantee that such changes applied during score calculation are actually done.

5.3.2. Simple Java score calculation

A simple way to implement your score calculation in Java.

- Advantages:
  - Plain old Java: no learning curve
  - Opportunity to delegate score calculation to an existing code base or legacy system
- Disadvantages:
  - Slower and less scalable

- Because there is no *incremental score calculation*

Just implement one method of the interface `SimpleScoreCalculator`:

```
public interface SimpleScoreCalculator<Sol extends Solution> {

    Score calculateScore(Sol solution);

}
```

For example in n queens:

```
public class NQueensSimpleScoreCalculator implements SimpleScoreCalculator<NQueens> {

    public SimpleScore calculateScore(NQueens nQueens) {
        int n = nQueens.getN();
        List<Queen> queenList = nQueens.getQueenList();

        int score = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                Queen leftQueen = queenList.get(i);
                Queen rightQueen = queenList.get(j);
               if (leftQueen.getRow() != null && rightQueen.getRow() != null) {
                    if (leftQueen.getRowIndex() == rightQueen.getRowIndex()) {
                        score--;
                    }
                    if (leftQueen.getAscendingDiagonalIndex() == rightQueen.getAscendingDiagona
                        score--;
                    }
                    if (leftQueen.getDescendingDiagonalIndex() == rightQueen.getDescendingDiago
                        score--;
                    }
                }
            }
        }
        return SimpleScore.valueOf(score);
    }

}
```

Configure it in your solver configuration:

```
  <scoreDirectorFactory>
```

```
    <scoreDefinitionType>...</scoreDefinitionType>

lanner.examples.nqueens.solver.score.NQueensSimpleScoreCalculator</
simpleScoreCalculatorClass>
  </scoreDirectorFactory>
```

Alternatively, build a `SimpleScoreCalculator` instance at runtime and set it with the programmatic API:

```
    solverFactory.getSolverConfig().getScoreDirectorFactoryConfig.setSimpleScoreCalculator(simp
```

5.3.3. Incremental Java score calculation

A way to implement your score calculation incrementally in Java.

- Advantages:
  - Very fast and scalable
    - Currently the fastest if implemented correctly
- Disadvantages:
  - Hard to write
    - A scalable implementation heavily uses maps, indexes, ... (things the Drools rule engine can do for you)
    - You have to learn, design, write and improve all these performance optimizations yourself
  - Hard to read
    - Regular score constraint changes can lead to a high maintenance cost

Implement all the methods of the interface `IncrementalScoreCalculator` and extend the class `AbstractIncrementalScoreCalculator`:

```
public interface IncrementalScoreCalculator<Sol extends Solution> {

    void resetWorkingSolution(Sol workingSolution);

    void beforeEntityAdded(Object entity);

    void afterEntityAdded(Object entity);

    void beforeVariableChanged(Object entity, String variableName);
```

```
    void afterVariableChanged(Object entity, String variableName);

    void beforeEntityRemoved(Object entity);

    void afterEntityRemoved(Object entity);

    Score calculateScore();

}
```

For example in n queens:

```
public class NQueensAdvancedIncrementalScoreCalculator extends AbstractIncrementalScoreCalculat

    private Map<Integer, List<Queen>> rowIndexMap;
    private Map<Integer, List<Queen>> ascendingDiagonalIndexMap;
    private Map<Integer, List<Queen>> descendingDiagonalIndexMap;

    private int score;

    public void resetWorkingSolution(NQueens nQueens) {
        int n = nQueens.getN();
        rowIndexMap = new HashMap<Integer, List<Queen>>(n);
        ascendingDiagonalIndexMap = new HashMap<Integer, List<Queen>>(n * 2);
        descendingDiagonalIndexMap = new HashMap<Integer, List<Queen>>(n * 2);
        for (int i = 0; i < n; i++) {
            rowIndexMap.put(i, new ArrayList<Queen>(n));
            ascendingDiagonalIndexMap.put(i, new ArrayList<Queen>(n));
            descendingDiagonalIndexMap.put(i, new ArrayList<Queen>(n));
            if (i != 0) {
              ascendingDiagonalIndexMap.put(n - 1 + i, new ArrayList<Queen>(n));
                descendingDiagonalIndexMap.put((-i), new ArrayList<Queen>(n));
            }
        }
        score = 0;
        for (Queen queen : nQueens.getQueenList()) {
            insert(queen);
        }
    }

    public void beforeEntityAdded(Object entity) {
        // Do nothing
    }

    public void afterEntityAdded(Object entity) {
        insert((Queen) entity);
    }
```

```
    public void beforeVariableChanged(Object entity, String variableName) {
        retract((Queen) entity);
    }

    public void afterVariableChanged(Object entity, String variableName) {
        insert((Queen) entity);
    }

    public void beforeEntityRemoved(Object entity) {
        retract((Queen) entity);
    }

    public void afterEntityRemoved(Object entity) {
        // Do nothing
    }

    private void insert(Queen queen) {
        Row row = queen.getRow();
        if (row != null) {
            int rowIndex = queen.getRowIndex();
            List<Queen> rowIndexList = rowIndexMap.get(rowIndex);
            score -= rowIndexList.size();
            rowIndexList.add(queen);
            List<Queen> ascendingDiagonalIndexList = ascendingDiagonalIndexMap.get(queen.getAsc
            score -= ascendingDiagonalIndexList.size();
            ascendingDiagonalIndexList.add(queen);
            List<Queen> descendingDiagonalIndexList = descendingDiagonalIndexMap.get(queen.getD
            score -= descendingDiagonalIndexList.size();
            descendingDiagonalIndexList.add(queen);
        }
    }

    private void retract(Queen queen) {
        Row row = queen.getRow();
        if (row != null) {
            List<Queen> rowIndexList = rowIndexMap.get(queen.getRowIndex());
            rowIndexList.remove(queen);
            score += rowIndexList.size();
            List<Queen> ascendingDiagonalIndexList = ascendingDiagonalIndexMap.get(queen.getAsc
            ascendingDiagonalIndexList.remove(queen);
            score += ascendingDiagonalIndexList.size();
            List<Queen> descendingDiagonalIndexList = descendingDiagonalIndexMap.get(queen.getD
            descendingDiagonalIndexList.remove(queen);
            score += descendingDiagonalIndexList.size();
        }
    }

    public SimpleScore calculateScore() {
```

```
        return SimpleScore.valueOf(score);
    }

}
```

Configure it in your solver configuration:

```
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>

score.NQueensAdvancedIncrementalScoreCalculator</
incrementalScoreCalculatorClass>
  </scoreDirectorFactory>
```

Optionally, to get better output when the `IncrementalScoreCalculator` is corrupted in `environmentMode` `FAST_ASSERT` or `FULL_ASSERT`, you can overwrite the method `buildScoreCorruptionAnalysis` from `AbstractIncrementalScoreCalculator`.

5.3.4. Drools score calculation

5.3.4.1. Overview

Implement your score calculation using the Drools rule engine. Every score constraint is written as one or more score rules.

- Advantages:
  - Incremental score calculation for free
    - Because most DRL syntax uses forward chaining, it does incremental calculation without any extra code
  - Score constraints are isolated as separate rules
    - Easy to add or edit existing score rules
  - Flexibility to augment your score constraints by
    - Defining them in decision tables
      - Excel (XLS) spreadsheet
      - Guvnor WebUI
    - Translate them into natural language with DSL
    - Store and release in the Guvnor repository

- Performance optimizations in future versions for free
  - In every release, the Drools rule engine tends to become faster.
- Disadvantages:
  - DRL learning curve
  - Usage of DRL
    - Polyglot fear can prohibit the use of a new language such as DRL in some organizations

5.3.4.2. Drools score rules configuration

There are several ways to define where your score rules live.

5.3.4.2.1. A scoreDrl resource on the classpath

This is the easy way: the score rule live in a DRL file which is a resource on the classpath. Just add your score rules `*.drl` file in the solver configuration:

```
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>
     <scoreDrl>/org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</
scoreDrl>
  </scoreDirectorFactory>
```

You can add multiple `<scoreDrl>` entries if needed, but normally you'll define all your score rules in 1 file.

Optionally, you can also set drools configuration properties but beware of backwards compatibility issues:

```
  <scoreDirectorFactory>
    ...
     <scoreDrl>/org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</
scoreDrl>
    <kieBaseConfigurationProperties>
      <drools.equalityBehavior>...</drools.equalityBehavior>
    </kieBaseConfigurationProperties>
  </scoreDirectorFactory>
```

5.3.4.2.2. A KieBase (possibly defined by Drools Workbench)

If you prefer to build the `KieBase` yourself or if you're combining Planner with Drools Workbench (formerly known as Guvnor), you can set the `KieBase` on the `SolverFactory` before building the `Solver`:

```
solverFactory.getSolverConfig().getScoreDirectorFactoryConfig.setKieBase(kieBase);
```

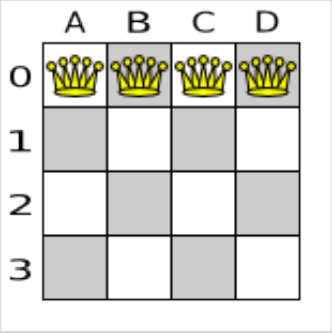
> **Note**
>
> To be able to define your score rules in Drools Workbench, you'll want to:
>
> 1. Upload the optaplanner-core jar as a POJO model.
> 2. Add a global variable called `scoreHolder` (see below).

5.3.4.3. Implementing a score rule

Here's an example of a score constraint implemented as a score rule in a DRL file:

```
rule "multipleQueensHorizontal"
    when
        Queen($id : id, row != null, $i : rowIndex)
        Queen(id > $id, rowIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end
```

This score rule will fire once for every 2 queens with the same `rowIndex`. The `(id > $id)` condition is needed to assure that for 2 queens A and B, it can only fire for (A, B) and not for (B, A), (A, A) or (B, B). Let's take a closer look at this score rule on this solution of 4 queens:



In this solution the multipleQueensHorizontal score rule will fire for 6 queen couples: (A, B), (A, C), (A, D), (B, C), (B, D) and (C, D). Because none of the queens are on the same vertical or diagonal line, this solution will have a score of `-6`. An optimal solution of 4 queens has a score of `0`.

> **Note**
>
> Notice that every score rule will relate to at least 1 planning entity class (directly or indirectly though a logically inserted fact).
>
> This is normal: it would be a waste of time to write a score rule that only relates to problem facts, as the consequence will never change during planning, no matter what the possible solution.

> **Note**
>
> The variable `kcontext` is a magic variable in Drools Expert. The scoreHolder's method uses it to do incremental score calculation correctly and to create a `ConstraintMatch` instance.

5.3.4.4. Weighing score rules

A `ScoreHolder` instance is asserted into the `KieSession` as a global called `scoreHolder`. Your score rules need to (directly or indirectly) update that instance.

```
global SimpleScoreHolder scoreHolder;

rule "multipleQueensHorizontal"
    when
        Queen($id : id, row != null, $i : rowIndex)
        Queen(id > $id, rowIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end

// multipleQueensVertical is obsolete because it is always 0

rule "multipleQueensAscendingDiagonal"
    when
        Queen($id : id, row != null, $i : ascendingDiagonalIndex)
        Queen(id > $id, ascendingDiagonalIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end
```

```
rule "multipleQueensDescendingDiagonal"
    when
        Queen($id : id, row != null, $i : descendingDiagonalIndex)
        Queen(id > $id, descendingDiagonalIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end
```

Most use cases will also weigh their constraint types or even their matches differently, by using a specific weight for each constraint match.

Here's an example from CurriculumCourse, where assigning a `Lecture` to a `Room` which is missing 2 seats is weighted equally bad as having 1 isolated `Lecture` in a `Curriculum`:

```
global HardSoftScoreHolder scoreHolder;

// RoomCapacity: For each lecture, the number of students that attend the course
 must be less or equal
// than the number of seats of all the rooms that host its lectures.
// Each student above the capacity counts as 1 point of penalty.
rule "roomCapacity"
    when
        $room : Room($capacity : capacity)
       $lecture : Lecture(room == $room, studentSize > $capacity, $studentSize :
 studentSize)
    then
       scoreHolder.addSoftConstraintMatch(kcontext, ($capacity - $studentSize));
end

// CurriculumCompactness: Lectures belonging to a curriculum should be adjacent
// to each other (i.e., in consecutive periods).
// For a given curriculum we account for a violation every time there is one
 lecture not adjacent
// to any other lecture within the same day.
// Each isolated lecture in a curriculum counts as 2 points of penalty.
rule "curriculumCompactness"
    when
        ...
    then
        scoreHolder.addSoftConstraintMatch(kcontext, -2);
end
```

5.3.5. Detecting invalid scores

Put the `environmentMode` in `FULL_ASSERT` (or `FAST_ASSERT`) to detect corruption in the *incremental score calculation*. For more information, *see the section about `environmentMode`*.

However, that will not verify that your score calculator implements your score constraints as your business actually desires.

A piece of incremental score calculator code can be difficult to write and to review. Assert its correctness by using a different implementation (for example a `SimpleScoreCalculator`) to do the assertions triggered by the `environmentMode`. Just configure the different implementation as a `assertionScoreDirectorFactory`:

```
  <environmentMode>FAST_ASSERT</environmentMode>
  ...
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>
     <scoreDrl>/org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</
scoreDrl>
    <assertionScoreDirectorFactory>

lanner.examples.nqueens.solver.score.NQueensSimpleScoreCalculator</
simpleScoreCalculatorClass>
    </assertionScoreDirectorFactory>
  </scoreDirectorFactory>
```

This way, the `scoreDrl` will be validated by the `SimpleScoreCalculator`.

5.4. Score calculation performance tricks

5.4.1. Overview

The `Solver` will normally spend most of its execution time running the score calculation (which is called in its deepest loops). Faster score calculation will return the same solution in less time with the same algorithm, which normally means a better solution in equal time.

5.4.2. Average calculation count per second

After solving a problem, the `Solver` will log the *average calculation count per second*. This is a good measurement of Score calculation performance, despite that it is affected by non score calculation execution time. It depends on the problem scale of the problem dataset. Normally, even for high scale problems, it is higher than `1000`, except when you're using `SimpleScoreCalculator`.

Important

When improving your score calculation, focus on maximizing the average calculation count per second, instead of maximizing the best score. A big improvement in score calculation can sometimes yield little or no best score improvement, for example when the algorithm is stuck in a local or global optima.

If you're watching the calculation count instead, score calculation improvements are far more visible.

Furthermore, watching the calculation count, allows you to remove or add score constraints, and still compare it with the original calculation count. Comparing the best score with the original would be wrong, because it's comparing apples and oranges.

5.4.3. Incremental score calculation (with delta's)

When a `Solution` changes, incremental score calculation (AKA delta based score calculation), will calculate the delta with the previous state to find the new `Score`, instead of recalculating the entire score on every solution evaluation.

For example, if a single queen A moves from row `1` to `2`, it won't bother to check if queen B and C can attack each other, since neither of them changed.

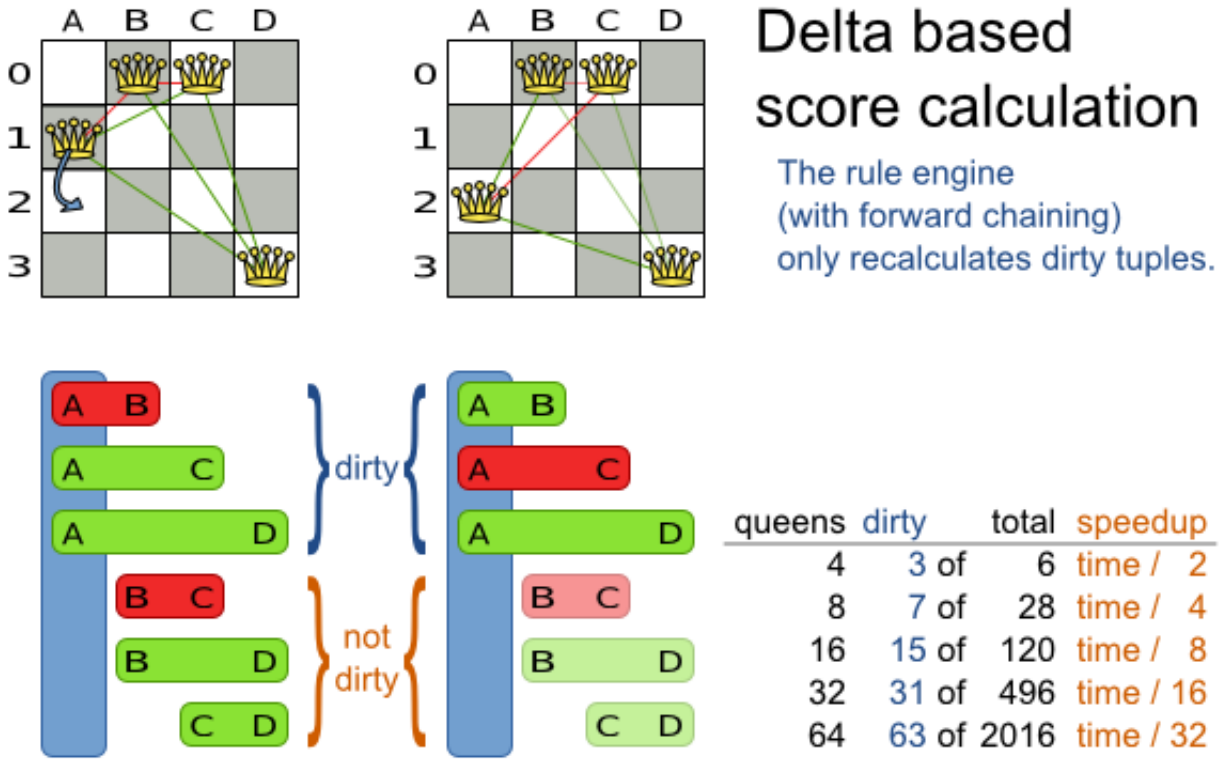


Figure 5.1. Incremental score calculation for the 4 queens puzzle

This is a huge performance and scalability gain. **Drools score calculation gives you this huge scalability gain without forcing you to write a complicated incremental score calculation algorithm.** Just let the Drools rule engine do the hard work.

Notice that the speedup is relative to the size of your planning problem (your *n*), making incremental score calculation far more scalable.

5.4.4. Avoid calling remote services during score calculation

Do not call remote services in your score calculation (except if you're bridging `SimpleScoreCalculator` to a legacy system). The network latency will kill your score calculation performance. Cache the results of those remote services if possible.

If some parts of a constraint can be calculated once, when the `Solver` starts, and never change during solving, then turn them into *cached problem facts*.

5.4.5. Unused constraint

If you know a certain constraint can never be broken, don't bother writing a score constraint for it. For example in n queens, the score calculation doesn't check if multiple queens occupy the same column, because a `Queen`'s `column` never changes and every `Solution` starts with each `Queen` on a different `column`.

> **Note**
>
> Don't go overboard with this. If some datasets don't use a specific constraint but others do, just return out of the constraint as soon as you can. There is no need to dynamically change your score calculation based on the dataset.

5.4.6. Build-in hard constraint

Instead of implementing a hard constraint, you can sometimes make it build-in too. For example: If `Lecture` A should never be assigned to `Room` X, but it uses ValueRangeProvider on Solution, the `Solver` will often try to assign it to `Room` X too (only to find out that it breaks a hard constraint). Use *filtered selection* to define that Course A should only be assigned a `Room` other than X.

This tends to give a good performance gain, not just because the score calculation is faster, but mainly because most optimization algorithms will spend less time evaluating unfeasible solutions.

> **Note**
>
> Don't go overboard with this. Many optimization algorithms rely on the freedom to break hard constraints when changing planning entities, to get out of local optima. There is a real risk of trading short term benefits for long term harm.

5.4.7. Other performance tricks

- Verify that your score calculation happens in the correct `Number` type. If you're making the sum of `int` values, don't let Drools sum it in a `double` which takes longer.

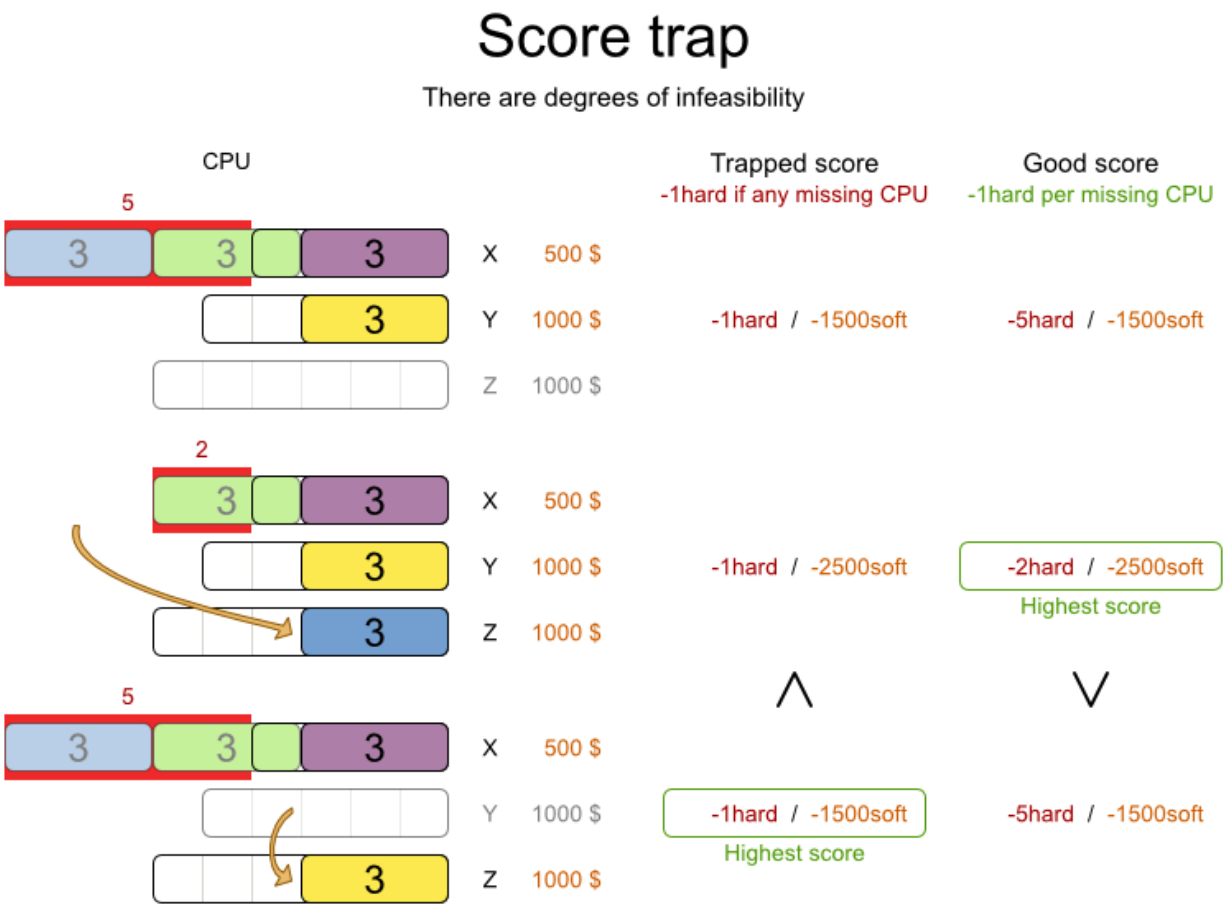
- For optimal performance, always use server mode (`java -server`). We have seen performance increases of 50% by turning on server mode.
- For optimal performance, use the latest Java version. For example, in the past we have seen performance increases of 30% by switching from java 1.5 to 1.6.
- Always remember that premature optimization is the root of all evil. Make sure your design is flexible enough to allow configuration based tweaking.

5.4.8. Score trap

Make sure that none of your score constraints cause a score trap. A trapped score constraint uses the same weight for different constraint matches, when it could just as easily use a different weight. It effectively lumps its constraint matches together, which creates a flatlined score function for that constraint. This can cause a solution state in which several moves need to be done to resolve or lower the weight of that single constraint. Some examples of score traps:

- If you need 2 doctors at each table, but you're only moving 1 doctor at a time. So the solver has no incentive to move a doctor to a table with no doctors. Punish a table with no doctors more then a table with only 1 doctor in that score constraint in the score function.
- 2 exams needs to be conducted at the same time, but you're only move 1 exam at a time. So the solver has a disincentive move one of those exams to another timeslot without moving the other in the same move. Add a course-grained move that moves both exams at the same time.

For example, consider this score trap. If the blue item moves from an overloaded computer to an empty computer, the hard score should improve. The trapped score implementation fails to do that:



The Solver should eventually get out of this trap, but it will take a lot of effort (especially if there are even more processes on the overloaded computer). Before they do that, they might actually start moving more processes into that overloaded computer, as there is no penalty for doing so.

Note

Avoiding score traps does not mean that your score function should be smart enough to avoid local optima. Leave it to the optimization algorithms to deal with the local optima.

Avoiding score traps means to avoid - for each score constraint individually - a flatlined score function.

Important

Always specify the degree of infeasibility. The business will often say: "if the solution is infeasible, it doesn't matter how infeasible it." While that's true for the business, it's not true for score calculation: it benefits from knowing how infeasible

it is. In practice, soft constraints usually do this naturally and it's just a matter of doing it for the hard constraints too.

There are several ways to deal with a score trap:

- Improve the score constraint to make a distinction in the score weight. For example: penalize `-1hard` for every missing CPU, instead of just `-1hard` if any CPU is missing.
- If changing the score constraint is not allowed from the business perspective, add a lower score level with a score constraint that makes such a distinction. For example: penalize `-1subsoft` for every missing CPU, on top of `-1hard` if any CPU is missing. The business ignores the subsoft score level.
- Add course-grained moves and union select them with the existing fine-grained moves. A course-grained move effectively does multiple moves to directly get out of a score trap with a single move. For example: move multiple items from the same container to another container.

5.4.9. stepLimit benchmark

Not all score constraints have the same performance cost. Sometimes 1 score constraint can kill the score calculation performance outright. Use the benchmarker to do a 1 minute run and check what happens to the average calculation count per second if you comment out all but 1 of the score constraints.

5.5. Reusing the score calculation outside the Solver

Other parts of your application, for example your webUI, might need to calculate the score too. Do that by reusing the `ScoreDirectorFactory` of the `Solver` to build a separate `ScoreDirector` for that webUI:

```
ScoreDirectorFactory scoreDirectorFactory = solver.getScoreDirectorFactory();
ScoreDirector guiScoreDirector = scoreDirectorFactory.buildScoreDirector();
```

Then use it when you need to calculate the `Score` of a `Solution`:

```
guiScoreDirector.setWorkingSolution(solution);
Score score = guiScoreDirector.calculateScore();
```

To explain in the GUI what entities are causing which part of the `Score`, get the `ConstraintMatch` objects from the `ScoreDirector` (after calling `calculateScore()`):

```
for (ConstraintMatchTotal constraintMatchTotal : guiScoreDirector.getConstraintMatchTotals()) {
```

```
    String constraintName = constraintMatchTotal.getConstraintName();
    Number weightTotal = constraintMatchTotal.getWeightTotalAsNumber();
    for (ConstraintMatch constraintMatch : constraintMatchTotal.getConstraintMatchSet()) {
       List<Object> justificationList = constraintMatch.getJustificationList();
        Number weight = constraintMatch.getWeightAsNumber();
        ...
    }
}
```

Chapter 6. Optimization algorithms

6.1. Search space size in the real world

The number of possible solutions for a planning problem can be mind blowing. For example:

- 4 queens has `256` possible solutions (`4 ^ 4`) and 2 optimal solutions.
- 5 queens has `3125` possible solutions (`5 ^ 5`) and 1 optimal solution.
- 8 queens has `16777216` possible solutions (`8 ^ 8`) and 92 optimal solutions.
- 64 queens has more than `10^115` possible solutions (`64 ^ 64`).
- Most real-life planning problems have an incredible number of possible solutions and only 1 or a few optimal solutions.

For comparison: the minimal number of atoms in the known universe (10^80). As a planning problem gets bigger, the search space tends to blow up really fast. Adding only 1 extra planning entity or planning value can heavily multiply the running time of some algorithms.

An algorithm that checks every possible solution (even with pruning) can easily run for billions of years on a single real-life planning problem. What we really want is to **find the best solution in the limited time at our disposal**. Planning competitions (such as the International Timetabling Competition) show that local search variations (tabu search, simulated annealing, ...) usually perform best for real-world problems given real-world time limitations.

6.2. Does Planner find the optimal solution?

The business wants the optimal solution, but they also have other requirements:

- Scale out: Large production datasets must not crash and have good results too.
- Optimize the right problem: The constraints must match the actual business needs.
- Available time: The solution must be found in time, before it becomes useless to execute.
- Reliability: Every dataset must have at least a decent result (better than a human planner).

Given these requirements, and despite the promises of some salesmen, it's usually impossible for anyone or anything to find the optimal solution. Therefore, Planner focuses on finding the best solution in available time. In *realistic, independent competitions*, Planner often comes out as the best *reusable* software.

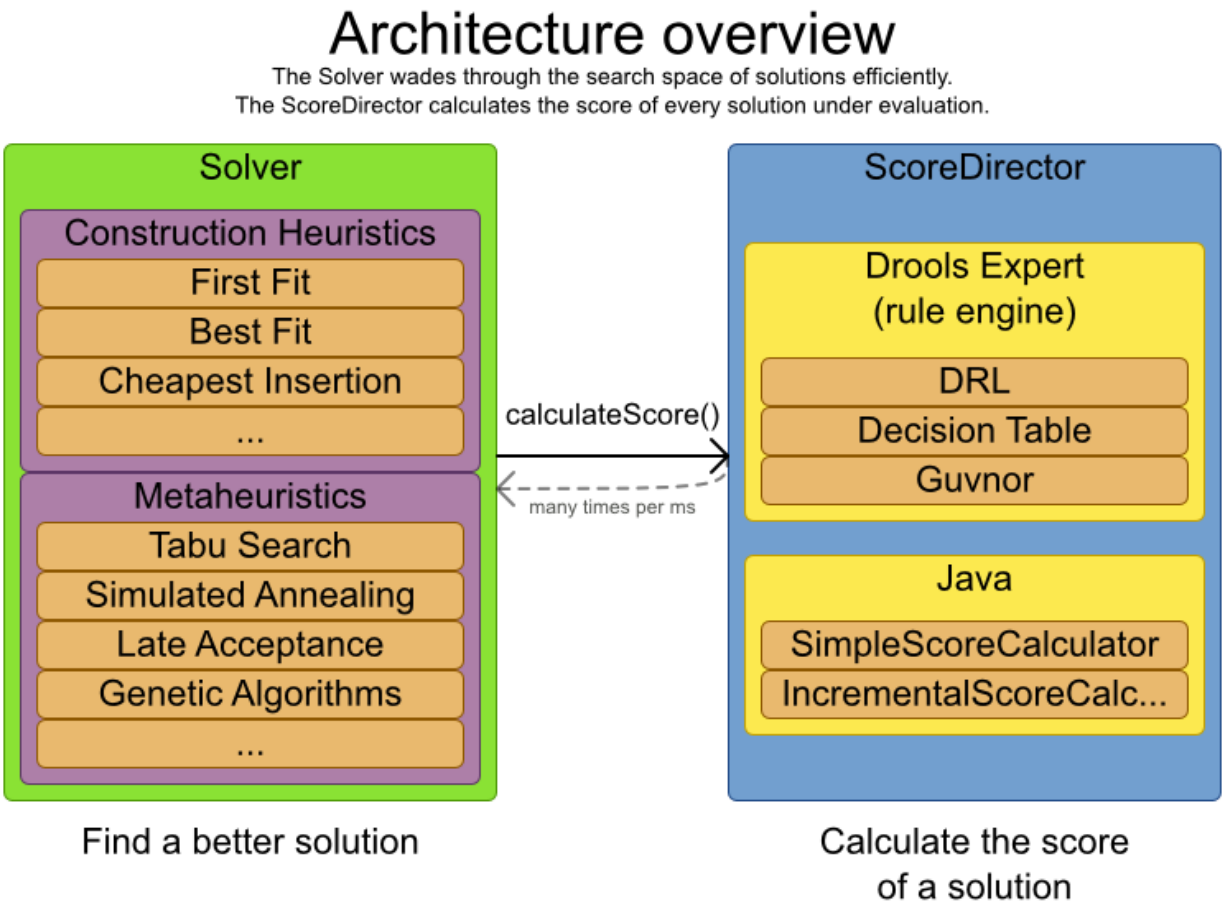
The nature of NP-complete problems make scaling a prime concern. **The result quality of a small dataset guarantees nothing about the result quality of a large dataset.** Scaling problems

cannot be mitigated by hardware purchases. Start testing with a production sized dataset as soon as possible. Don't asses quality on small datasets (unless production encounters such datasets). Instead, solve a production sized dataset and compare with the results of longer execution, different algorithms and - if available - the human planner.

6.3. Architecture overview

OptaPlanner is the first framework to combine optimization algorithms (metaheuristics, ...) with score calculation by a rule engine such as Drools Expert. This combination turns out to be a very efficient, because:

- A rule engine such as Drools Expert is **great for calculating the score** of a solution of a planning problem. It makes it easy and scalable to add additional soft or hard constraints such as "a teacher shouldn't teach more then 7 hours a day". It does delta based score calculation without any extra code. However it tends to be not suitable to actually find new solutions.
- An optimization algorithm is **great at finding new improving solutions** for a planning problem, without necessarily brute-forcing every possibility. However it needs to know the score of a solution and offers no support in calculating that score efficiently.



6.4. Optimization algorithms overview

Table 6.1. Optimization algorithms overview

Algorithm	Scalable?	Optimal?	Easy to use?	Tweakable?	Requires CH?
Exact algorithms					
Brute force	0/5	5/5	5/5	0/5	No
Branch and bound	0/5	5/5	4/5	1/5	No
Construction heuristics (CH)					
First Fit	5/5	1/5	5/5	1/5	No
First Fit Decreasing	5/5	2/5	4/5	2/5	No
Best Fit	5/5	2/5	4/5	2/5	No
Best Fit Decreasing	5/5	2/5	4/5	2/5	No
Cheapest Insertion	3/5	2/5	5/5	2/5	No
Metaheuristics (MH)					
Local Search					
Hill-Climbing	5/5	2/5	4/5	3/5	Yes
Tabu Search	5/5	4/5	3/5	5/5	Yes
Simulated Annealing	5/5	4/5	2/5	5/5	Yes
Late Acceptance	5/5	4/5	3/5	5/5	Yes
Late Simulated Annealing	5/5	4/5	3/5	5/5	Yes
Evolutionary Algorithms					
Evolutionary Strategies	4/5	3/5	?/5	?/5	Yes
Genetic Algorithms	4/5	3/5	?/5	?/5	Yes

If you want to learn more about metaheuristics, read the free book *Essentials of Metaheuristics* [http://www.cs.gmu.edu/~sean/book/metaheuristics/] or *Clever Algorithms* [http://www.cleveralgorithms.com/].

6.5. Which optimization algorithms should I use?

The *best* optimization algorithms configuration for your use case depends heavily on your use case. Nevertheless, this vanilla recipe will get you into the game with a pretty good configuration, probably much better than what you're used to.

Start with a quick configuration that involves little or no configuration and optimization code:

1. First Fit

Next, implement planning entity difficulty comparison and turn it into:

1. First Fit Decreasing

Next, implement moves and add tabu search behind it:

1. First Fit Decreasing
2. Tabu Search (use planning entity tabu)

At this point *the free lunch is over*. The return on invested time lowers. The result is probably already more than good enough.

But you can do even better, at a lower return on invested time. Use the Benchmarker and try a couple of Simulated Annealing or Late Acceptance configurations, for example:

1. First Fit Decreasing
2. Late Acceptance (try several sizes)

And combine them with Tabu Search:

1. First Fit Decreasing
2. Late Acceptance (relatively long time)
3. Tabu Search (relatively short time)

If you have time, continue experimenting even further. Blog about your experiments!

6.6. `SolverPhase`

A `Solver` can use multiple optimization algorithms in sequence. **Each optimization algorithm is represented by a `SolverPhase`.** There is never more than 1 `SolverPhase` solving at the same time.

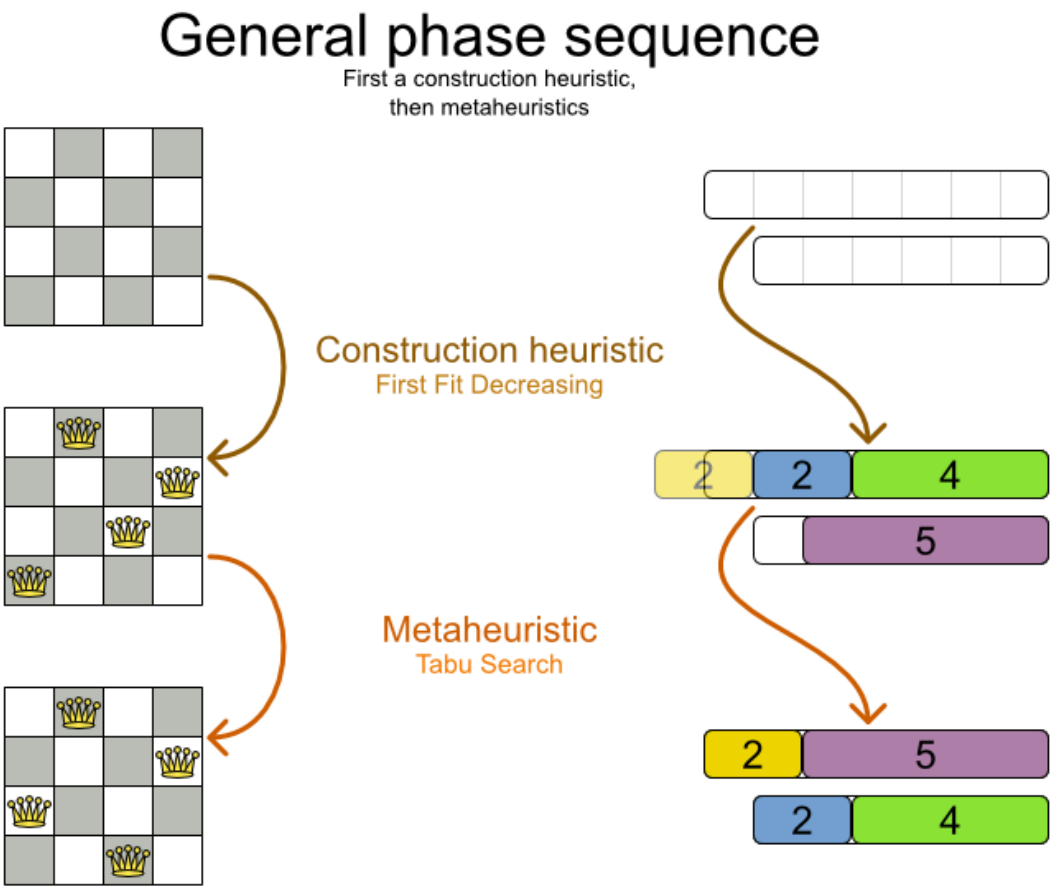
> **Note**
>
> Some `SolverPhase` implementations can combine techniques from multiple optimization algorithms, but they are still just 1 `SolverPhase`. For example: a local search `SolverPhase` can do simulated annealing with planning entity tabu.

Here's a configuration that runs 3 phases in sequence:

```
<solver>
  ...
  <constructionHeuristic>
```

```
    ... <!-- First phase: First Fit Decreasing -->
  </constructionHeuristic>
  <localSearch>
    ... <!-- Second phase: Simulated Annealing -->
  </localSearch>
  <localSearch>
    ... <!-- Third phase: Tabu Search -->
  </localSearch>
</solver>
```

The solver phases are run in the order defined by solver configuration. When the first phase terminates, the second phase starts, and so on. When the last phase terminates, the `Solver` terminates. Usually, a solver will first run a construction heuristic and then run 1 or multiple metaheuristics:



Some phases (especially construction heuristics) will terminate automatically. Other phases (especially metaheuristics) will only terminate if the phase is configured to terminate:

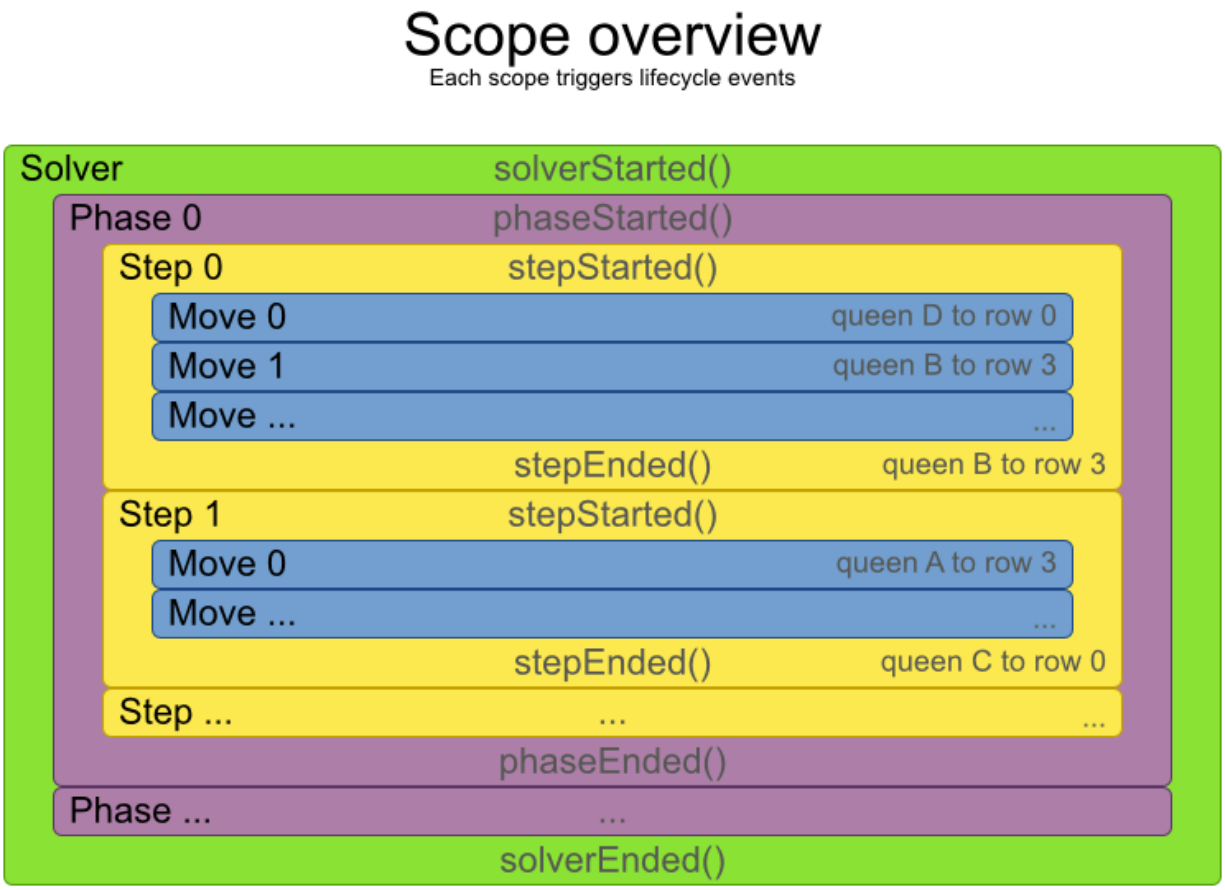
```
<solver>
```

```
  ...
  <termination><!-- Solver termination -->
    <maximumSecondsSpend>90</maximumSecondsSpend>
  </termination>
  <localSearch>
    <termination><!-- Phase termination -->
       <maximumSecondsSpend>60</maximumSecondsSpend><!-- Give the next phase a
 chance to run too, before the Solver terminates -->
    </termination>
    ...
  </localSearch>
  <localSearch>
    ...
  </localSearch>
</solver>
```

If the `Solver` terminates (before the last phase terminates itself), the current phase is terminated and all subsequent phases won't run.

6.7. Scope overview

A solver will iteratively run phases. Each phase will usually iteratively run steps. Each step, in turn, usually iteratively runs moves. These form 4 nested scopes: solver, phase, step and move.



Configure *logging* to display the log messages of each scope.

6.8. Termination

Not all phases terminate automatically and sometimes you don't want to wait that long anyway. A `Solver` can be terminated synchronously by up-front configuration or asynchronously from another thread.

Especially metaheuristic phases will need to be told when to stop solving. This can be because of a number of reasons: the time is up, the perfect score has been reached, ... The only thing you can't depend on, is on finding the optimal solution (unless you know the optimal score), because a metaheuristic algorithm generally doesn't know it when it finds the optimal solution. For real-life problems this doesn't turn out to be much of a problem, because finding the optimal solution could take billions of years, so you 'll want to terminate sooner anyway. The only thing that matters is finding the best solution in the available time.

For synchronous termination, configure a `Termination` on a `Solver` or a `SolverPhase` when it needs to stop. You can implement your own `Termination`, but the build-in implementations should suffice for most needs. Every `Termination` can calculate a *time gradient* (needed for

some optimization algorithms), which is a ratio between the time already spend solving and the estimated entire solving time of the `Solver` or `SolverPhase`.

6.8.1. TimeMillisSpendTermination

Terminates when an amount of time has been reached:

```
<termination>
    <maximumTimeMillisSpend>500</maximumTimeMillisSpend>
</termination>
```

```
<termination>
    <maximumSecondsSpend>10</maximumSecondsSpend>
</termination>
```

```
<termination>
    <maximumMinutesSpend>5</maximumMinutesSpend>
</termination>
```

```
<termination>
    <maximumHoursSpend>1</maximumHoursSpend>
</termination>
```

Note

If you use this `Termination`, you will most likely sacrifice perfect reproducibility (even with `environmentMode` `REPRODUCIBLE`) because of an available CPU time difference:

- The available CPU time influences the number of steps that can be taken, which might be a few more or less.
- The `Termination` might produce slightly different time gradient values, which will send time gradient based algorithms (such as simulated annealing) on a radically different path.

6.8.2. ScoreAttainedTermination

Terminates when a certain score has been reached. You can use this `Termination` if you know the perfect score, for example for 4 queens:

```
<termination>
    <scoreAttained>0</scoreAttained>
</termination>
```

For a planning problem with hard and soft constraints, it could look like this:

```
<termination>
    <scoreAttained>0hard/-5000soft</scoreAttained>
</termination>
```

You can use this `Termination` to terminate once it reaches a feasible solution.

6.8.3. StepCountTermination

Terminates when an amount of steps has been reached:

```
<termination>
    <maximumStepCount>100</maximumStepCount>
</termination>
```

This `Termination` can only be used for a `SolverPhase`, not for the `Solver` itself.

6.8.4. UnimprovedStepCountTermination

Terminates when the best score hasn't improved in a number of steps:

```
<termination>
    <maximumUnimprovedStepCount>100</maximumUnimprovedStepCount>
</termination>
```

If the score hasn't improved recently, it's probably not going to improve soon anyway and it's not worth the effort to continue. We have observed that once a new best solution is found (even after a long time of no improvement on the best solution), the next few steps tend to improve the best solution too.

This `Termination` can only be used for a `SolverPhase`, not for the `Solver` itself.

6.8.5. Combining multiple Terminations

Terminations can be combined, for example: terminate after 100 steps or if a score of 0 has been reached:

```
    <termination>
        <terminationCompositionStyle>OR</terminationCompositionStyle>
        <maximumStepCount>100</maximumStepCount>
        <scoreAttained>0</scoreAttained>
    </termination>
```

Alternatively you can use AND, for example: terminate after reaching a feasible score of at least -100 and no improvements in 5 steps:

```
    <termination>
        <terminationCompositionStyle>AND</terminationCompositionStyle>
        <maximumUnimprovedStepCount>5</maximumUnimprovedStepCount>
        <scoreAttained>-100</scoreAttained>
    </termination>
```

This example ensures it doesn't just terminate after finding a feasible solution, but also completes any obvious improvements on that solution before terminating.

6.8.6. Asynchronous termination from another thread

Sometimes you'll want to terminate a Solver early from another thread, for example because a user action or a server restart. This cannot be configured by a `Termination` as it's impossible to predict when and if it will occur. Therefore the `Solver` interface has these 2 thread-safe methods:

```
public interface Solver {

    // ...

    boolean terminateEarly();
    boolean isTerminateEarly();

}
```

If you call the `terminateEarly()` method from another thread, the `Solver` will terminate at its earliest convenience and the `solve()` method will return in the original `Solver` thread.

6.9. SolverEventListener

Each time a new best solution is found, the `Solver` fires a `BestSolutionChangedEvent`.

To listen to such events, add a `SolverEventListener` to the `Solver`:

```
public interface Solver {

    // ...

    void addEventListener(SolverEventListener eventListener);
    void removeEventListener(SolverEventListener eventListener);

}
```

6.10. Custom SolverPhase

Between phases or before the first phase, you might want to execute a custom action on the `Solution` to get a better score. Yet you'll still want to reuse the score calculation. For example, to implement a custom construction heuristic without implementing an entire `SolverPhase`.

> **Note**
>
> Most of the time, a custom construction heuristic is not worth the hassle. The supported constructions heuristics are configurable (so you can tweak them with the `Benchmarker`), `Termination` aware and support partially initialized solutions too.

Implement the `CustomSolverPhaseCommand` interface:

```
public interface CustomSolverPhaseCommand {

    void changeWorkingSolution(ScoreDirector scoreDirector);

}
```

For example:

```
public class ExaminationSolutionInitializer implements CustomSolverPhaseCommand {

    public void changeWorkingSolution(ScoreDirector scoreDirector) {
        Examination examination = (Examination) scoreDirector.getWorkingSolution();
```

```
        for (Exam exam : examination.getExamList()) {
            Score unscheduledScore = scoreDirector.calculateScore();
            ...
            for (Period period : examination.getPeriodList()) {
                scoreDirector.beforeVariableChanged(exam, "period");
                exam.setPeriod(period)
                scoreDirector.afterVariableChanged(exam, "period");
                Score score = scoreDirector.calculateScore();
                ...
            }
            ...
        }
    }

}
```

Warning

Any change on the planning entities in a `CustomSolverPhaseCommand` must be notified to the `ScoreDirector`.

Warning

Do not change any of the planning facts in a `CustomSolverPhaseCommand`. That will corrupt the `Solver` because any previous score or solution was for a different problem. If you want to do that, see *repeated planning* and real-time planning instead.

And configure it like this:

```
<solver>
  ...
  <customSolverPhase>

initializer.ExaminationSolutionInitializer</
customSolverPhaseCommandClass>
  </customSolverPhase>
  ... <!-- Other phases -->
</solver>
```

It's possible to configure multiple `customSolverPhaseCommandClass` instances, which will be run in sequence.

Important

If the changes of a `CustomSolverPhaseCommand` don't result in a better score, the best solution won't be changed (so effectively nothing will have changed for the next `SolverPhase` or `CustomSolverPhaseCommand`). To force such changes anyway, use `forceUpdateBestSolution`:

```
  <customSolverPhase>
                <customSolverPhaseCommandClass>...MyUninitializer</
customSolverPhaseCommandClass>
    <forceUpdateBestSolution>true</forceUpdateBestSolution>
  </customSolverPhase>
```

Note

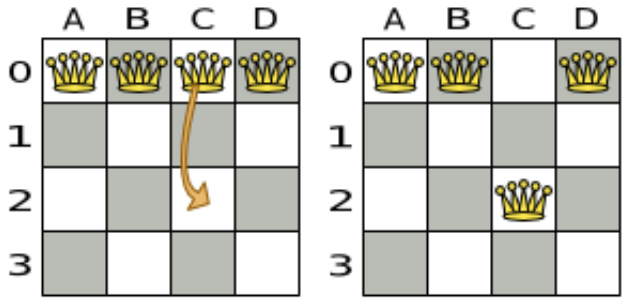
If the `Solver` or `SolverPhase` wants to terminate while a `CustomSolverPhaseCommand` is still running, it will wait to terminate until the `CustomSolverPhaseCommand` is done, however long that takes.

Chapter 7. `Move` and neighborhood selection

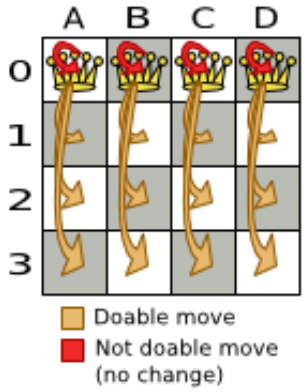
7.1. `Move` and neighborhood introduction

7.1.1. What is a `Move`?

A `Move` is a change (or set of changes) from a solution A to a solution B. For example, the move below changes queen `C` from row `0` to row `2`:

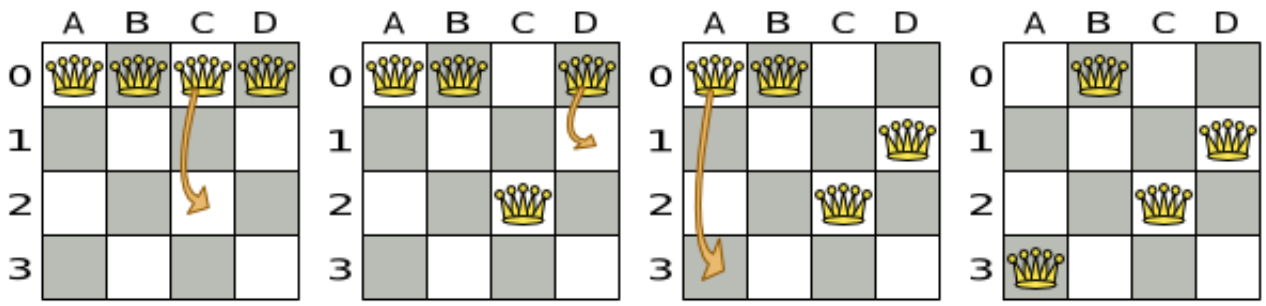


The new solution is called a *neighbor* of the original solution, because it can be reached in a single `Move`. Although a single move can change multiple queens, the neighbors of a solution should always be a very small subset of all possible solutions. For example, on that original solution, these are all possible `changeMove`'s:



If we ignore the 4 `changeMove`'s that have not impact and are therefore not doable, we can see that number of moves is `n * (n - 1) = 12`. This is far less than the number of possible solutions, which is `n ^ n = 256`. As the problem scales out, the number of possible moves increases far less than the number of possible solutions.

Yet, in 4 `changeMove`'s or less we can reach any solution. For example we can reach a very different solution in 3 `changeMove`'s:



> **Note**
>
> There are many other types of moves besides `changeMove`'s. Many move types are included out-of-the-box, but you can also implement custom moves.
>
> A `Move` can affect multiple entities or even create/delete entities. But it must not change the problem facts.

All optimization algorithms use `Move`'s to transition from one solution to a neighbor solution. Therefor, all the optimization algorithms are confronted with `Move` selection: the craft of creating and iterating moves efficiently and the art of finding the most promising subset of random moves to evaluate first.

7.1.2. What is a `MoveSelector`?

A `MoveSelector`'s main function is to create `Iterator<Move>` when needed. An optimization algorithm will iterate through a subset of those moves.

Here's an example how to configure a `changeMoveSelector` for the optimization algorithm Local Search:

```
<localSearch>
  <changeMoveSelector/>
  ...
</localSearch>
```

Out of the box, this works and all properties of the `changeMoveSelector` are defaulted sensibly (unless that fails fast due to ambiguity). On the other hand, the configuration can be customized significantly for specific use cases. For example: you want want to configure a filter to discard pointless moves.

7.1.3. Subselecting of entities, values and other moves

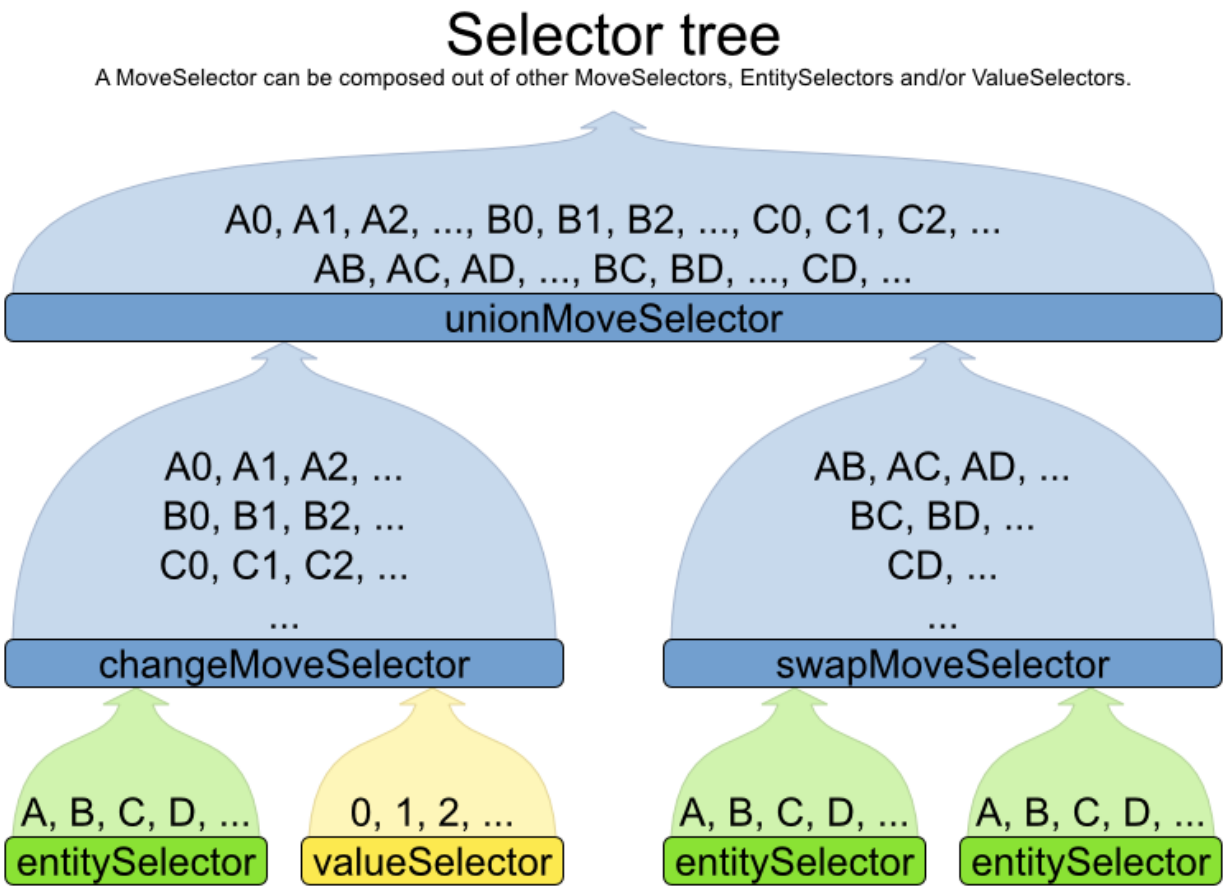
To create a `Move`, we need to select 1 or more planning entities and/or planning values to move. Just like `MoveSelectors`, `EntitySelectors` and `ValueSelectors` need to support a similar

feature set (such as scalable just-in-time selection). Therefor, they implement a common interface `Selector` and they are configured similarly.

A MoveSelector is often composed out of `EntitySelectors`, `ValueSelectors` or even other `MoveSelectors`, which can be configured individually if desired:

```
<unionMoveSelector>
  <changeMoveSelector>
    <entitySelector>
      ...
    </entitySelector>
    <valueSelector>
      ...
    </valueSelector>
    ...
  </changeMoveSelector>
  ...
</unionMoveSelector>
```

Together, this structure forms a `Selector` tree:



The root of this tree is a `MoveSelector` which is injected into the optimization algorithm implementation to be (partially) iterated in every step.

7.2. General `Selector` features

7.2.1. `CacheType`: Create moves ahead of time or Just In Time

A `Selector`'s `cacheType` determines when a selection (such as a `Move`, an entity, a value, ...) is created and how long it lives.

Almost every `Selector` supports setting a `cacheType`:

```
<changeMoveSelector>
  <cacheType>PHASE</cacheType>
  ...
</changeMoveSelector>
```

The following `cacheTypes` are supported:

- `JUST_IN_TIME` (default): Not cached. Construct each selection (`Move`, ...) just before it's used. This scales up well in memory footprint.
- `STEP`: Cached. Create each selection (`Move`, ...) at the beginning of a step and cache them in a list for the remainder of the step. This scales up badly in memory footprint.
- `PHASE`: Cached. Create each selection (`Move`, ...) at the beginning of a `SolverPhase` and cache them in a list for the remainder of the `SolverPhase`. Some selections cannot be phase cached because the list changes every step. This scales up badly in memory footprint, but has a slight performance gain.
- `SOLVER`: Cached. Create each selection (`Move`, ...) at the beginning of a `Solver` and cache them in a list for the remainder of the `Solver`. Some selections cannot be solver cached because the list changes every step. This scales up badly in memory footprint, but has a slight performance gain.

A `cacheType` can be set on composite selectors too:

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <changeMoveSelector/>
  <swapMoveSelector/>
  ...
</unionMoveSelector>
```

Nested selectors of a cached selector cannot be configured to be cached themselves, unless it's a higher `cacheType`. For example: a `STEP` cached `unionMoveSelector` can hold a `PHASE` cached `changeMoveSelector`, but not a `STEP` cached `changeMoveSelector`.

7.2.2. SelectionOrder: original, sorted, random, shuffled or probabilistic

A `Selector`'s `selectionOrder` determines the order in which the selections (such as `Moves`, entities, values, ...) are iterated. An optimization algorithm will usually only iterate through a subset of its `MoveSelector`'s selections, starting from the start, so the `selectionOrder` is critical to decide which `Moves` are actually evaluated.

Almost every `Selector` supports setting a `selectionOrder`:

```
<changeMoveSelector>
  ...
  <selectionOrder>RANDOM</selectionOrder>
  ...
</changeMoveSelector>
```

The following `selectionOrders` are supported:

- `ORIGINAL`: Select the selections (`Moves`, entities, values, ...) in default order. Each selection will be selected only once.
  - For example: A0, A1, A2, A3, ..., B0, B1, B2, B3, ..., C0, C1, C2, C3, ...
- SORTED: Select the selections (`Moves`, entities, values, ...) in sorted order. Each selection will be selected only once. Requires `cacheType >= STEP`. Mostly used on an `entitySelector` or `valueSelector` for construction heuristics. See *sorted selection*.
  - For example: A0, B0, C0, ..., A2, B2, C2, ..., A1, B1, C1, ...
- RANDOM (default): Select the selections (`Moves`, entities, values, ...) in non-shuffled random order. A selection might be selected multiple times. This scales up well in performance because it does not require caching.
  - For example: C2, A3, B1, C2, A0, C0, ...
- SHUFFLED: Select the selections (`Moves`, entities, values, ...) in shuffled random order. Each selection will be selected only once. Requires `cacheType >= STEP`. This scales up badly in performance, not just because it requires caching, but also because a random number is generated for each element, even if it's not selected (which is the grand majority when scaling up).
  - For example: C2, A3, B1, A0, C0, ...
- PROBABILISTIC: Select the selections (`Moves`, entities, values, ...) in random order, based on the selection probability of each element. A selection with a higher probability has a higher chance to be selected than elements with a lower probability. A selection might be selected multiple times. Requires `cacheType >= STEP`. Mostly used on an `entitySelector` or `valueSelector`. See *probabilistic selection*.
  - For example: B1, B1, A1, B2, B1, C2, B1, B1, ...

A `selectionOrder` can be set on composite selectors too.

> **Note**
>
> When a `Selector` is cached, all of its nested `Selectors` will naturally default to `selectionOrder ORIGINAL`. Avoid overwriting the `selectionOrder` of those nested `Selectors`.

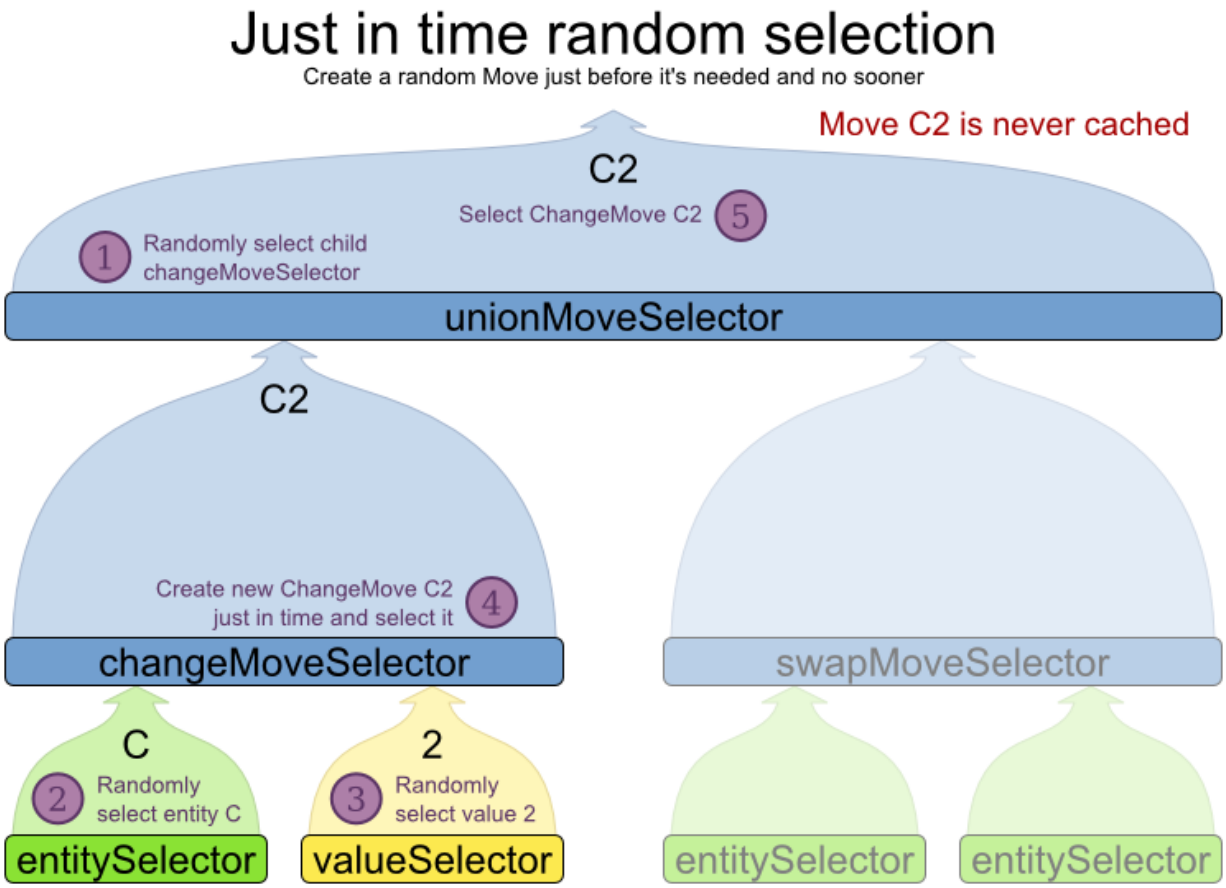
7.2.3. Recommended combinations of `CacheType` and `SelectionOrder`

7.2.3.1. Just in time random selection (default)

This combination is great for big use cases (10 000 entities or more), as it scales up well in memory footprint and performance. Other combinations are often not even viable on such sizes. It works for smaller use cases too, so it's a good way to start out. It's the default, so this explicit configuration of `cacheType` and `selectionOrder` is actually obsolete:

```
<unionMoveSelector>
  <cacheType>JUST_IN_TIME</cacheType>
  <selectionOrder>RANDOM</selectionOrder>

  <changeMoveSelector/>
  <swapMoveSelector/>
</unionMoveSelector>
```

Here's how it works. When `Iterator<Move>.next()` is called, a child `MoveSelector` is randomly selected (1), which creates a random `Move` is created (2, 3, 4) and is then returned (5):



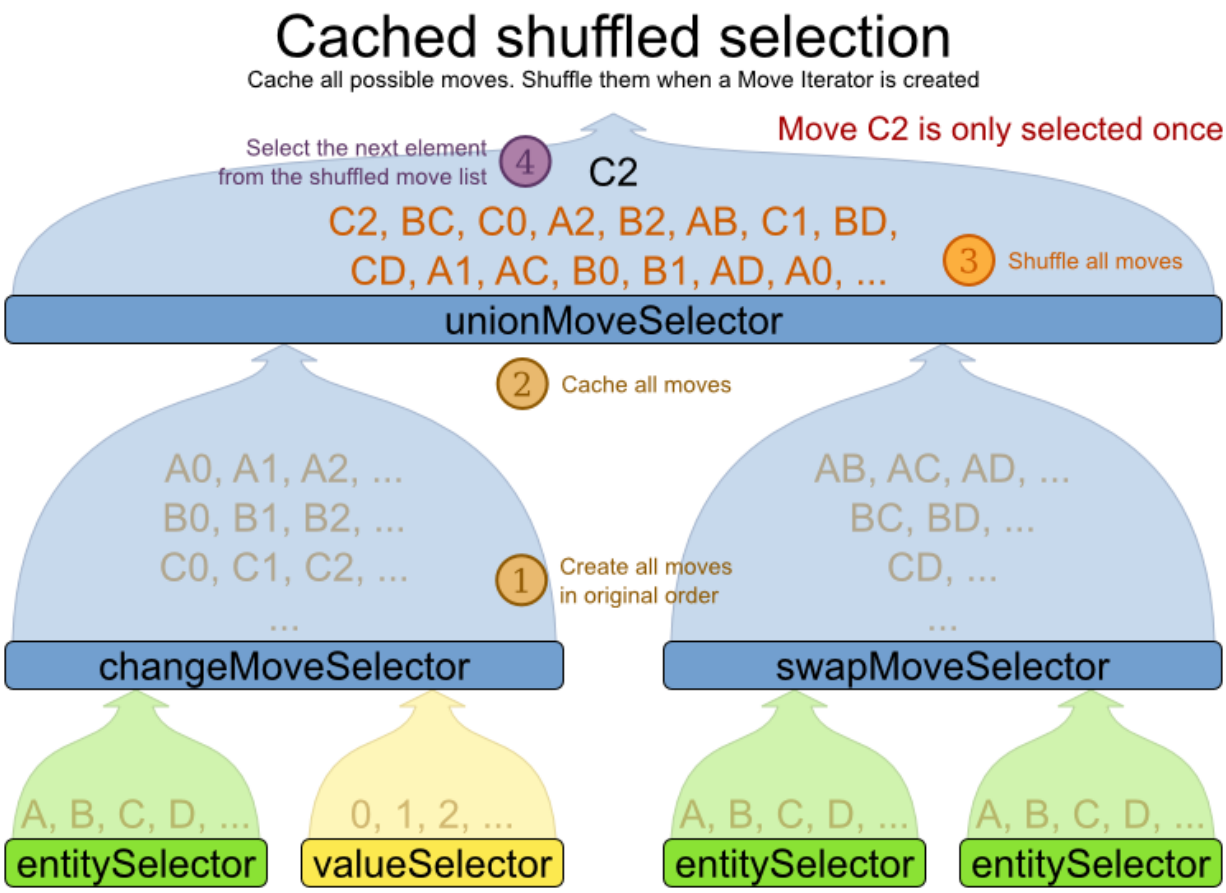
Notice that **it never creates a list of `Moves`** and it generates random numbers only for `Moves` that are actually selected.

7.2.3.2. Cached shuffled selection

This combination often wins for small and medium use cases (5000 entities or less). Beyond that size, it scales up badly in memory footprint and performance.

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SHUFFLED</selectionOrder>

  <changeMoveSelector/>
  <swapMoveSelector/>
</unionMoveSelector>
```

Here's how it works: At the start of the phase (or step depending on the `cacheType`), all moves are created (1) and cached (2). When `MoveSelector.iterator()` is called, the moves are shuffled (3). When `Iterator<Move>.next()` is called, the next element in the shuffled list is returned (4):



Notice that **each `Move` will only be selected once**, even though they are selected in random order.

Use cacheType PHASE if none of the (possibly nested) Selectors require `STEP`. Otherwise, do something like this:

```
<unionMoveSelector>
  <cacheType>STEP</cacheType>
  <selectionOrder>SHUFFLED</selectionOrder>

  <changeMoveSelector>
    <cacheType>PHASE</cacheType>
  </changeMoveSelector>
  <swapMoveSelector/>
    <cacheType>PHASE</cacheType>
  </swapMoveSelector>
  <pillarSwapMoveSelector/><!-- Does not support cacheType PHASE -->
</unionMoveSelector>
```

7.2.3.3. Cached random selection

This combination is often a worthy competitor for medium use cases, especially with fast stepping optimization algorithms (such as simulated annealing). Unlike cached shuffled selection, it doesn't waste time shuffling the move list at the beginning of every step.

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>RANDOM</selectionOrder>

  <changeMoveSelector/>
  <swapMoveSelector/>
</unionMoveSelector>
```

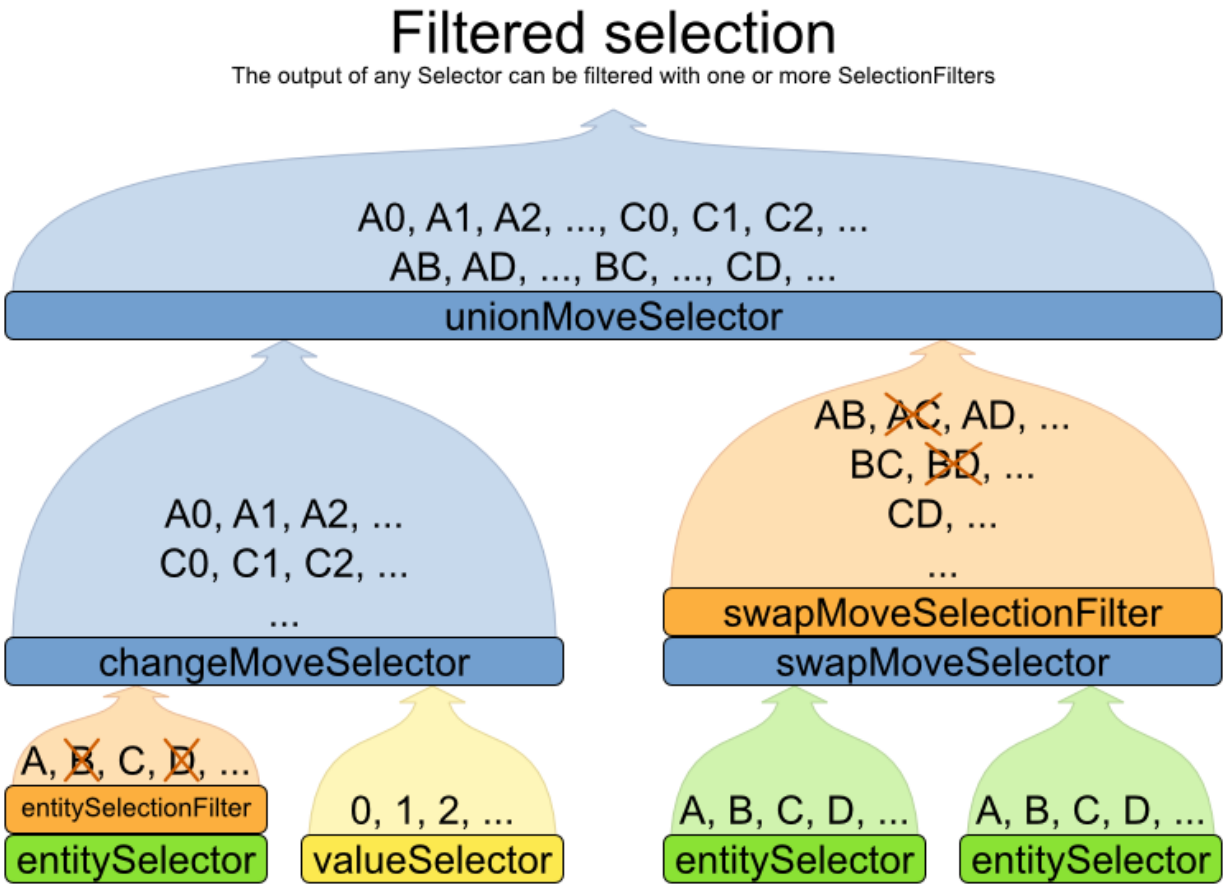
7.2.4. Filtered selection

There are certain moves that you don't want to select, because:

- The move is pointless and would only waste CPU time. For example, swapping 2 lectures of the same course will result in the same score and the same schedule because all lectures of 1 course are interchangeable (same teacher, same students, same topic).
- Doing the move would break *a build-in hard constraint*, so the solution would be infeasible but the score function doesn't check build-in hard constraints (for performance gain). For example, don't change a gym lecture to a room which is not a gym room.

- Note that any build-in hard constraint must usually be filtered on every move type. For example, don't swap the room of a gym lecture with another lecture if the other lecture's original room isn't a gym room.

Filtered selection can happen on any Selector in the selector tree, including any `MoveSelector`, `EntitySelector` or `ValueSelector`. It works with any kind of `cacheType` and `selectionOrder`.



Filtering uses the interface `SelectionFilter`:

```
public interface SelectionFilter<T> {

    boolean accept(ScoreDirector scoreDirector, T selection);

}
```

Implement the method `accept` to return `false` on a discarded `selection`. Unaccepted moves will not be selected and will therefore never have their method `doMove` called.

```
public class DifferentCourseSwapMoveFilter implements SelectionFilter<SwapMove> {

    public boolean accept(ScoreDirector scoreDirector, SwapMove move) {
        Lecture leftLecture = (Lecture) move.getLeftEntity();
        Lecture rightLecture = (Lecture) move.getRightEntity();
        return !leftLecture.getCourse().equals(rightLecture.getCourse());
    }

}
```

Apply the filter on the lowest level possible. In most cases, you 'll need to know both the entity and the value involved and you'll have to apply a `filterClass` on the `moveSelector`:

```
    <swapMoveSelector>

ner.examples.curriculumcourse.solver.move.DifferentCourseSwapMoveFilter</
filterClass>
    </swapMoveSelector>
```

But if possible, apply it on a lower levels, such as a `filterClass` on the `entitySelector` or `valueSelector`:

```
    <changeMoveSelector>
      <entitySelector>
        <filterClass>...EntityFilter</filterClass>
      </entitySelector>
    </changeMoveSelector>
```

You can configure multiple `filterClass` elements on a single selector.

7.2.5. Sorted selection

Sorted selection can happen on any Selector in the selector tree, including any `MoveSelector`, `EntitySelector` or `ValueSelector`. It does not work with `cacheType` `JUST_IN_TIME` and `it only works with selectionOrder` `SORTED`.

It's mostly used in construction heuristics.

7.2.5.1. Sorted selection by `Comparator`

The easiest way to sort a `Selector` is with a plain old `Comparator`:

```
public class CloudProcessDifficultyComparator implements Comparator<CloudProcess> {
```

```
    public int compare(CloudProcess a, CloudProcess b) {
        return new CompareToBuilder()
              .append(a.getRequiredMultiplicand(), b.getRequiredMultiplicand())
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```

You 'll also need to configure it (unless it's annotated applied for this optimization algorithm):

```
    <entitySelector>
      <cacheType>PHASE</cacheType>
      <selectionOrder>SORTED</selectionOrder>
                  <sorterComparatorClass>...CloudProcessDifficultyComparator</
sorterComparatorClass>
      <sorterOrder>DESCENDING</sorterOrder>
    </entitySelector>
```

7.2.5.2. Sorted selection by `SelectionSorterWeightFactory`

If you need the entire `Solution` to sort a `Selector`, use a `SelectionSorterWeightFactory` instead:

```
public interface SelectionSorterWeightFactory<Sol extends Solution, T> {

    Comparable createSorterWeight(Sol solution, T selection);

}
```

```
public class QueenDifficultyWeightFactory implements SelectionSorterWeightFactory<NQueens, Quee

    public Comparable createSorterWeight(NQueens nQueens, Queen queen) {
        int distanceFromMiddle = calculateDistanceFromMiddle(nQueens.getN(), queen.getColumnInd
        return new QueenDifficultyWeight(queen, distanceFromMiddle);
    }

    // ...

    public static class QueenDifficultyWeight implements Comparable<QueenDifficultyWeight> {

        private final Queen queen;
        private final int distanceFromMiddle;
```

```
        public QueenDifficultyWeight(Queen queen, int distanceFromMiddle) {
            this.queen = queen;
            this.distanceFromMiddle = distanceFromMiddle;
        }

        public int compareTo(QueenDifficultyWeight other) {
            return new CompareToBuilder()
                // The more difficult queens have a lower distance to the middle
                      .append(other.distanceFromMiddle, distanceFromMiddle) //
 Decreasing
                    // Tie breaker
                  .append(queen.getColumnIndex(), other.queen.getColumnIndex())
                    .toComparison();
        }

    }

}
```

You 'll also need to configure it (unless it's annotated and automatically applied for this optimization algorithm):

```
    <entitySelector>
      <cacheType>PHASE</cacheType>
      <selectionOrder>SORTED</selectionOrder>
                    <sorterWeightFactoryClass>...QueenDifficultyWeightFactory</
sorterWeightFactoryClass>
      <sorterOrder>DESCENDING</sorterOrder>
    </entitySelector>
```

7.2.5.3. Sorted selection by `SelectionSorter`

Alternatively, you can also use the interface `SelectionSorter` directly:

```
public interface SelectionSorter<T> {

    void sort(ScoreDirector scoreDirector, List<T> selectionList);

}
```

```
    <entitySelector>
      <cacheType>PHASE</cacheType>
      <selectionOrder>SORTED</selectionOrder>
```

```
      <sorterClass>...MyEntitySorter</sorterClass>
    </entitySelector>
```

7.2.6. Probabilistic selection

Probabilistic selection can happen on any Selector in the selector tree, including any `MoveSelector`, `EntitySelector` or `ValueSelector`. It does not work with `cacheType JUST_IN_TIME` and `it only works with selectionOrder PROBABILISTIC`.

Each selection has a `probabilityWeight`, which determines the chance that's that selection will be selected:

```
public interface SelectionProbabilityWeightFactory<T> {

    double createProbabilityWeight(ScoreDirector scoreDirector, T selection);

}
```

```
    <entitySelector>
      <cacheType>PHASE</cacheType>
      <selectionOrder>PROBABILISTIC</selectionOrder>
      <probabilityWeightFactoryClass>...MyEntityProbabilityWeightFactoryClass</
probabilityWeightFactoryClass>
    </entitySelector>
```

For example, if there are 3 entities: process A (probabilityWeight 2.0), process B (probabilityWeight 0.5) and process C (probabilityWeight 0.5), then process A will be selected 4 times more than B and C.

7.2.7. Mimic selection (record/replay)

During mimic selection, 1 normal selector records its selection and 1 or multiple other special selectors replay that selection. The recording selector acts as a normal selector and supports all other configuration properties. A replaying selector mimics the recording selection and support no other configuration properties.

The recording selector needs an `id`. A replaying selector must reference a recorder's id with a `mimicSelectorRef`:

```
      <cartesianProductMoveSelector>
        <changeMoveSelector>
          <entitySelector id="entitySelector"/>
          <valueSelector>
```

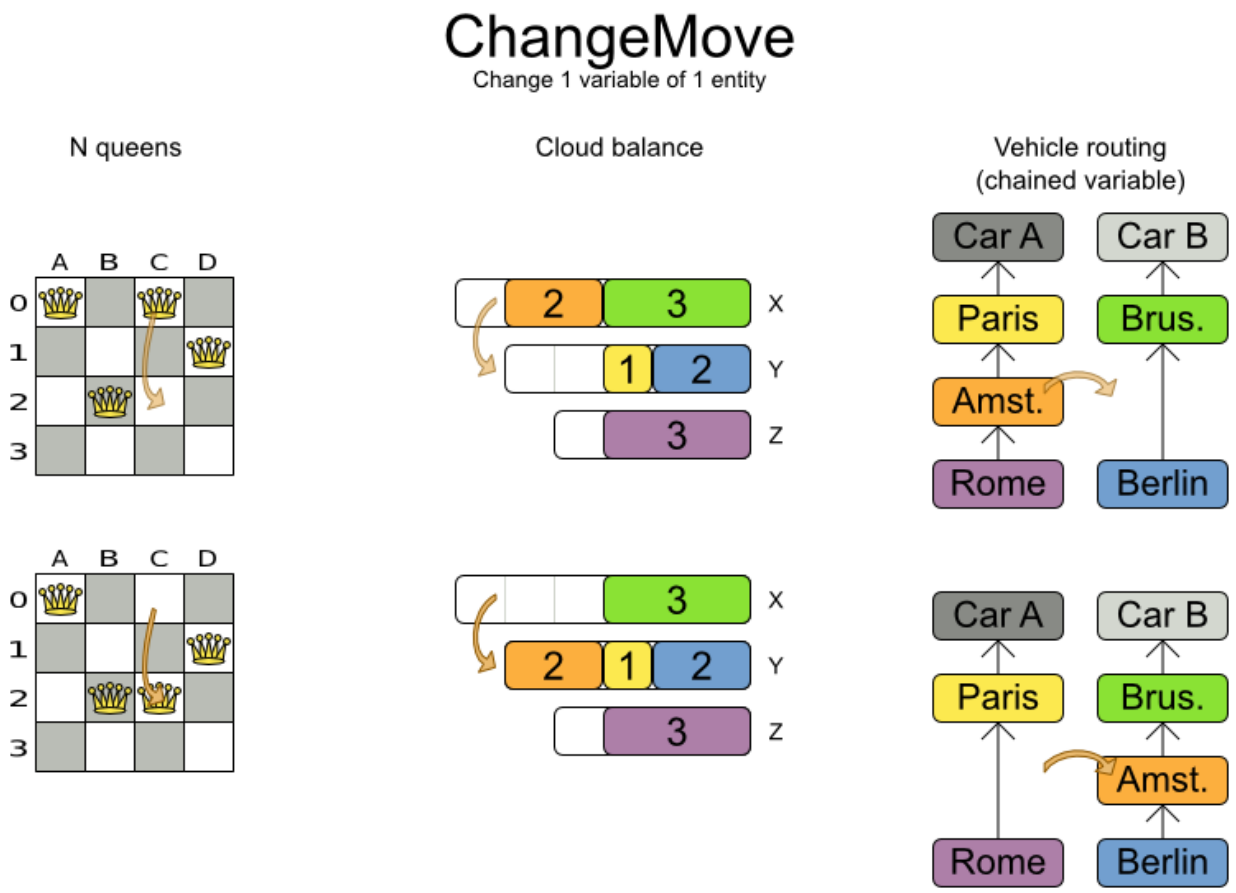
```
        <variableName>period</variableName>
      </valueSelector>
    </changeMoveSelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="entitySelector"/>
      <valueSelector>
        <variableName>room</variableName>
      </valueSelector>
    </changeMoveSelector>
  </cartesianProductMoveSelector>
```

Mimic selection is usefull to create a composite move from 2 moves that affect the same entity.

7.3. Generic MoveSelectors

7.3.1. `changeMoveSelector`

For 1 planning variable, the `ChangeMove` selects 1 planning entity and 1 planning value and assigns the entity's variable to that value.



Simplest configuration:

```
<changeMoveSelector/>
```

Advanced configuration:

```
<changeMoveSelector>
  ... <!-- Normal selector properties -->
  <entitySelector>
    <entityClass>...Lecture</entityClass>
    ...
  </entitySelector>
  <valueSelector>
    <variableName>room</variableName>
    ...
  </valueSelector>
</changeMoveSelector>
```

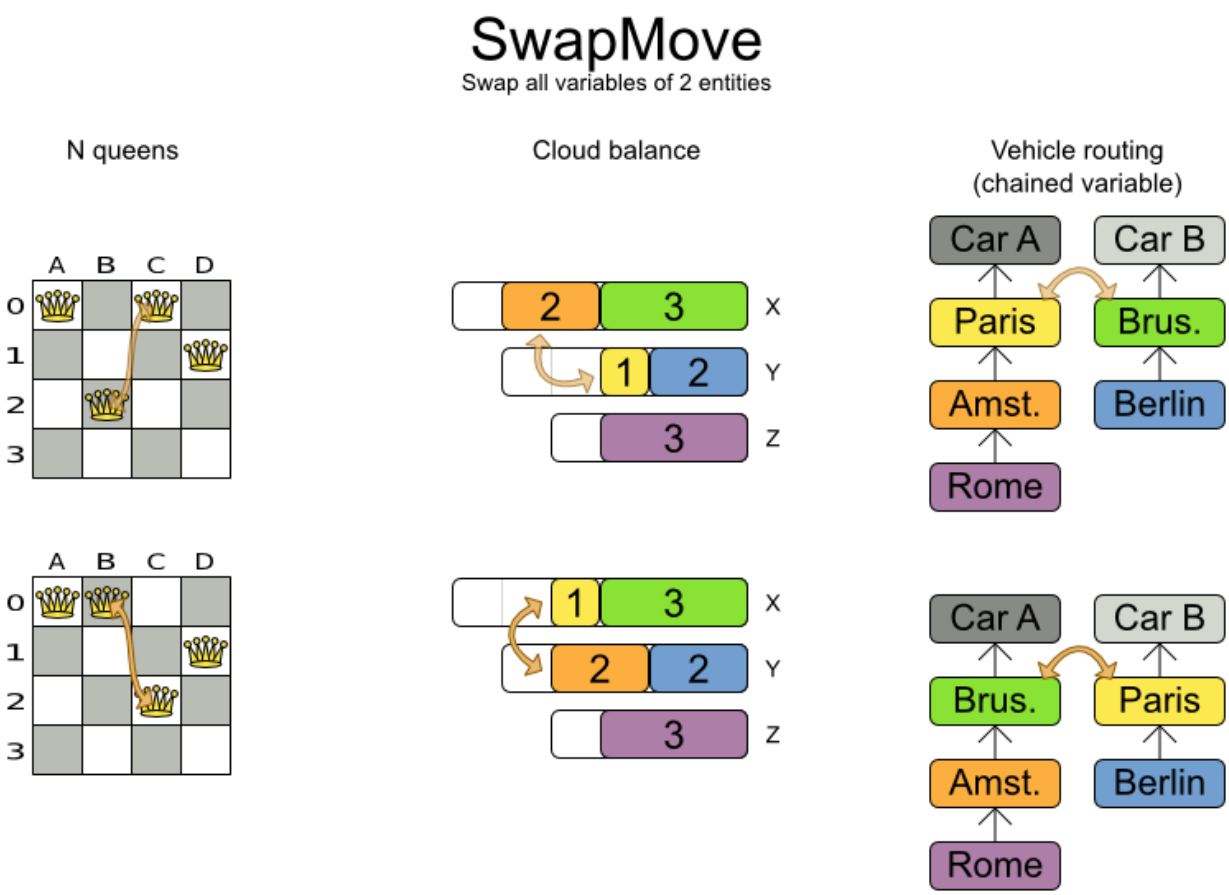
A `ChangeMove` is the finest grained move.

Important

Almost every `moveSelector` configuration injected into a metaheuristic algorithm should include a changeMoveSelector or a custom implementation. This guarantees that every possible `Solution` can be reached through applying a number of moves in sequence (not taking *score traps* into account). Of course, normally it is unioned with other, more course grained move selectors.

7.3.2. swapMoveSelector

The `SwapMove` selects 2 different planning entities and swaps the planning values of all their planning variables.



Although a `SwapMove` on a single variable is essentially just 2 `ChangeMoves`, it's often the winning step where the first of the 2 `ChangeMoves` would not be the winning step because it leave the solution in a state with broken hard constraints. For example: swapping the room of 2 lectures doesn't bring the solution in a intermediate state where both lectures are in the same room which breaks a hard constraint.

Simplest configuration:

```
<swapMoveSelector/>
```

Advanced configuration:

```
<swapMoveSelector>
  ... <!-- Normal selector properties -->
  <entitySelector>
    <entityClass>...Lecture</entityClass>
    ...
  </entitySelector>
```

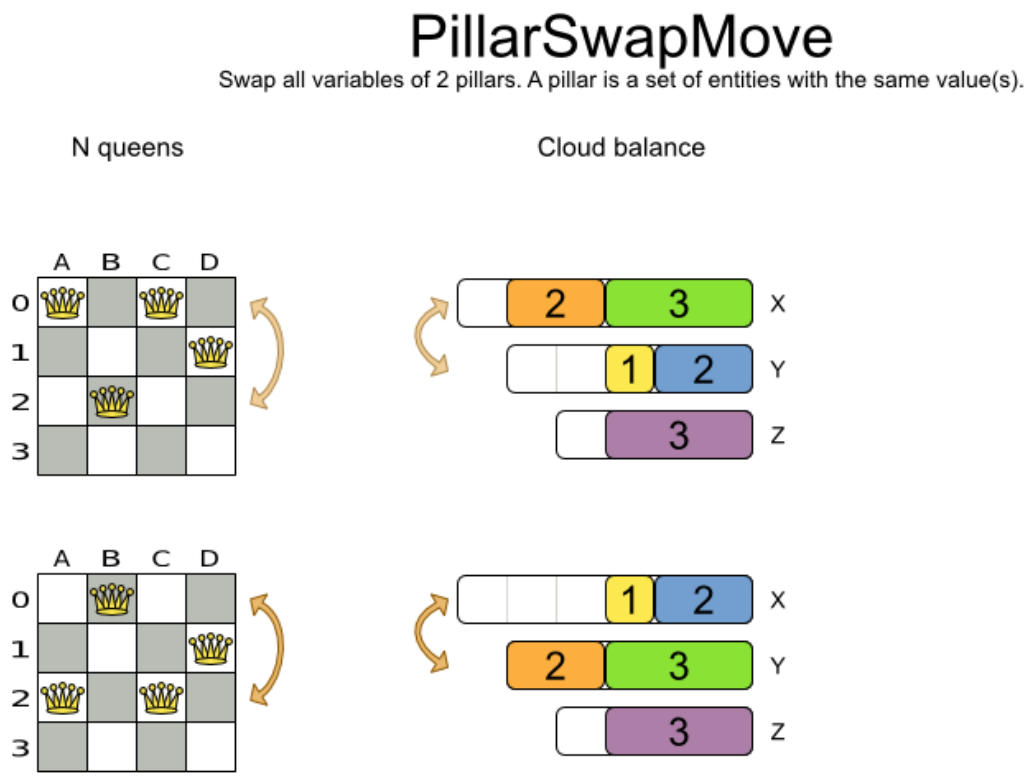
```
    <secondaryEntitySelector>
      ...
    </secondaryEntitySelector>
    <variableNameInclude>room</variableNameInclude>
    <variableNameInclude>...</variableNameInclude>
  </swapMoveSelector>
```

The `secondaryEntitySelector` is rarely needed: if it is not specified, entities from the same `entitySelector` are swapped.

If one or more `variableNameInclude` properties are specified, not all planning variables will be swapped, but only those specified. For example for course scheduling, specifying only `variableNameInclude` room will make it only swap room, not period.

7.3.3. pillarSwapMoveSelector

A *pillar* is a set of planning entities which have the same planning value(s) for each of their planning variables. The `pillarSwapMove` selects 2 different entity pillars and swaps the values of all their variables for all their entities.



Simplest configuration:

```
<pillarSwapMoveSelector/>
```

Advanced configuration:

```
<pillarSwapMoveSelector>
  ... <!-- Normal selector properties -->
  <pillarSelector>
    <entitySelector>
      <entityClass>...Lecture</entityClass>
      ...
    </entitySelector>
  </pillarSelector>
  <secondaryPillarSelector>
    <entitySelector>
      ...
    </entitySelector>
  </secondaryPillarSelector>
  <variableNameInclude>room</variableNameInclude>
  <variableNameInclude>...</variableNameInclude>
</pillarSwapMoveSelector>
```

The `secondaryPillarSelector` is rarely needed: if it is not specified, entities from the same `pillarSelector` are swapped.

The other properties are explained in *swapMoveSelector*.

7.3.4. subChainChangeMoveSelector

A *subChain* is a set of planning entities with a chained planning variable which form part of a chain. The `subChainChangeMove` selects a subChain and moves it to another place in a different or the same anchor chain.

Simplest configuration:

```
<subChainChangeMoveSelector/>
```

Advanced configuration:

```
<subChainChangeMoveSelector>
  ... <!-- Normal selector properties -->
  <subChainSelector>
    <entitySelector>
      <entityClass>...Customer</entityClass>
```

```
        ...
      </entitySelector>
      <minimumSubChainSize>2</minimumSubChainSize>
      <maximumSubChainSize>40</maximumSubChainSize>
    </subChainSelector>
    <valueSelector>
      <variableName>previousStandstill</variableName>
      ...
    </valueSelector>
    <selectReversingMoveToo>true</selectReversingMoveToo>
  </subChainChangeMoveSelector>
```

The `subChainSelector` selects a number of entities, no less than `minimumSubChainSize` (defaults to 2) and no more than `maximumSubChainSize` (defaults to infinity).

The property `selectReversingMoveToo` (defaults to true) enabled selecting the reverse of every subchain too.

7.3.5. subChainSwapMoveSelector

The `subChainSwapMove` selects 2 different subChains and moves it to another place in a different or the same anchor chain.

Simplest configuration:

```
  <subChainSwapMoveSelector/>
```

Advanced configuration:

```
  <subChainSwapMoveSelector>
    ... <!-- Normal selector properties -->
    <subChainSelector>
      <entitySelector>
        <entityClass>...Customer</entityClass>
        ...
      </entitySelector>
      <minimumSubChainSize>2</minimumSubChainSize>
      <maximumSubChainSize>40</maximumSubChainSize>
    </subChainSelector>
    <secondarySubChainSelector>
      <entitySelector>
        <entityClass>...Customer</entityClass>
        ...
      </entitySelector>
      <minimumSubChainSize>2</minimumSubChainSize>
      <maximumSubChainSize>40</maximumSubChainSize>
```

```
      </secondarySubChainSelector>
      <selectReversingMoveToo>true</selectReversingMoveToo>
    </subChainSwapMoveSelector>
```

The `secondarySubChainSelector` is rarely needed: if it is not specified, entities from the same `subChainSelector` are swapped.

The other properties are explained in *subChainChangeMoveSelector*.

7.4. Combining multiple `MoveSelector`s

7.4.1. unionMoveSelector

A `unionMoveSelector` selects a `Move` by selecting 1 of its child `MoveSelectors` to supply the next `Move`.

Simplest configuration:

```
    <unionMoveSelector>
      <...MoveSelector/>
      <...MoveSelector/>
      <...MoveSelector/>
      ...
    </unionMoveSelector>
```

Advanced configuration:

```
    <unionMoveSelector>
      ... <!-- Normal selector properties -->
          <selectorProbabilityWeightFactoryClass>...ProbabilityWeightFactory</
selectorProbabilityWeightFactoryClass>
      <changeMoveSelector>
        <fixedProbabilityWeight>...</fixedProbabilityWeight>
        ...
      </changeMoveSelector>
      <swapMoveSelector>
        <fixedProbabilityWeight>...</fixedProbabilityWeight>
        ...
      </swapMoveSelector>
      <...MoveSelector>
        <fixedProbabilityWeight>...</fixedProbabilityWeight>
        ...
      </...MoveSelector>
      ...
    </unionMoveSelector>
```

The `selectorProbabilityWeightFactory` determines in `selectionOrder RANDOM` how often a child MoveSelector is selected to supply the next Move. By default, each child `MoveSelector` has the same chance of being selected. Change the fixedProbabilityWeight of such a child to select it more often. For example, the `unionMoveSelector` can return a `SwapMove` twice as often as a `ChangeMove`:

```
    <unionMoveSelector>
      <changeMoveSelector>
        <fixedProbabilityWeight>1.0</fixedProbabilityWeight>
        ...
      </changeMoveSelector>
      <swapMoveSelector>
        <fixedProbabilityWeight>2.0</fixedProbabilityWeight>
        ...
      </swapMoveSelector>
    </unionMoveSelector>
```

The number of possible `ChangeMoves` is very different from the number of possible `SwapMoves` and furthermore it's problem dependent. To give each individual `Move` the same selection chance (as opposed to each `MoveSelector`), use the `FairSelectorProbabilityWeightFactory`:

```
    <unionMoveSelector>

lectorProbabilityWeightFactory</
  selectorProbabilityWeightFactoryClass>
      <changeMoveSelector/>
      <swapMoveSelector/>
    </unionMoveSelector>
```

7.4.2. cartesianProductMoveSelector

A `cartesianProductMoveSelector` selects a new `CompositeMove`. It builds that `CompositeMove` by selecting 1 `Move` per child `MoveSelector` and adding it to the `CompositeMove`.

Simplest configuration:

```
    <cartesianProductMoveSelector>
      <...MoveSelector/>
      <...MoveSelector/>
      <...MoveSelector/>
      ...
    </cartesianProductMoveSelector>
```

Advanced configuration:

```
    <cartesianProductMoveSelector>
      ... <!-- Normal selector properties -->
      <ignoreEmptyChildIterators>true</ignoreEmptyChildIterators>
      <changeMoveSelector>
        ...
      </changeMoveSelector>
      <swapMoveSelector>
        ...
      </swapMoveSelector>
      <...MoveSelector>
        ...
      </...MoveSelector>
      ...
    </cartesianProductMoveSelector>
```

The propery `ignoreEmptyChildIterators` (true by default) will ignore every empty `childMoveSelector` to avoid returning no moves. For example: a cartesian product of `changeMoveSelector` A and B, for which B is empty (because all it's entities are immovable) returns no moves if `ignoreEmptyChildIterators` is `false` and the moves of A if `ignoreEmptyChildIterators` is `true`.

7.5. EntitySelector

Simplest configuration:

```
      <entitySelector/>
```

Advanced configuration:

```
      <entitySelector>
        ... <!-- Normal selector properties -->
        <entityClass>org.optaplanner.examples.curriculumcourse.domain.Lecture</
entityClass>
      </entitySelector>
```

The `entityClass` property is only required if it cannot be deduced automatically because there are multiple entity classes.

7.6. ValueSelector

Simplest configuration:

```
    <valueSelector/>
```

Advanced configuration:

```
    <valueSelector>
      ... <!-- Normal selector properties -->
      <variableName>room</variableName>
    </valueSelector>
```

The `variableName` property is only required if it cannot be deduced automatically because there are multiple variables (for the related entity class).

7.7. Custom moves

7.7.1. Which move types might be missing in my implementation?

To determine which move types might be missing in your implementation, run a benchmarker *for a short amount of time* and *configure it to write the best solutions to disk*. Take a look at such a best solution: it will likely be a local optima. Try to figure out if there's a move that could get out of that local optima faster.

If you find one, implement that course-grained move, mix it with the existing moves and benchmark it against the previous configurations to see if you want to keep it.

7.7.2. Custom moves introduction

Instead of reusing the generic `Moves` (such as `ChangeMove`) you can also implement your own `Moves`. Generic and custom `MoveSelectors` can be combined as wanted.

A custom `Move` can be tailored to work to the advantage of your constraints. For example, in examination scheduling, changing the period of an exam A also changes te period of all the exams that need to coincide with exam A.

A custom `Move` is also slightly faster than a generic `Move`. However, it's far more work to implement and much harder to avoid bugs. After implementing a custom `Move`, make sure to turn on `environmentMode` `FULL_ASSERT` to check for score corruptions.

7.7.3. The interface `Move`

Your custom moves must implement the `Move` interface:

```
public interface Move {
```

```
    boolean isMoveDoable(ScoreDirector scoreDirector);

    Move createUndoMove(ScoreDirector scoreDirector);
    void doMove(ScoreDirector scoreDirector);

    Collection<? extends Object> getPlanningEntities();
    Collection<? extends Object> getPlanningValues();

}
```

Let's take a look at the `Move` implementation for 4 queens which moves a queen to a different row:

```
public class RowChangeMove implements Move {

    private Queen queen;
    private Row toRow;

    public RowChangeMove(Queen queen, Row toRow) {
        this.queen = queen;
        this.toRow = toRow;
    }

    // ... see below

}
```

An instance of `RowChangeMove` moves a queen from its current row to a different row.

Planner calls the `doMove(ScoreDirector)` method to do a move. The `Move` implementation must notify the `ScoreDirector` of any changes it make to planning entity's variables:

```
    public void doMove(ScoreDirector scoreDirector) {
          scoreDirector.beforeVariableChanged(queen, "row"); // before changes
 are made to the queen.row
        queen.setRow(toRow);
         scoreDirector.afterVariableChanged(queen, "row"); // after changes are
 made to the queen.row
    }
```

You need to call the methods `scoreDirector.beforeVariableChanged(Object, String)` and `scoreDirector.afterVariableChanged(Object, String)` directly before and after modifying the entity.

> **Note**
>
> You can alter multiple entities in a single move and effectively create a big move (also known as a coarse-grained move).

> **Warning**
>
> A `Move` can only change/add/remove planning entities, it must not change any of the problem facts.

Planner automatically filters out *non doable moves* by calling the `isDoable(ScoreDirector)` method on a move. A *non doable move* is:

- A move that changes nothing on the current solution. For example, moving queen B0 to row 0 is not doable, because it is already there.
- A move that is impossible to do on the current solution. For example, moving queen B0 to row 10 is not doable because it would move it outside the board limits.

In the n queens example, a move which moves the queen from its current row to the same row isn't doable:

```
public boolean isMoveDoable(ScoreDirector scoreDirector) {
    return !ObjectUtils.equals(queen.getRow(), toRow);
}
```

Because we won't generate a move which can move a queen outside the board limits, we don't need to check it. A move that is currently not doable could become doable on the working `Solution` of a later step.

Each move has an *undo move*: a move (normally of the same type) which does the exact opposite. In the example above the undo move of *C0 to C2* would be the move *C2 to C0*. An undo move is created from a `Move`, before the `Move` has been done on the current solution.

```
public Move createUndoMove(ScoreDirector scoreDirector) {
    return new RowChangeMove(queen, queen.getRow());
}
```

Notice that if C0 would have already been moved to C2, the undo move would create the move *C2 to C2*, instead of the move *C2 to C0*.

A solver phase might do and undo the same `Move` more than once. In fact, many solver phases will iteratively do an undo a number of moves to evaluate them, before selecting one of those and doing that move again (without undoing it this time).

A `Move` must implement the `getPlanningEntities()` and `getPlanningValues()` methods. They are used by entity tabu and value tabu respectively. When they are called, the `Move` has already been done.

```
public List<? extends Object> getPlanningEntities() {
    return Collections.singletonList(queen);
}

public Collection<? extends Object> getPlanningValues() {
    return Collections.singletonList(toRow);
}
```

If your `Move` changes multiple planning entities, return all of them in `getPlanningEntities()` and return all their values (to which they are changing) in `getPlanningValues()`.

```
public Collection<? extends Object> getPlanningEntities() {
    return Arrays.asList(leftCloudProcess, rightCloudProcess);
}

public Collection<? extends Object> getPlanningValues() {
    return Arrays.asList(leftCloudProcess.getComputer(), rightCloudProcess.getComputer());
}
```

A `Move` must implement the `equals()` and `hashCode()` methods. 2 moves which make the same change on a solution, should be equal.

```
public boolean equals(Object o) {
    if (this == o) {
        return true;
    } else if (o instanceof RowChangeMove) {
        RowChangeMove other = (RowChangeMove) o;
        return new EqualsBuilder()
                .append(queen, other.queen)
                .append(toRow, other.toRow)
                .isEquals();
    } else {
        return false;
    }
}
```

```
    public int hashCode() {
        return new HashCodeBuilder()
                .append(queen)
                .append(toRow)
                .toHashCode();
    }
```

Notice that it checks if the other move is an instance of the same move type. This `instanceof` check is important because a move will be compared to a move with another move type if you're using more then 1 move type.

It's also recommended to implement the `toString()` method as it allows you to read Planner's logging more easily:

```
    public String toString() {
        return queen + " => " + toRow;
    }
```

Now that we can implement a single custom `Move`, let's take a look at generating such custom moves.

7.7.4. `MoveListFactory`: the easy way to generate custom moves

The easiest way to generate custom moves is by implementing the interface `MoveListFactory`:

```
public interface MoveListFactory {

    List<Move> createMoveList(Solution solution);

}
```

For example:

```
public class RowChangeMoveFactory implements MoveListFactory {

    public List<Move> createMoveList(Solution solution) {
        NQueens nQueens = (NQueens) solution;
        List<Move> moveList = new ArrayList<Move>();
        for (Queen queen : nQueens.getQueenList()) {
            for (Row toRow : nQueens.getRowList()) {
                moveList.add(new RowChangeMove(queen, toRow));
            }
        }
        return moveList;
```

```
    }

}
```

Simple configuration (which can be nested in a `unionMoveSelector` just like any other `MoveSelector`):

```
    <moveListFactory>

org.optaplanner.examples.nqueens.solver.move.factory.RowChangeMoveFactory</
moveListFactoryClass>
    </moveListFactory>
```

Advanced configuration:

```
    <moveListFactory>
      ... <!-- Normal moveSelector properties -->

org.optaplanner.examples.nqueens.solver.move.factory.RowChangeMoveFactory</
moveListFactoryClass>
    </moveListFactory>
```

Because the `MoveListFactory` generates all moves at once in a List<Move>, it does not support `cacheType JUST_IN_TIME`. Therefore, `moveListFactory` uses `cacheType STEP` by default and it scales badly in memory footprint.

7.7.5. `MoveIteratorFactory`: generate custom moves just in time

Use this advanced form to generate custom moves by implementing the interface `MoveIteratorFactory`:

```
public interface MoveIteratorFactory {

    long getSize(ScoreDirector scoreDirector);

    Iterator<Move> createOriginalMoveIterator(ScoreDirector scoreDirector);

    Iterator<Move> createRandomMoveIterator(ScoreDirector scoreDirector, Random workingRandom);

}
```

The method `getSize()` must give an estimation of the size. It doesn't need to be correct. The method `createOriginalMoveIterator` is called if the `selectionOrder is ORIGINAL` or

if it is cached. The method `createRandomMoveIterator` is called for `selectionOrder RANDOM` combined with cacheType `JUST_IN_TIME`.

> **Important**
>
> Don't create a collection (list, array, map, set) of `Moves` when creating the `Iterator<Move>`: the whole purpose of `MoveIteratorFactory` over `MoveListFactory` is giving you the ability to create a `Move` just in time in the `Iterator`'s method `next()`.

Simple configuration (which can be nested in a `unionMoveSelector` just like any other `MoveSelector`):

```
<moveIteratorFactory>
  <moveIteratorFactoryClass>...</moveIteratorFactoryClass>
</moveIteratorFactory>
```

Advanced configuration:

```
<moveIteratorFactory>
  ... <!-- Normal moveSelector properties -->
  <moveIteratorFactoryClass>...</moveIteratorFactoryClass>
</moveIteratorFactory>
```

7.7.6. Move generation through DRL

Not yet supported.

Chapter 8. Construction heuristics

8.1. Overview

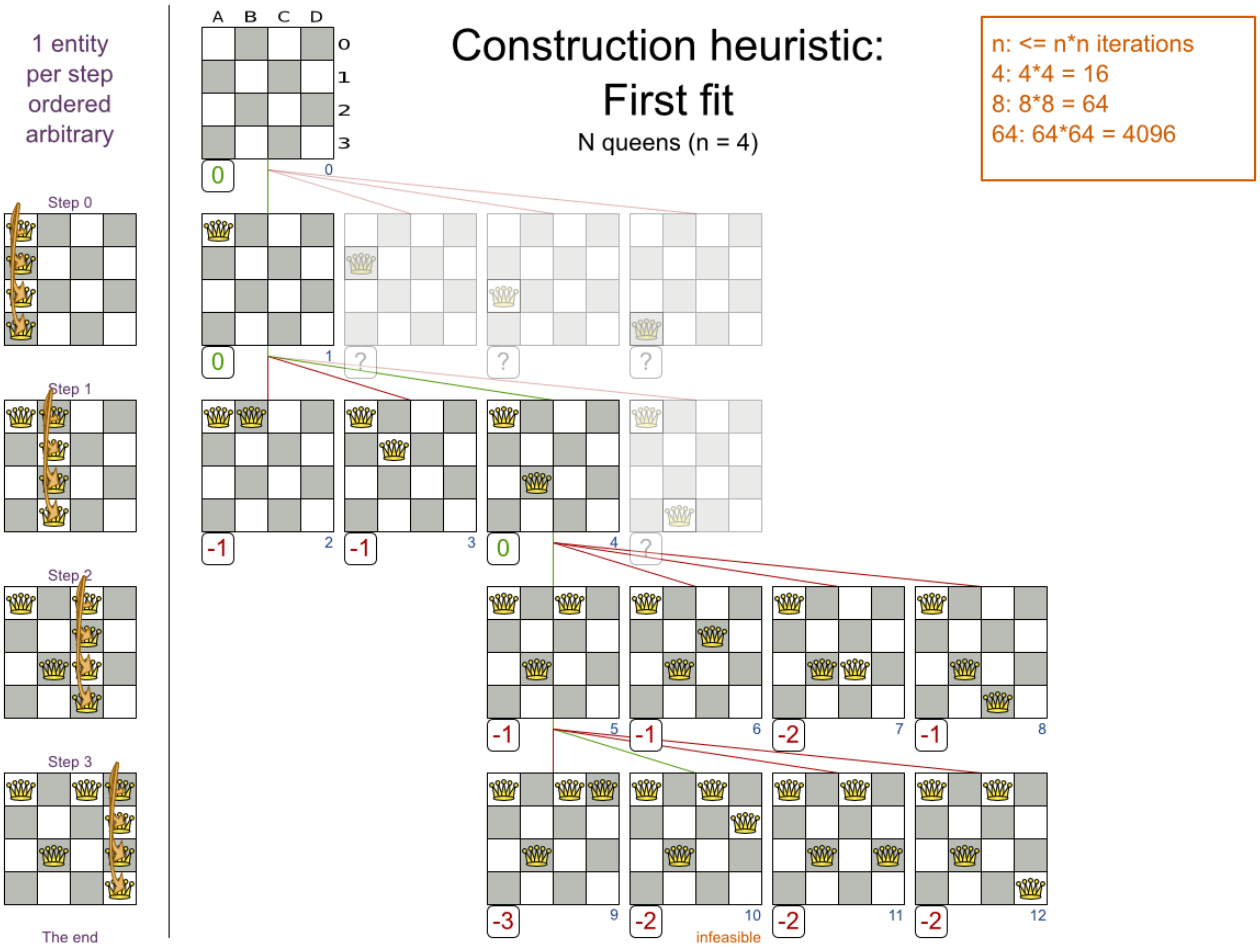
A construction heuristic builds a pretty good initial solution in a finite length of time. Its solution isn't always feasible, but it finds it fast and metaheuristics can finish the job.

Construction heuristics terminate automatically, so there's usually no need to configure a `Termination` on the construction heuristic phase specifically.

8.2. First Fit

8.2.1. Algorithm description

The *First Fit* algorithm cycles through all the planning entities (in default order), initializing 1 planning entity at a time. It assigns the planning entity to the best available planning value, taking the already initialized planning entities into account. It terminates when all planning entities have been initialized. It never changes a planning entity after it has been assigned.



Notice that it starts with putting `Queen` A into row 0 (and never moving it later), which makes it impossible to reach the optimal solution. Suffixing this construction heuristic with metaheurstics can remedy that.

8.2.2. Configuration

Configure this `SolverPhase`:

```
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT</constructionHeuristicType>
  <!-- Speedup that can be applied to most, but not all use cases: -->
  <!--forager-->
    <!-- <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType> -->
  <!--/forager-->
</constructionHeuristic>
```

Note

The `pickEarlyType` of `FIRST_NON_DETERIORATING_SCORE` is a big speedup: for an entity, it picks the first move for which the score does not deteriorate the last step score, ignoring all subsequent moves. It should be applied when initializing a planning entity cannot improve the last step score: it can only make the score lower or equal. So if:

- There are no positive constraints.
- And there is no negative constraint that can stop being broken by adding a planning entity (except if another negative constraint gets broken which outweighs the first negative constraint).

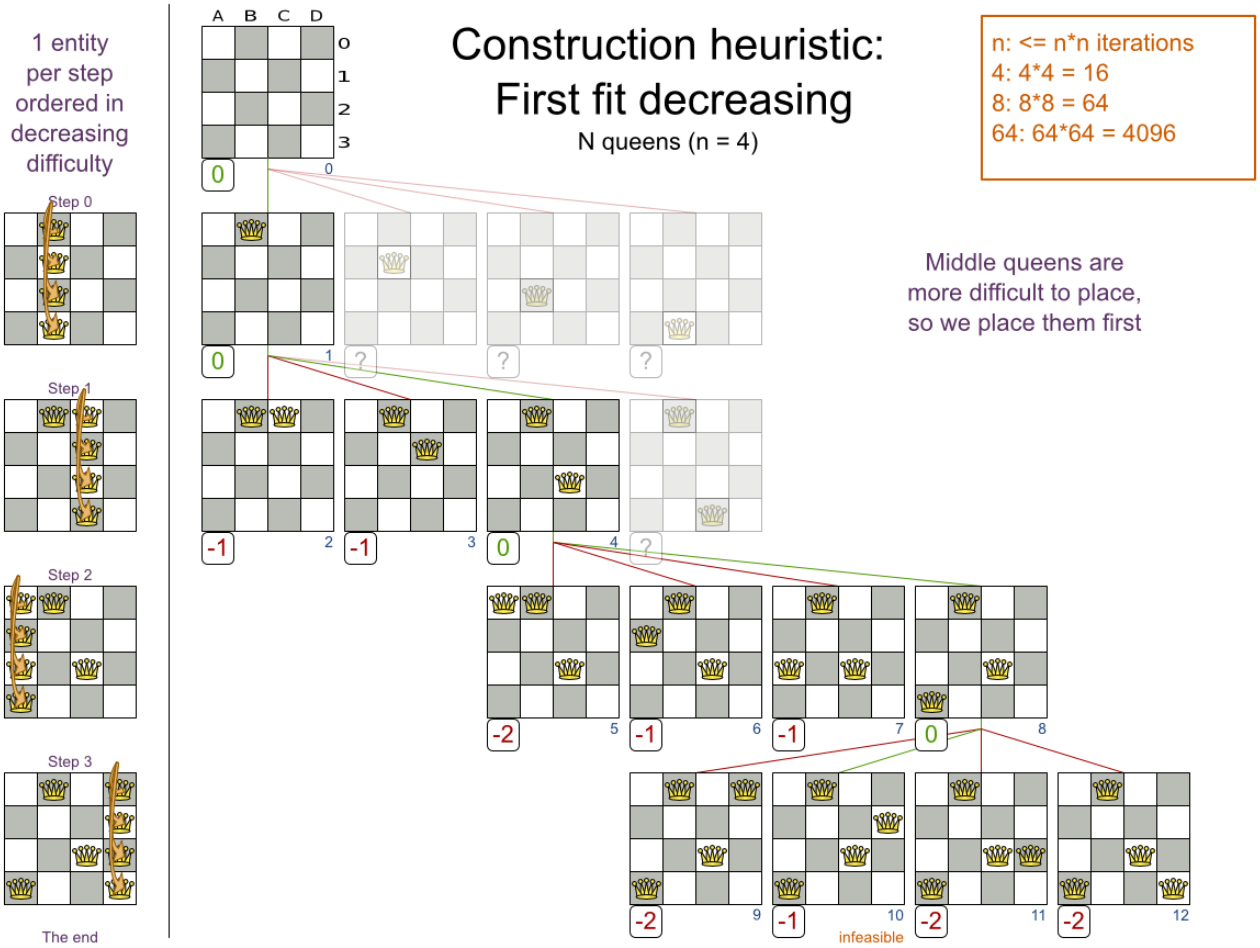
If that is not the case, then it might still be good to apply it in some cases. Use the `Benchmarker` to decide.

8.3. First Fit Decreasing

8.3.1. Algorithm description

Like `First Fit`, but assigns the more difficult planning entities first, because they are less likely to fit in the leftovers. So it sorts the planning entities on decreasing difficulty.

Requires the model to support *planning entity difficulty comparison*.



> **Note**
>
> One would expect that this algorithm always performs better than `First Fit`. That's not always the case, but usually is.

8.3.2. Configuration

Configure this `SolverPhase`:

```
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT_DECREASING</constructionHeuristicType>
  <!-- Speedup that can be applied to most, but not all use cases: -->
  <!--forager-->
    <!-- <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType> -->
  <!--/forager-->
</constructionHeuristic>
```

8.4. Best Fit

8.4.1. Algorithm description

Like `First Fit`, but uses the weaker planning values first, because the strong planning values are more likely to be able to accommodate later planning entities. So it sorts the planning values on increasing strength.

Requires the model to support *planning value strength comparison*.

> **Note**
>
> One would expect that this algorithm always performs better than `First Fit`. That's not always the case.

8.4.2. Configuration

Configure this `SolverPhase`:

```
<constructionHeuristic>
  <constructionHeuristicType>BEST_FIT</constructionHeuristicType>
  <!-- Speedup that can be applied to most, but not all use cases: -->
  <!--forager-->
    <!-- <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType> -->
  <!--/forager-->
</constructionHeuristic>
```

8.5. Best Fit Decreasing

8.5.1. Algorithm description

Combines `First Fit Decreasing` and `Best Fit`. So it sorts the planning entities on decreasing difficulty and the planning values on increasing strength.

Requires the model to support *planning entity difficulty comparison* and *planning value strength comparison*.

> **Note**
>
> One would expect that this algorithm always performs better than `First Fit`, `First Fit Decreasing` and `Best Fit`. That's not always the case.

8.5.2. Configuration

Configure this `SolverPhase`:

```
<constructionHeuristic>
  <constructionHeuristicType>BEST_FIT_DECREASING</constructionHeuristicType>
  <!-- Speedup that can be applied to most, but not all use cases: -->
  <!--forager-->
    <!-- <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType> -->
  <!--/forager-->
</constructionHeuristic>
```

8.6. Cheapest insertion

8.6.1. Algorithm description

TODO

8.6.2. Configuration

TODO Not implemented yet.

8.7. Regret insertion

8.7.1. Algorithm description

TODO

8.7.2. Configuration

TODO Not implemented yet.

Chapter 9. Local search

9.1. Overview

Local search starts from an initial solution and evolves that single solution into a mostly better and better solution. It uses a single search path of solutions, not a search tree. At each solution in this path it evaluates a number of moves on the solution and applies the most suitable move to take the step to the next solution. It does that for a high number of iterations until it's terminated (usually because its time has run out).

Local search acts a lot like a human planner: it uses a single search path and moves facts around to find a good feasible solution. Therefore it's pretty natural to implement.

Local search often needs to start from an initialized solution, therefore it's recommended to configure a construction heuristic solver phase before it.

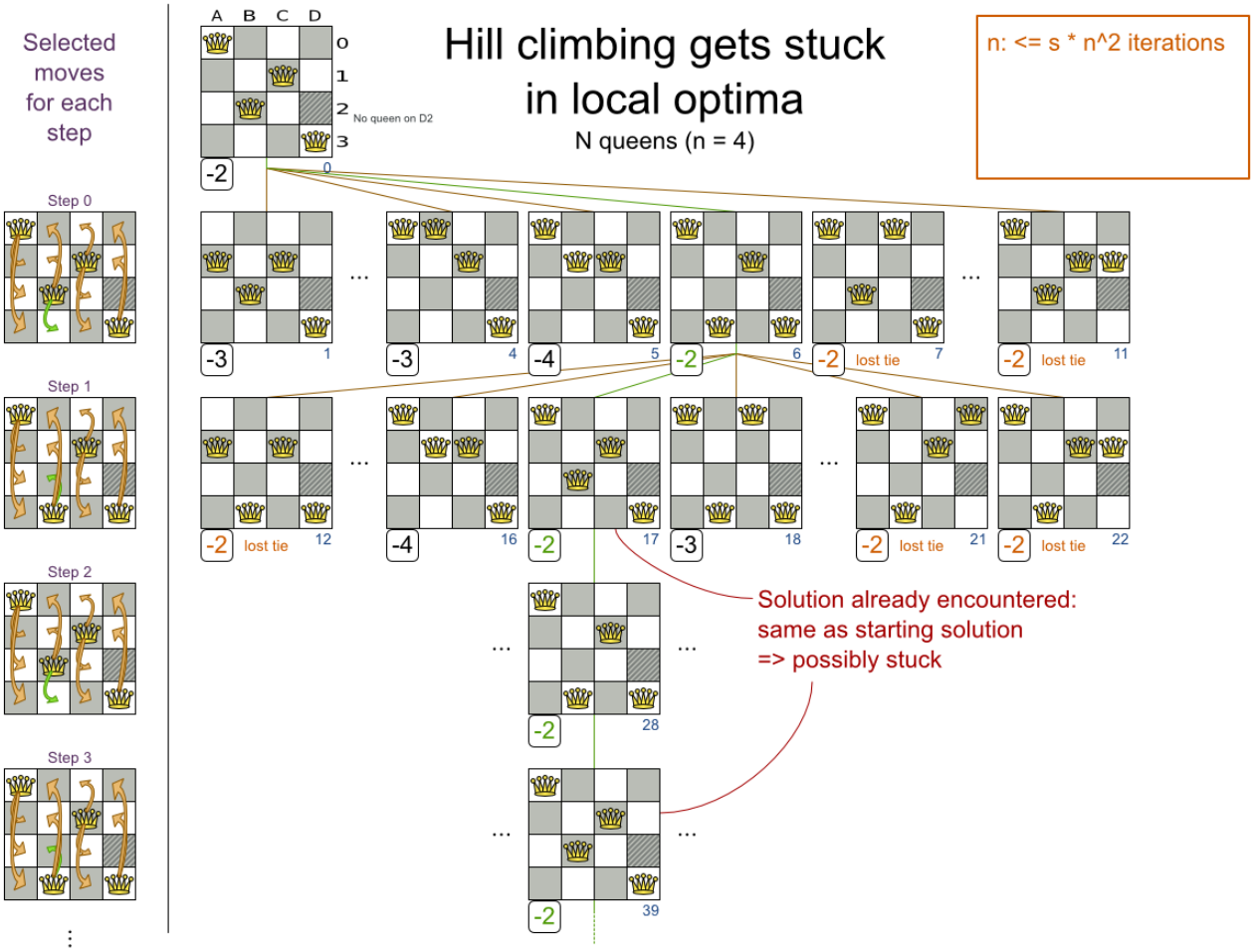
9.2. Hill Climbing (Simple Local Search)

9.2.1. Algorithm description

Hill Climbing tries all selected moves and then takes the best move, which is the move which leads to the solution with the highest score. That best move is called the step move. From that new solution, it again tries all selected moves and takes the best move and continues like that iteratively. If multiple selected moves tie for the best move, one of them is randomly chosen as the best move.

Notice that one a queen has moved, it can be moved again later. This is a good thing, because in an NP-complete problem it's impossible to predict what will be the optimal final value for a planning variable.

Hill Climbing can easily get stuck in a local optima:

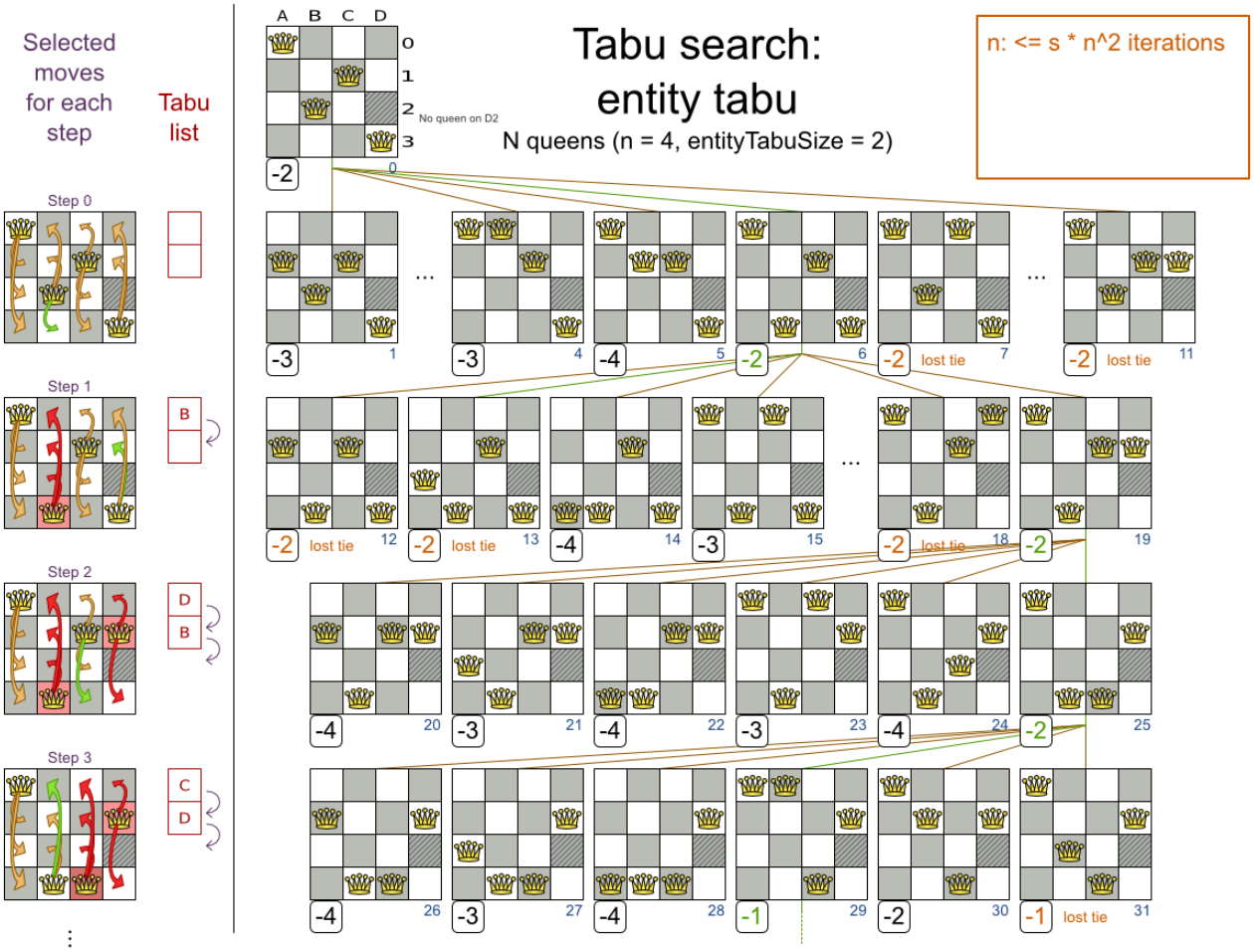


Improvements upon hill climbing (such as tabu search and simulated annealing) address the problem of being stuck in local optima.

9.3. Tabu Search

9.3.1. Algorithm description

Tabu Search works like hill climbing, but it maintains a tabu list to avoid getting stuck in local optima. The tabu list holds recently used objects that are *taboo* to use for now. Moves that involve an object in the tabu list, are not accepted. The tabu list objects can be anything related to the move, such as the planning entity, planning value, move, solution, ... Here's an example with entity tabu for 4 queens, so the queens are put in the tabu list:



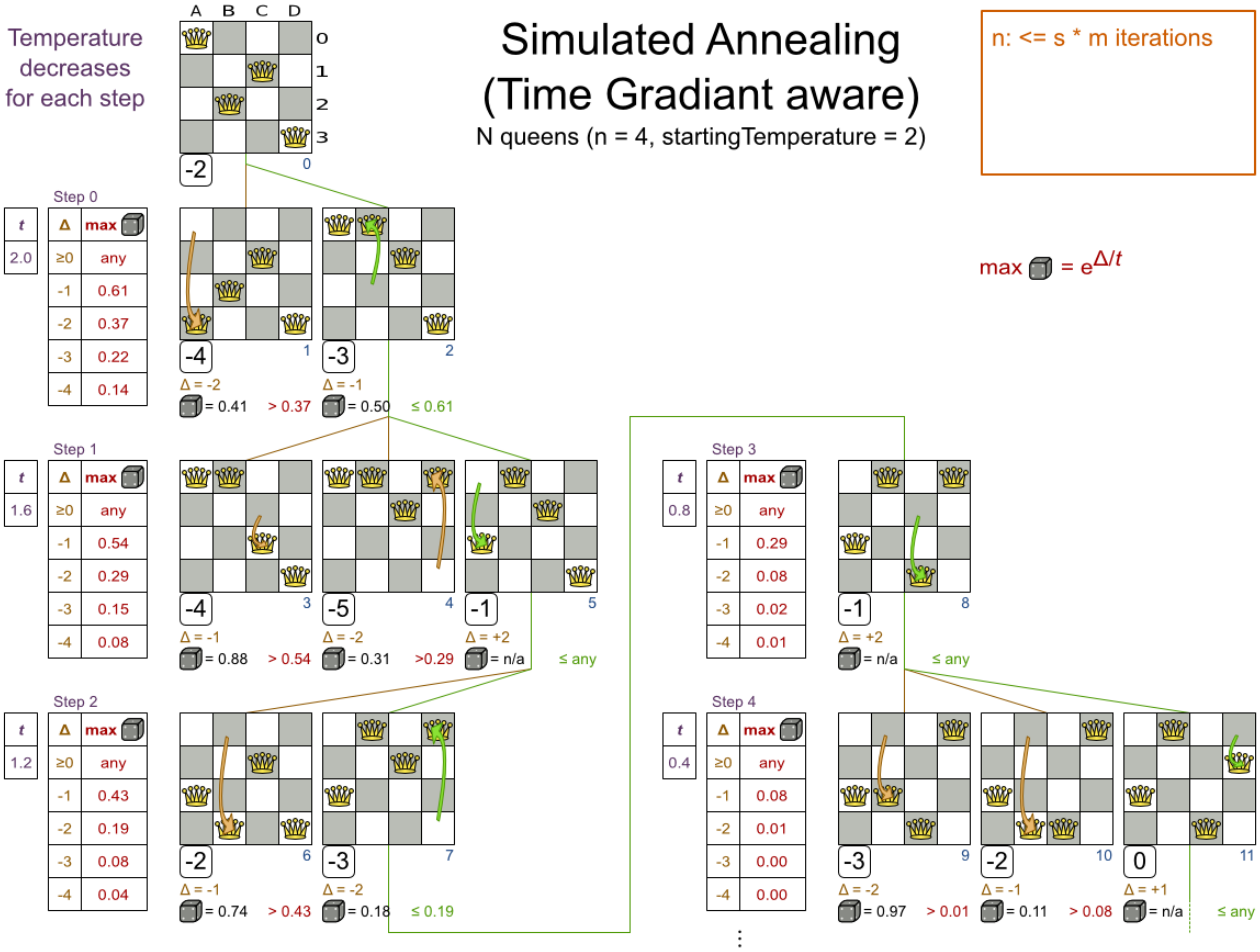
Scientific paper: *Tabu Search - Part 1 and Part 2* by Fred Glover (1989 - 1990)

See Tabu Search acceptor below.

9.4. Simulated Annealing

9.4.1. Algorithm description

Simulated Annealing evaluates only a few moves per step, so it steps quickly. In the classic implementation, the first accepted move is the winning step. A move is accepted if it doesn't decrease the score or - in case it does decrease the score - if passes a random check. The chance that a decreasing move passes the random check decreases relative to the size of the score decrement and the time the phase has been running (which is represented as the temperature).



See Simulated Annealing acceptor below.

9.5. Late Acceptance

9.5.1. Algorithm description

Late Acceptance (also known as Late Acceptance Hill Climbing) also evaluates only a few moves per step. A move is accepted if does not decrease the score, or if it leads to a score that is at least the late score (which is the winning score of a fixed number of steps ago).

Scientific paper: *The Late Acceptance Hill-Climbing Heuristic by Edmund K. Burke, Yuri Bykov (2012)* [www.cs.stir.ac.uk/research/publications/techreps/pdf/TR192.pdf]

See Late Acceptance acceptor below.

9.6. Late Simulated Annealing (experimental)

9.6.1. Algorithm description

Late Simulated Annealing combines the best characteristics of Simulated Annealing and Late Acceptance. A move is accepted if does not decrease the score, or if it leads to a score that is at least the late score, or - failing those criteria - if it passes a random check. The chance that a decreasing move passes the random check decreases relative to the size of the score decrement and the late score.

See Late Simulated Annealing acceptor below.

9.7. About neighborhoods, moves and steps

9.7.1. Move generation tips

At each solution, local search will try all possible moves and pick the best move to change to the next solution. It's up to you to generate those moves. Let's take a look at all the possible moves on the starting solution of 4 queens:

It's highly recommended that you verify all solutions are connected by your move set. This means that by combining a finite number of moves you can reach any solution from any solution. Otherwise you're already excluding solutions at the start. Especially if you're using only big moves, you should check it. Just because big moves outperform small moves in a short test run, it doesn't mean that they will outperform them in a long test run.

You can mix different move types. Usually you're better off preferring small (fine-grained) moves over big (course-grained) moves because the score delta calculation will pay off more. However, as the traveling tournament example proves, if you can remove a hard constraint by using a certain set of big moves, you can win performance and scalability. A big moves version could evaluate a lot less unfeasible solutions, which enables it to outperform and outscale a small moves version.

9.7.2. A step

A step is the winning move. The local search solver tries every move on the current solution and picks the best accepted move as the step:

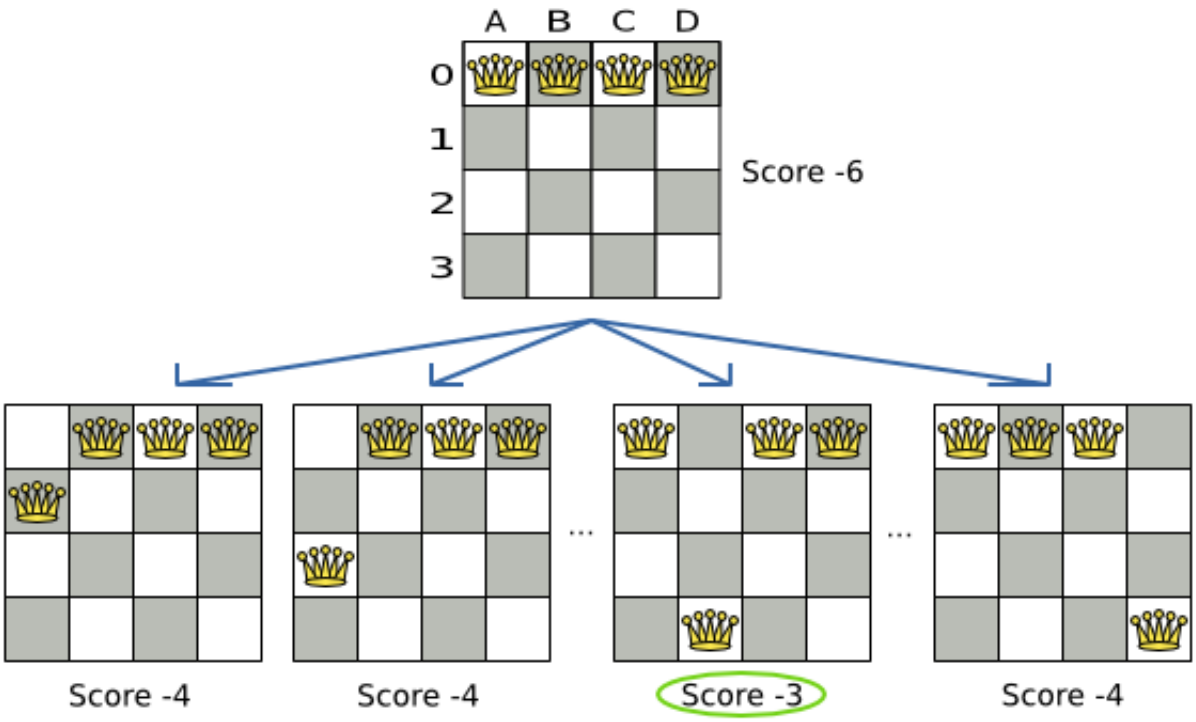


Figure 9.1. Decide the next step at step 0 (4 queens example)

Because the move *B0 to B3* has the highest score (`-3`), it is picked as the next step. Notice that *C0 to C3* (not shown) could also have been picked because it also has the score `-3`. If multiple moves have the same highest score, one is picked randomly, in this case *B0 to B3*.

The step is made and from that new solution, the local search solver tries all the possible moves again, to decide the next step after that. It continually does this in a loop, and we get something like this:

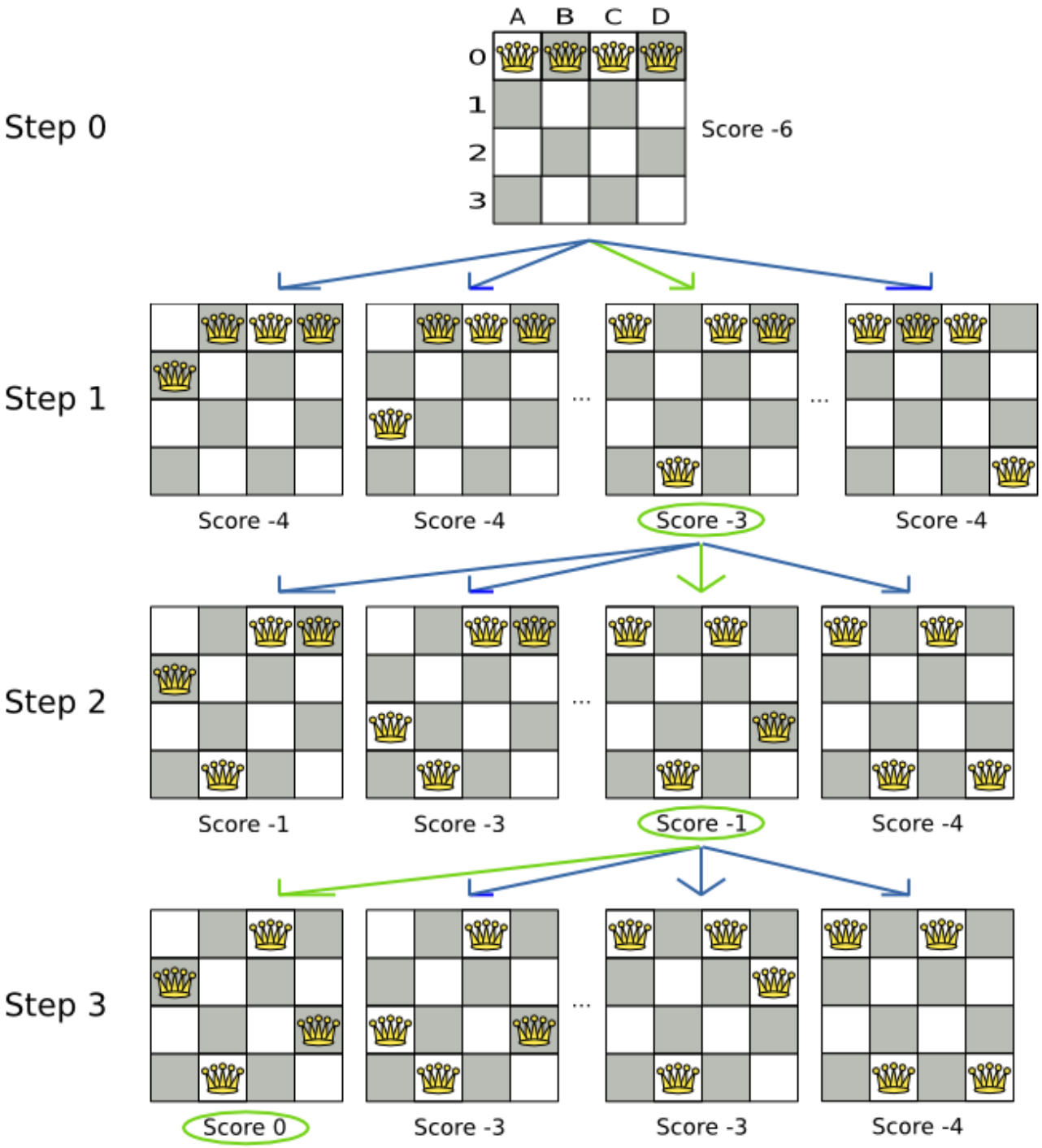


Figure 9.2. All steps (4 queens example)

Notice that the local search solver doesn't use a search tree, but a search path. The search path is highlighted by the green arrows. At each step it tries all possible moves, but unless it's the step, it doesn't investigate that solution further. This is one of the reasons why local search is very scalable.

As you can see, the local search solver solves the 4 queens problem by starting with the starting solution and make the following steps sequentially:

1. *B0 to B3*

2. *D0 to B2*

3. *A0 to B1*

If we turn on `debug` logging for the category `org.optaplanner`, then those steps are shown into the log:

```
INFO  Solving started: time spend (0), score (-6), new best score (-6), random
 seed (0).
DEBUG      Step index (0), time spend (20), score (-3), new best score (-3),
 accepted/selected move count (12/12) for picked step (col1@row0 => row3).
DEBUG      Step index (1), time spend (31), score (-1), new best score (-1),
 accepted/selected move count (12/12) for picked step (col0@row0 => row1).
DEBUG       Step index (2), time spend (40), score (0), new best score (0),
 accepted/selected move count (12/12) for picked step (col3@row0 => row2).
INFO  Phase (0) localSearch ended: step total (3), time spend (41), best score (0).
INFO  Solving ended: time spend (41), best score (0), average calculate count
 per second (1780).
```

Notice that the logging uses the `toString()` method of our `Move` implementation: `col1@row0 => row3`.

The local search solver solves the 4 queens problem in 3 steps, by evaluating only 37 possible solutions (3 steps with 12 moves each + 1 starting solution), which is only fraction of all 256 possible solutions. It solves 16 queens in 31 steps, by evaluating only 7441 out of 18446744073709551616 possible solutions. Note: with construction heuristics it's even a lot more efficient.

9.7.3. Getting stuck in local optima

A *hill climber* always takes improving moves. This may seem like a good thing, but it's not. It suffers from a number of problems:

- It can get stuck in a local optimum. For example if it reaches a solution X with a score -1 and there is no improving move, it is forced to take a next step that leads to a solution Y with score -2, after that however, it's very real that it will pick the step back to solution X with score -1. It will then start looping between solution X and Y.
- It can start walking in its own footsteps, picking the same next step at every step.

Of course OptaPlanner implements better local searches, such as *tabu search* and *simulated annealing* which can avoid these problems. We recommend to never use hill climbing, unless you're absolutely sure there are no local optima in your planning problem.

9.8. Deciding the next step

The local search solver decides the next step with the aid of 3 configurable components:

- A `MoveSelector` which selects the possible moves of the current solution. See the chapter about Move and neighborhood selection.
- An *acceptor* which filters out unacceptable moves. It can also weigh a move it accepts.
- A *forager* which gathers all accepted moves and picks the next step from them.

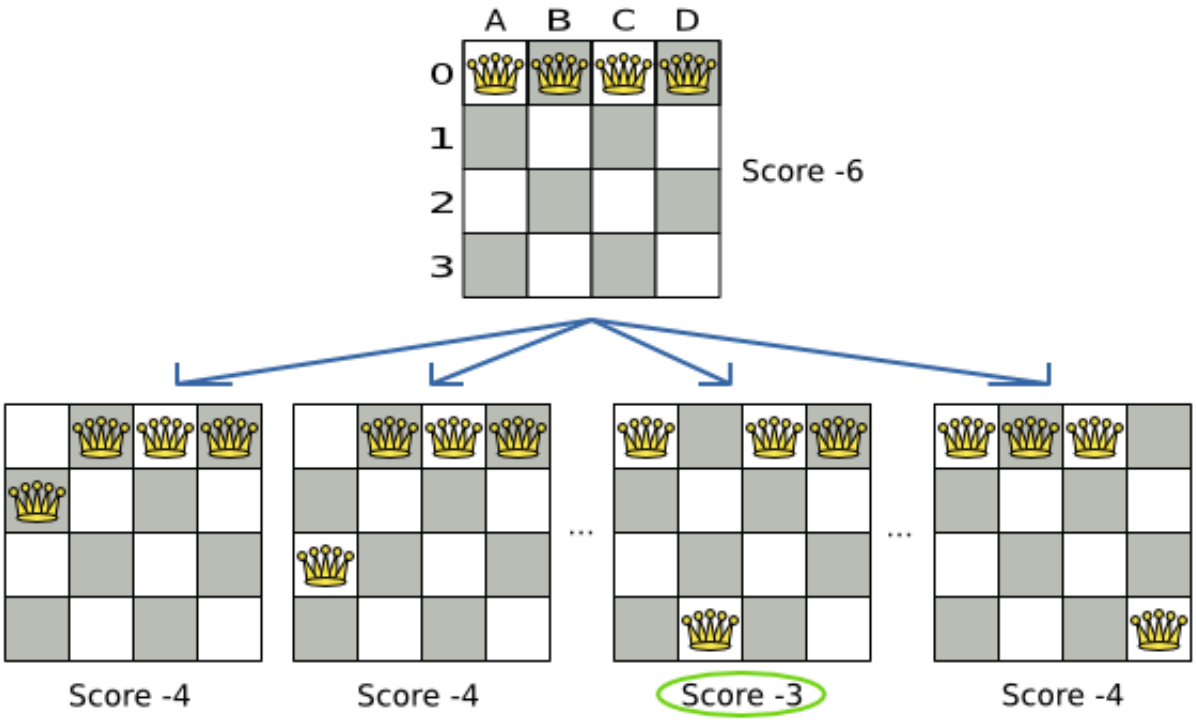


Figure 9.3. Decide the next step at step 0 (4 queens example)

In the above example the selector generated the moves shown with the blue lines, the acceptor accepted all of them and the forager picked the move *B0 to B3*.

If we turn on `trace` logging for the category `org.optaplanner`, then the decision making is shown in the log:

```
INFO  Solver started: time spend (0), score (-6), new best score (-6), random
 seed (0).
TRACE         Move index (0) not doable, ignoring move (col0@row0 => row0).
TRACE          Move index (1), score (-4), accepted (true) for move (col0@row0
 => row1).
```

```
TRACE          Move index (2), score (-4), accepted (true) for move (col0@row0
 => row2).
TRACE          Move index (3), score (-4), accepted (true) for move (col0@row0
 => row3).
...
TRACE          Move index (6), score (-3), accepted (true) for move (col1@row0
 => row3).
...
TRACE          Move index (9), score (-3), accepted (true) for move (col2@row0
 => row3).
...
TRACE         Move index (12), score (-4), accepted (true) for move (col3@row0
 => row3).
DEBUG      Step index (0), time spend (6), score (-3), new best score (-3),
 accepted/selected move count (12/12) for picked step (col1@row0 => row3).
...
```

Because the last solution can degrade (especially in tabu search and simulated annealing), the `Solver` remembers the best solution it has encountered through the entire search path. Each time the current solution is better than the last best solution, the current solution is cloned and referenced as the new best solution.

9.8.1. Acceptor

An acceptor is used (together with a forager) to active tabu search, simulated annealing, great deluge, ... For each move it checks whether it is accepted or not.

You can implement your own `Acceptor`, although the build-in acceptors should suffice for most needs. You can also combine multiple acceptors.

9.8.1.1. Tabu Search acceptor

When tabu search takes steps it creates tabu's. It does not accept a move as the next step if that move breaks tabu. OptaPlanner implements several tabu types:

- *Solution tabu* makes recently visited solutions tabu. It does not accept a move that leads to one of those solutions. If you can spare the memory, don't be cheap on the tabu size.

```
    <acceptor>
        <solutionTabuSize>1000</solutionTabuSize>
    </acceptor>
```

- *Move tabu* makes recent steps tabu. It does not accept a move equal to one of those steps.

```
    <acceptor>
```

```
        <moveTabuSize>7</moveTabuSize>
    </acceptor>
```

- *Undo move tabu* makes the undo move of recent steps tabu.

```
    <acceptor>
        <undoMoveTabuSize>7</undoMoveTabuSize>
    </acceptor>
```

- *Planning entity tabu* makes the planning entities of recent steps tabu. For example, for N queens it makes the recently moved queens tabu. It's recommended to start with this tabu type.

```
    <acceptor>
        <entityTabuSize>7</entityTabuSize>
    </acceptor>
```

To avoid hard coding the tabu size, configure a tabu ratio, relative to the number of entities, for example 2%:

```
    <acceptor>
        <entityTabuRatio>0.02</entityTabuRatio>
    </acceptor>
```

- *Planning value tabu* makes the planning values of recent steps tabu. For example, for N queens it makes the recently moved to rows tabu.

```
    <acceptor>
        <valueTabuSize>7</valueTabuSize>
    </acceptor>
```

You can even combine tabu types:

```
    <acceptor>
        <entityTabuSize>7</entityTabuSize>
        <valueTabuSize>3</valueTabuSize>
    </acceptor>
```

If you pick a too small tabu size, your solver can still get stuck in a local optimum. On the other hand, with the exception of solution tabu, if you pick a too large tabu size, your solver can get stuck by bouncing of the walls. Use the benchmarker to fine tweak your configuration.

A tabu search acceptor should be combined with a high `acceptedCountLimit`, such as `1000`.

```
    <forager>
        <acceptedCountLimit>1000</acceptedCountLimit>
    </forager>
```

9.8.1.2. Simulated Annealing acceptor

Simulated Annealing does not always pick the move with the highest score, neither does it evaluate many moves per step. At least at first. Instead, it gives non improving moves also a chance to be picked, depending on its score and the time gradient of the `Termination`. In the end, it gradually turns into a hill climber, only accepting improving moves.

In many use cases, simulated annealing surpasses tabu search. By changing a few lines of configuration, you can easily switch from tabu search to simulated annealing and back.

Start with a `simulatedAnnealingStartingTemperature` set to the maximum score delta a single move can cause. Use the `Benchmarker` to tweak the value.

```
    <acceptor>
                            <simulatedAnnealingStartingTemperature>2hard/100soft</
simulatedAnnealingStartingTemperature>
    </acceptor>
    <forager>
        <acceptedCountLimit>4</acceptedCountLimit>
    </forager>
```

A simulated annealing acceptor should be combined with a low `acceptedCountLimit`. The classic algorithm uses an `acceptedCountLimit` of `1`, but often `4` performs better.

You can even combine it with a tabu acceptor at the same time. Use a lower tabu size than in a pure tabu search configuration.

```
    <acceptor>
                                    <simulatedAnnealingStartingTemperature>10.0</
simulatedAnnealingStartingTemperature>
      <entityTabuSize>5</entityTabuSize>
    </acceptor>
    <forager>
        <acceptedCountLimit>4</acceptedCountLimit>
    </forager>
```

This differs from phasing, another powerful technique, where first simulated annealing is used, followed by tabu search.

9.8.1.3. Late Acceptance acceptor

Late Acceptance accepts any move that has a score which is higher than the best score of a number of steps ago. That number of steps is the `lateAcceptanceSize`.

```
<acceptor>
  <lateAcceptanceSize>500</lateAcceptanceSize>
</acceptor>
<forager>
    <acceptedCountLimit>4</acceptedCountLimit>
</forager>
```

9.8.1.4. Late Simulated Annealing acceptor (experimental)

Late Simulated Annealing accepts any move that has a score which is higher than the best score of a number of steps ago and has a chance to accept any move below that score too. That number of steps is the `lateSimulatedAnnealingSize`.

```
<acceptor>
  <lateSimulatedAnnealingSize>100</lateSimulatedAnnealingSize>
</acceptor>
<forager>
    <acceptedCountLimit>1000</acceptedCountLimit>
</forager>
```

9.8.2. Forager

A forager gathers all accepted moves and picks the move which is the next step. Normally it picks the accepted move with the highest score. If several accepted moves have the highest score, one is picked randomly.

You can implement your own `Forager`, although the build-in forager should suffice for most needs.

9.8.2.1. Accepted count limit

When there are many possible moves, it becomes inefficient to evaluate all of them at every step. To evaluate only a random subset of all the moves, use:

- An `acceptedCountLimit` integer, which specifies how many accepted moves should be evaluated during each step. By default, all accepted moves are evaluated at every step.

```
<forager>
```

```
    <acceptedCountLimit>1000</acceptedCountLimit>
  </forager>
```

Unlike the n queens problem, real world problems require the use of `acceptedCountLimit`. Start from an `acceptedCountLimit` that takes a step in less then 2 seconds. Turn on INFO logging to see the step times. Use the `Benchmarker` to tweak the value.

> **Important**
>
> With a low `acceptedCountLimit` it is recommended to avoid using `selectionOrder` SHUFFLED because the shuffling generates a random number for every element in the selector, taking up a lot of time, but only a few are actually selected.

9.8.2.2. Pick early type

A forager can pick a move early during a step, ignoring subsequent selected moves. There are 3 pick early types:

- `NEVER`: A move is never picked early: all accepted moves are evaluated that the selection allows. This is the default.

```
    <forager>
        <pickEarlyType>NEVER</pickEarlyType>
    </forager>
```

- `FIRST_BEST_SCORE_IMPROVING`: Pick the first accepted move that improves the best score. If none improve the best score, it behaves exactly like the pickEarlyType NEVER.

```
    <forager>
        <pickEarlyType>FIRST_BEST_SCORE_IMPROVING</pickEarlyType>
    </forager>
```

- `FIRST_LAST_STEP_SCORE_IMPROVING`: Pick the first accepted move that improves the last step score. If none improve the last step score, it behaves exactly like the pickEarlyType NEVER.

```
    <forager>
        <pickEarlyType>FIRST_LAST_STEP_SCORE_IMPROVING</pickEarlyType>
    </forager>
```

9.9. Using a custom Termination, MoveSelector, EntitySelector, ValueSelector or Acceptor

You can plug in a custom `Termination`, `MoveSelector`, `EntitySelector`, `ValueSelector` or `Acceptor` by extending the abstract class and also the related `*Config` class.

For example, to use a custom `MoveSelector`, extend the `AbstractMoveSelector` class, extend the `MoveSelectorConfig` class and configure it in the solver configuration.

> **Note**
>
> It's not possible to directly inject a `Termination`, ... instance, instead of also extend a `Config` class because:
>
> - A `SolverFactory` can build multiple `Solver` instances, which each require a distinct `Termination`, ... instance.
> - A solver configuration needs to be serializable to and from XML. This makes benchmarking with `PlannerBenchmark` particularly easy because you can configure different `Solver` variants in XML.
> - A `Config` class is often easier and clearer to configure. For example: `TerminationConfig` translates `maximumMinutesSpend` and `maximumSecondsSpend` into `maximumTimeMillisSpend`.

If you build a better implementation that's not domain specific, consider contributing it back as a pull request on github and we'll optimize it and take it along in future refactors.

Chapter 10. Evolutionary algorithms

10.1. Overview

Evolutionary algorithms work on a population of solutions and evolve that population.

10.2. Evolutionary Strategies

This algorithm has not been implemented yet.

10.3. Genetic algorithms

This algorithm has not been implemented yet.

Chapter 11. Hyperheuristics

11.1. Overview

A hyperheuristic automates the decision which heuristic(s) to use on a specific data set.

A future version of Planner will have native support for hyperheuristics. Meanwhile, it's pretty easy to implement it yourself: Based on the size or difficulty of a data set (which is a criterion), use a different Solver configuration (or adjust the default configuration using the Solver configuration API). The benchmark report can help to identify such criteria.

Chapter 12. Exact methods

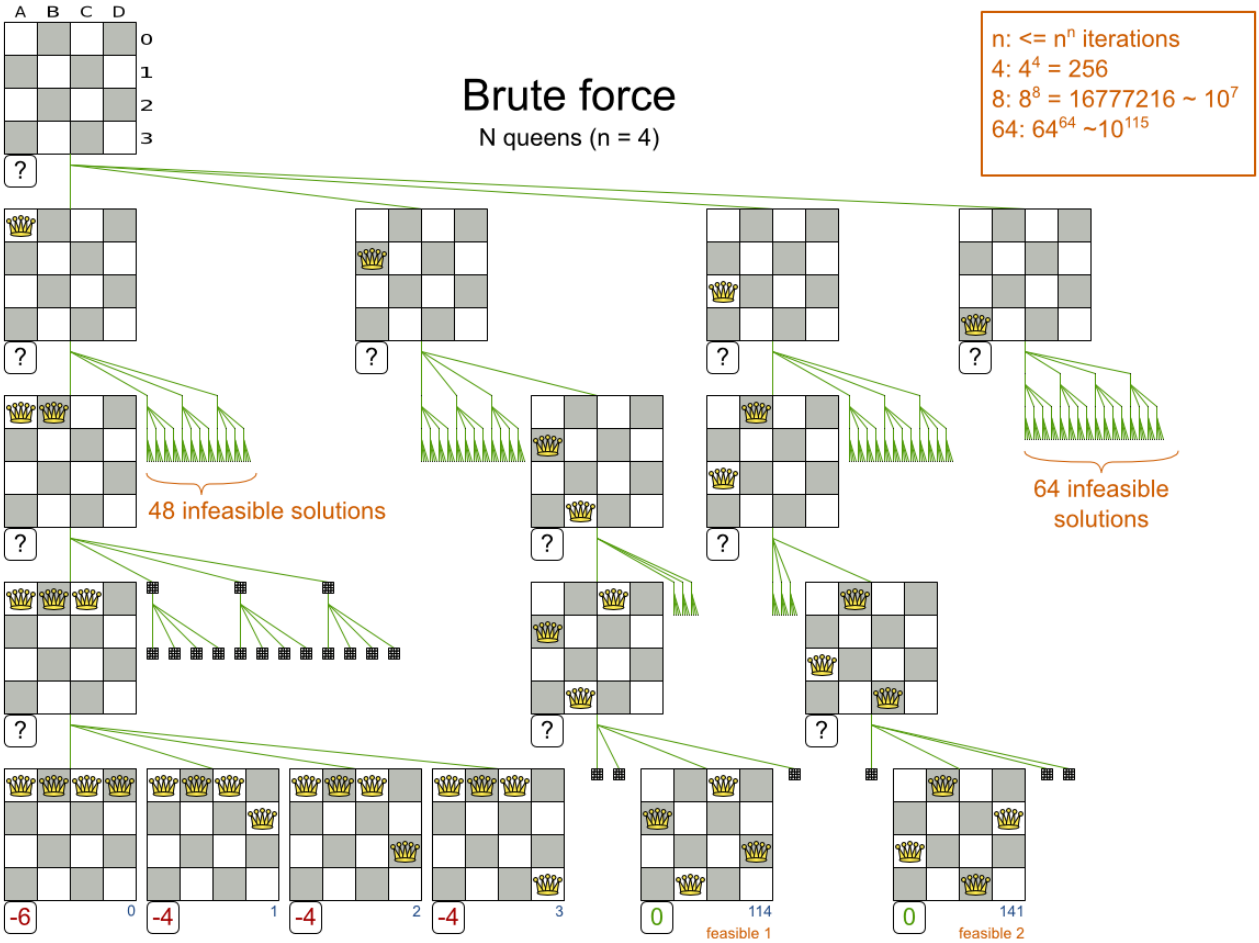
12.1. Overview

Exact methods will always find the global optimum and recognize it too. That being said, they don't scale (not even beyond toy data sets) and are therefore mostly useless.

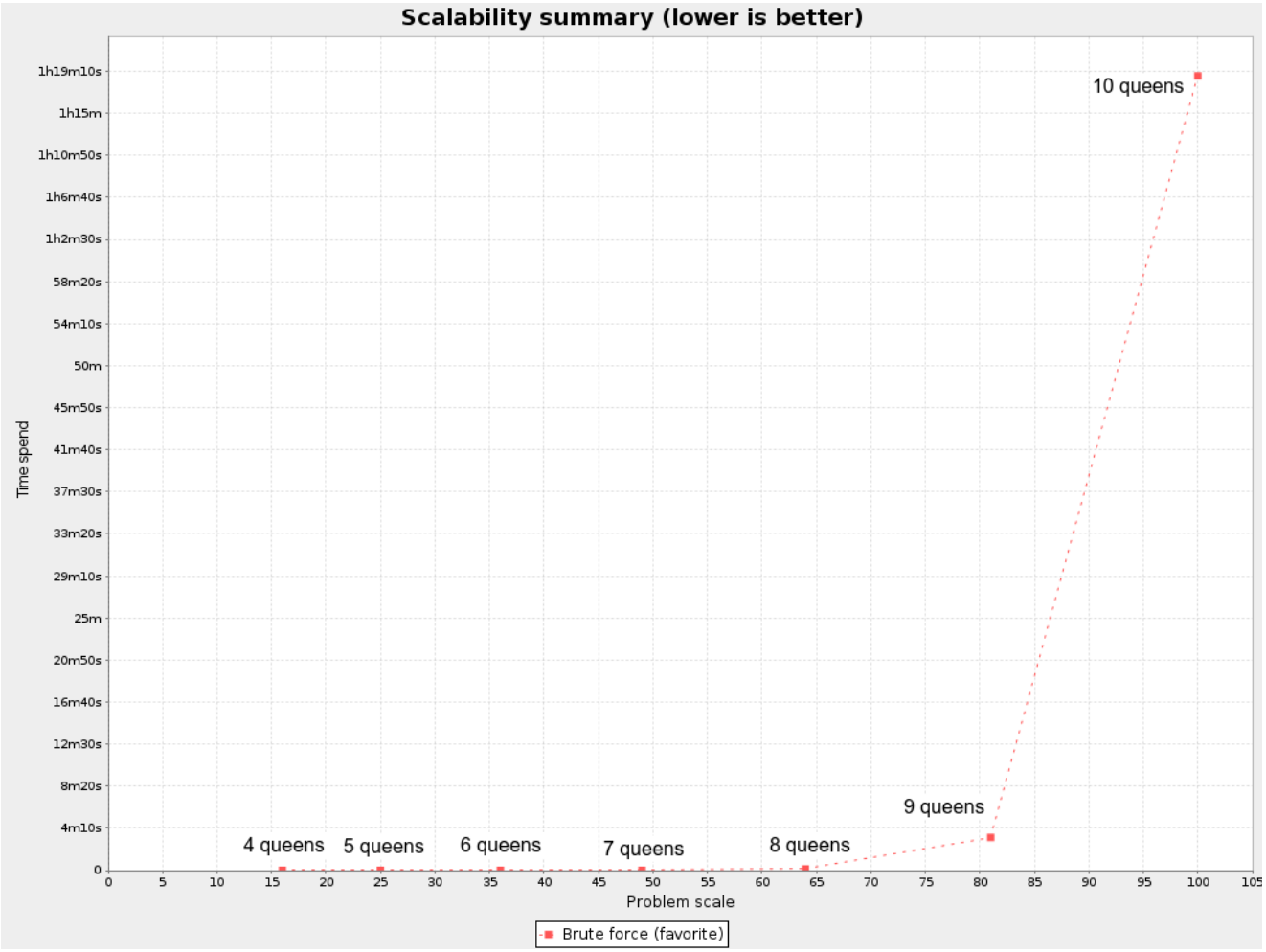
12.2. Brute Force

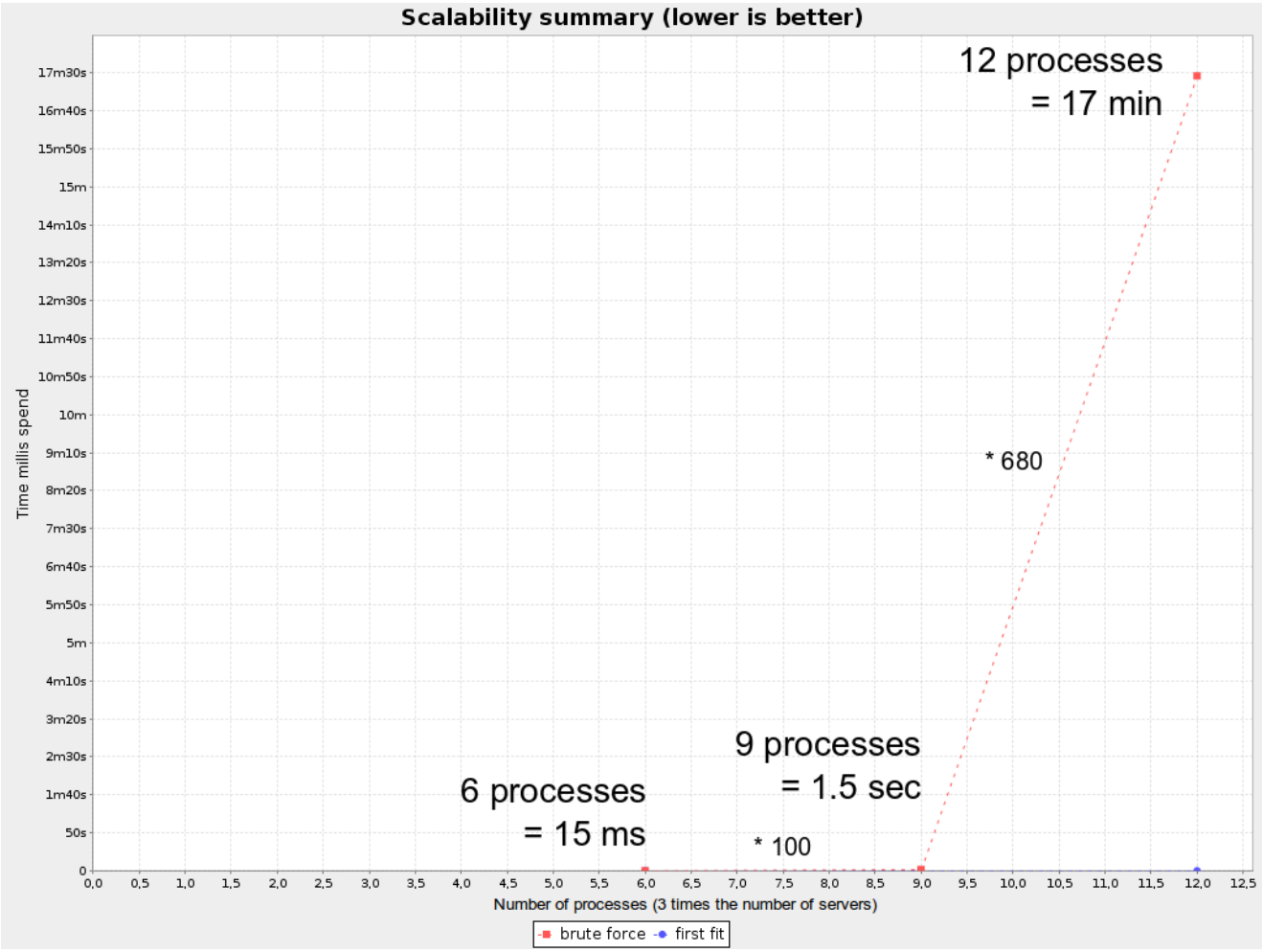
12.2.1. Algorithm description

The Brute Force algorithm creates and evaluates every possible solution.



Notice that it creates a search tree that explodes as the problem size increases. **Brute Force is mostly unusable for a real-world problem due to time limitations**, as proven by these scalability graphs from the benchmarker:


Scalability summary (lower is better)
Time spend
1h19m10s
1h15m
1h10m50s
1h6m40s
1h2m30s
58m20s
54m10s
50m
45m50s
41m40s
37m30s
33m20s
29m10s
25m
20m50s
16m40s
12m30s
8m20s
4m10s
0
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105
Problem scale
4 queens
5 queens
6 queens
7 queens
8 queens
9 queens
10 queens
Brute force (favorite)



12.2.2. Configuration

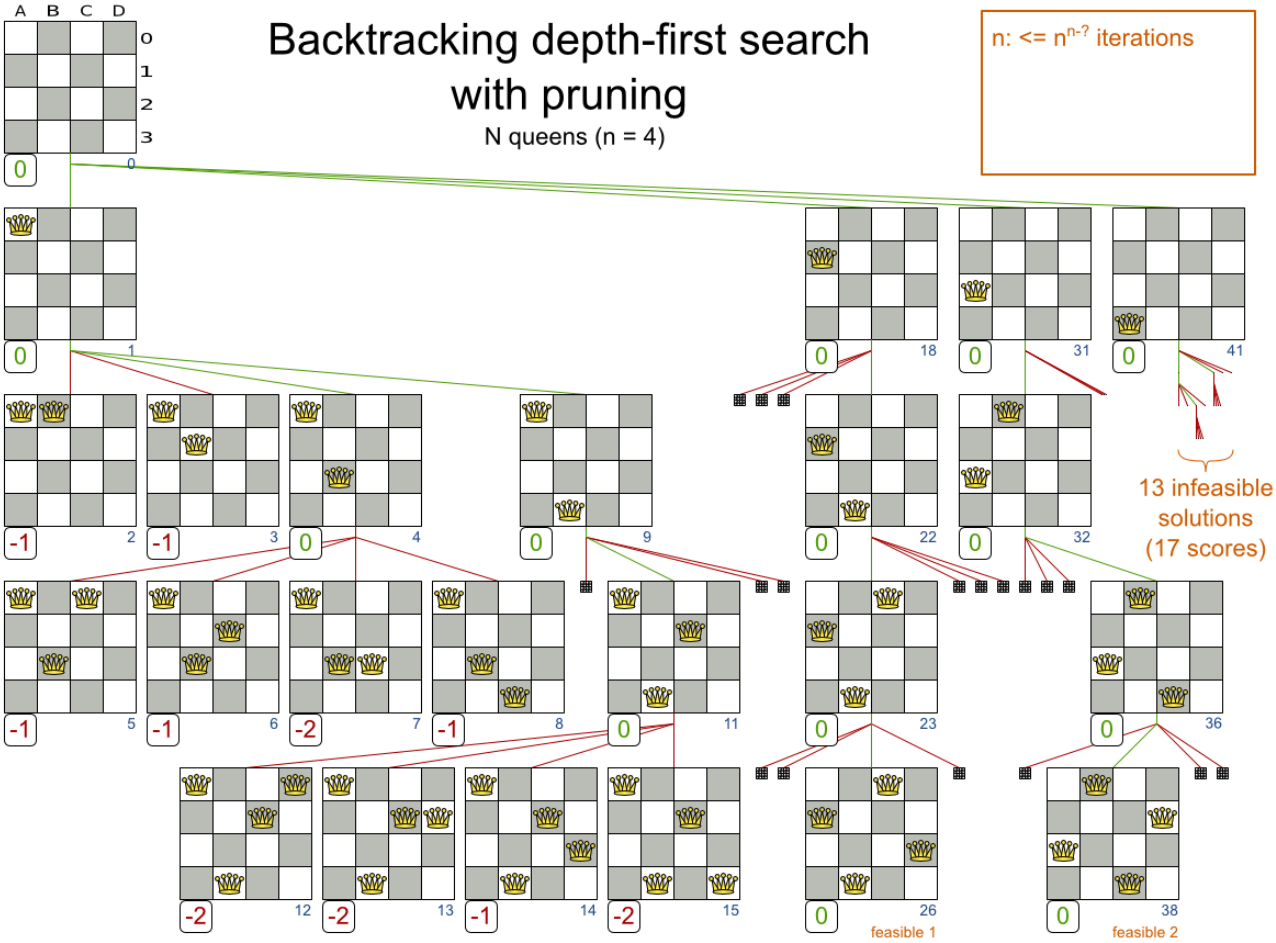
Using the brute force algorithm is easy:

```
<solver>
  ...
  <bruteForce>
  </bruteForce>
</solver>
```

12.3. Depth-first search

12.3.1. Algorithm description

Depth-First Search is an improvement over Brute Force, as it regularly prunes away an entire subset of solutions which cannot have a better solution than the best solution already found at that point. for example: at index 15, it can prune away all unvisited solutions with queen A on row 0, because none will be better than the solution of index 14 with a score of `-1`.



Notice that it (much like Brute Force) creates a search tree that explodes as the problem size increases. **Depth-First Search is mostly unusable for a real-world NP-complete problem due to time limitations.**

Technically, this Backtracking Depth-First Search with pruning is a form of *Branch And Bound*, although Branch and Bound is often more flexible in its pruning.

12.3.2. Configuration

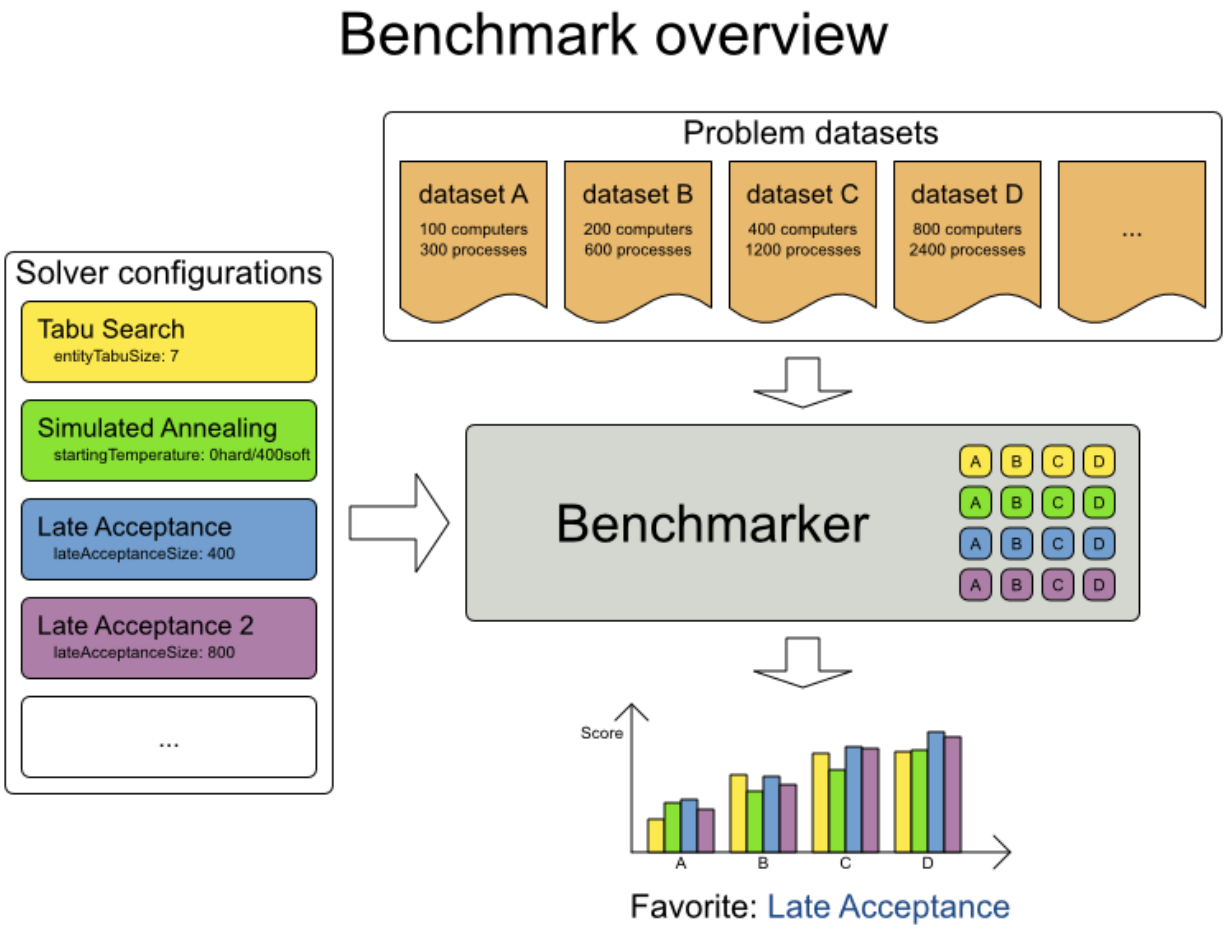
Depth-first search is not yet implemented. Patches welcome.

Chapter 13. Benchmarking and tweaking

13.1. Finding the best `Solver` configuration

OptaPlanner supports several optimization algorithms, but you're probably wondering which is the best one? Although some optimization algorithms generally perform better than others, it really depends on your problem domain. Most solver phases have parameters which can be tweaked. Those parameters can influence the results a lot, even though most solver phases work pretty well out-of-the-box.

Luckily, OptaPlanner includes a benchmarker, which allows you to play out different solver phases with different settings against each other, so you can pick the best configuration for your planning problem.



13.2. Doing a benchmark

13.2.1. Adding a dependency on optaplanner-benchmark

The benchmarker is in a separate artifact called `optaplanner-benchmark`.

If you use Maven, add a dependency in your `pom.xml` file:

```
    <dependency>
      <groupId>org.optaplanner</groupId>
      <artifactId>optaplanner-benchmark</artifactId>
    </dependency>
```

This is similar for Gradle, Ivy and Buildr. The version must be exactly the same as the `optaplanner-core` version used (which is automatically the case if you import `optaplanner-bom`).

If you use ANT, you've probably already copied the required jars from the download zip's `binaries` directory.

13.2.2. Building and running a `PlannerBenchmark`

You can build a `PlannerBenchmark` instance with the `XmlPlannerBenchmarkFactory`. Configure it with a benchmark configuration xml file:

```
        PlannerBenchmarkFactory plannerBenchmarkFactory = new XmlPlannerBenchmarkFactory(
                                    "/org/optaplanner/examples/nqueens/benchmark/
nqueensBenchmarkConfig.xml");
        PlannerBenchmark plannerBenchmark = benchmarkFactory.buildPlannerBenchmark();
        plannerBenchmark.benchmark();
```

A basic benchmark configuration file looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens</benchmarkDirectory>
  <!--<parallelBenchmarkCount>AUTO</parallelBenchmarkCount>-->
  <warmUpSecondsSpend>30</warmUpSecondsSpend>

  <inheritedSolverBenchmark>
    <problemBenchmarks>
       <xstreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</
xstreamAnnotatedClass>
      <inputSolutionFile>data/nqueens/unsolved/32queens.xml</inputSolutionFile>
      <inputSolutionFile>data/nqueens/unsolved/64queens.xml</inputSolutionFile>
```

```
      <problemStatisticType>BEST_SCORE</problemStatisticType>
    </problemBenchmarks>
    <solver>
               <solutionClass>org.optaplanner.examples.nqueens.domain.NQueens</
solutionClass>
           <planningEntityClass>org.optaplanner.examples.nqueens.domain.Queen</
planningEntityClass>
      <scoreDirectorFactory>
        <scoreDefinitionType>SIMPLE</scoreDefinitionType>
                              <scoreDrl>/org/optaplanner/examples/nqueens/solver/
nQueensScoreRules.drl</scoreDrl>
      </scoreDirectorFactory>
      <termination>
        <maximumSecondsSpend>20</maximumSecondsSpend>
      </termination>
      <constructionHeuristic>
                                <constructionHeuristicType>FIRST_FIT_DECREASING</
constructionHeuristicType>
        <forager>
          <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType>
        </forager>
      </constructionHeuristic>
    </solver>
  </inheritedSolverBenchmark>

  <solverBenchmark>
    <name>Entity tabu</name>
    <solver>
      <localSearch>
        <changeMoveSelector>
          <selectionOrder>ORIGINAL</selectionOrder>
        </changeMoveSelector>
        <acceptor>
          <entityTabuSize>5</entityTabuSize>
        </acceptor>
        <forager>
          <pickEarlyType>NEVER</pickEarlyType>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>
  <solverBenchmark>
    <name>Value tabu</name>
    <solver>
      <localSearch>
        <changeMoveSelector>
          <selectionOrder>ORIGINAL</selectionOrder>
        </changeMoveSelector>
        <acceptor>
```

```
          <valueTabuSize>5</valueTabuSize>
        </acceptor>
        <forager>
          <pickEarlyType>NEVER</pickEarlyType>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>
  <solverBenchmark>
    <name>Move tabu</name>
    <solver>
      <localSearch>
        <changeMoveSelector>
          <selectionOrder>ORIGINAL</selectionOrder>
        </changeMoveSelector>
        <acceptor>
          <moveTabuSize>5</moveTabuSize>
        </acceptor>
        <forager>
          <pickEarlyType>NEVER</pickEarlyType>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>
</plannerBenchmark>
```

This `PlannerBenchmark` will try 3 configurations (1 move tabu, 1 entity tabu and 1 value tabu) on 2 data sets (32 and 64 queens), so it will run 6 solvers.

Every `solverBenchmark` element contains a solver configuration (for example with a local search solver phase) and one or more `inputSolutionFile` elements. It will run the solver configuration on each of those unsolved solution files. The element `name` is optional, because it is generated if absent. The inputSolutionFile is read by a *ProblemIO*.

To lower verbosity, the common part of multiple `solverBenchmark` entities can be extracted to the `inheritedSolverBenchmark` element. Yet, every element can still be overwritten per `solverBenchmark` element. Note that inherited solver phases such as `<constructionHeuristic>` or `<localSearch>` are not overwritten but instead are added to the tail of the solver phases list.

You need to specify a `benchmarkDirectory` (relative to the working directory). A benchmark report will be written in that directory.

Note

It's recommended that the `benchmarkDirectory` is a directory ignored for source control and not cleaned by your build system. This way the generated files are not

bloating your source control and they aren't lost when doing a build. Usually that directory is called `local`.

13.2.3. ProblemIO: input and output of Solution files

13.2.3.1. `ProblemIO` interface

The benchmarker needs to be able to read the input files to contain a `Solution` write the best `Solution` of each benchmark to an output file. For that it uses a class that implements the `ProblemIO` interface:

```
public interface ProblemIO {

    String getFileExtension();

    Solution read(File inputSolutionFile);

    void write(Solution solution, File outputSolutionFile);

}
```

Warning

Your input files need to have been written with the same `ProblemIO` class as they are being read by the benchmarker.

13.2.3.2. `XStreamProblemIO`: the default `ProblemIO`

By default, a benchmarker uses a `XStreamProblemIO` instance to read and write solutions.

You need to tell the benchmarker about your `Solution` class which is annotated with XStream annotations:

```
    <problemBenchmarks>
       <xstreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</
xstreamAnnotatedClass>
      <inputSolutionFile>data/nqueens/unsolved/32queens.xml</inputSolutionFile>
      ...
    </problemBenchmarks>
```

Your input files need to have been written with a `XStreamProblemIO` instance, not just any `XStream` instance, because the `XStreamProblemIO` uses a customized `XStream` instance.

Warning

XStream (and XML in general) is a very verbose format. Reading or writing large datasets in this format can cause an `OutOfMemoryError` and performance degradation.

13.2.3.3. Custom `ProblemIO`

Alternatively, you can implement your own `ProblemIO` implementation and configure it with the `problemIOClass` element:

```
    <problemBenchmarks>

mples.machinereassignment.persistence.MachineReassignmentProblemIO</
problemIOClass>
            <inputSolutionFile>data/machinereassignment/import/model_a1_1.txt</
inputSolutionFile>
      ...
    </problemBenchmarks>
```

Warning

A `ProblemIO` implementation must be thread-safe.

13.2.4. Writing the output solution of the benchmark runs

The best solution of each benchmark run can be written to the in the `benchmarkDirectory`. By default, this is disabled, because the files are rarely used and considered bloat. Also, on large datasets, writing the best solution of each single benchmark can take quite some time and memory (causing an `OutOfMemoryError`), especially in a verbose format like XStream.

You can enable to write the output solution in the `benchmarkDirectory` with `writeOutputSolutionEnabled`:

```
    <problemBenchmarks>
      ...
      <writeOutputSolutionEnabled>true</writeOutputSolutionEnabled>
      ...
    </problemBenchmarks>
```

13.2.5. Warming up the HotSpot compiler

Without a warm up, the results of the first (or first few) benchmarks are not reliable, because they will have lost CPU time on HotSpot JIT compilation (and possibly DRL compilation too).

The avoid that distortion, the benchmarker can run some of the benchmarks for a specified amount of time, before running the real benchmarks. Generally, a warm up of 30 seconds suffices:

```
<plannerBenchmark>
  ...
  <warmUpSecondsSpend>30</warmUpSecondsSpend>
  ...
</plannerBenchmark>
```

13.3. Benchmark report

13.3.1. HTML report

After the running a benchmark, a HTML report will be written in the `benchmarkDirectory` with the filename `index.html`. Open it in your browser. It has a nice overview of your benchmark including:

- Summary statistics: graphs and tables
- Problem statistics per `inputSolutionFile`
- Each solver configuration (ranked): easy to copy and paste.
- Benchmark information

The HTML report will use your default locale to format numbers. If you need to share the benchmark report with people from another country, you might want to overwrite the `benchmarkReportLocale`:

```
<plannerBenchmark>
  ...
  <benchmarkReportLocale>en_US</benchmarkReportLocale>
  ...
</plannerBenchmark>
```

13.3.2. Summary statistics

13.3.2.1. Best score summary (graph and table)

Shows the best score per `inputSolutionFile` for each solver configuration.

Useful for visualizing the best solver configuration.

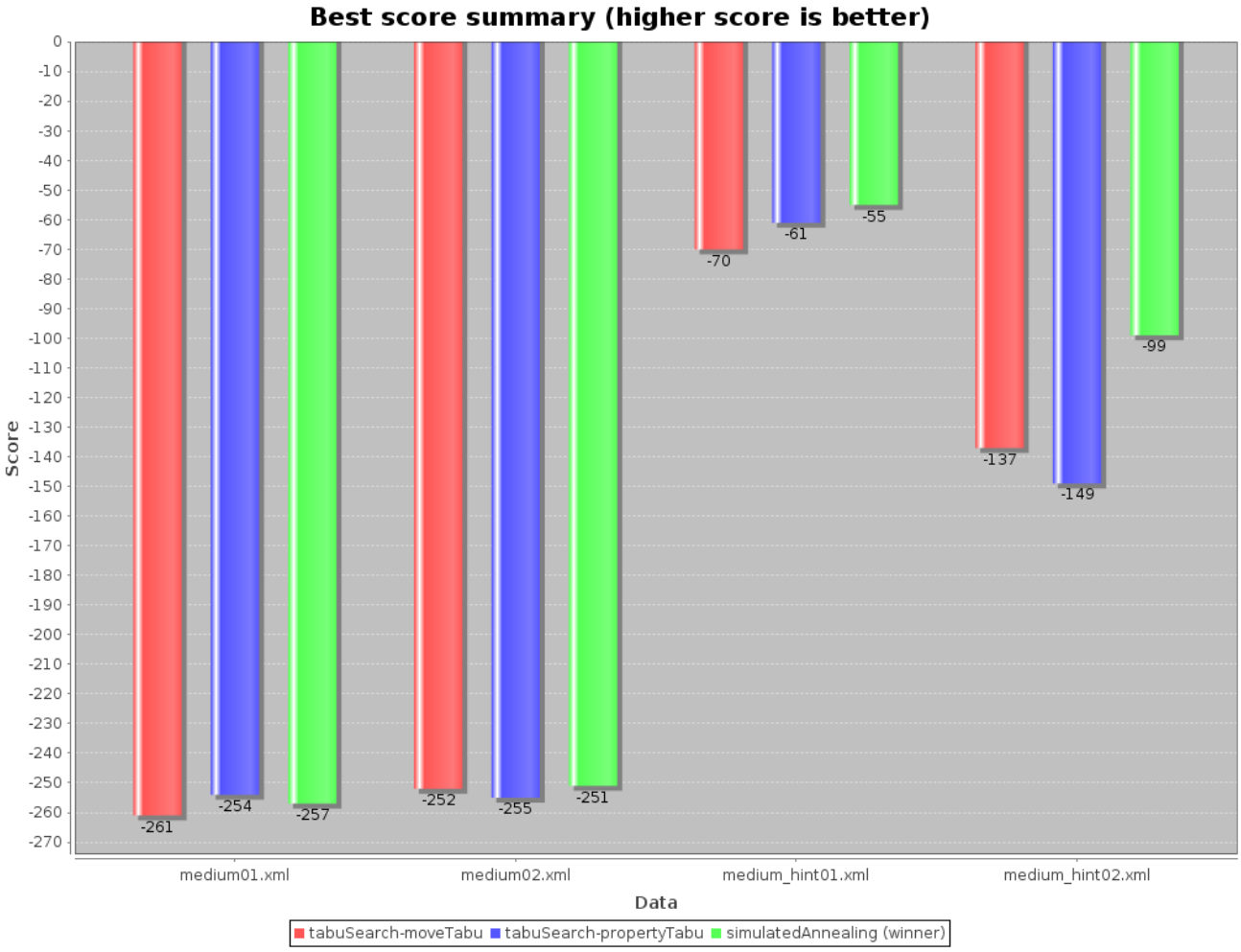


Figure 13.1. Best score summary statistic

13.3.2.2. Best score scalability summary (graph)

Shows the best score per problem scale for each solver configuration.

Useful for visualizing the scalability of each solver configuration.

13.3.2.3. Winning score difference summary (graph and table)

Shows the winning score difference score per `inputSolutionFile` for each solver configuration. The winning score difference is the score difference with the score of the winning solver configuration for that particular `inputSolutionFile`.

Useful for zooming in on the results of the best score summary.

13.3.2.4. Worst score difference percentage (ROI) summary (graph and table)

Shows the return on investment (ROI) per `inputSolutionFile` for each solver configuration if you'd upgrade from the worst solver configuration for that particular `inputSolutionFile`.

Useful for visualizing the return on investment (ROI) to decision makers.

13.3.2.5. Average calculation count summary (graph and table)

Shows the score calculation speed: the average calculation count per second per problem scale for each solver configuration.

Useful for comparing different score calculators and/or score rule implementations (presuming that the solver configurations do not differ otherwise). Also useful to measure the scalability cost of an extra constraint.

13.3.2.6. Time spend summary (graph and table)

Shows the time spend per `inputSolutionFile` for each solver configuration. This is pointless if it's benchmarking against a fixed time limit.

Useful for visualizing the performance of construction heuristics (presuming that no other solver phases are configured).

13.3.2.7. Time spend scalability summary (graph)

Shows the time spend per problem scale for each solver configuration. This is pointless if it's benchmarking against a fixed time limit.

Useful for extrapolating the scalability of construction heuristics (presuming that no other solver phases are configured).

13.3.2.8. Best score per time spend summary (graph)

Shows the best score per time spend for each solver configuration. This is pointless if it's benchmarking against a fixed time limit.

Useful for visualizing trade-off between the best score versus the time spend for construction heuristics (presuming that no other solver phases are configured).

13.3.3. Statistic per data set (graph and CSV)

13.3.3.1. Enabling a problem statistic

The benchmarker supports outputting problem statistics as graphs and CSV (comma separated values) files to the `benchmarkDirectory`.

To configure graph and CSV output of a statistic, just add a `problemStatisticType` line:

```
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens/solved</benchmarkDirectory>
```

```
  <inheritedSolverBenchmark>
    <problemBenchmarks>
      ...
      <problemStatisticType>BEST_SCORE</problemStatisticType>
      <problemStatisticType>CALCULATE_COUNT_PER_SECOND</problemStatisticType>
    </problemBenchmarks>
    ...
  </inheritedSolverBenchmark>
  ...
</plannerBenchmark>
```

Multiple `problemStatisticType` elements are allowed. Some statistic types might influence performance and benchmark results noticeably.

> **Note**
>
> These statistic per data set can slow down the solver noticeably, which can affect the benchmark results. That's why they are optional and not enabled by default.
>
> The non-optional summary statistics cannot slow down the solver noticeably.

The following types are supported:

13.3.3.2. Best score over time statistic (graph and CSV)

To see how the best score evolves over time, add:

```
    <problemBenchmarks>
      ...
      <problemStatisticType>BEST_SCORE</problemStatisticType>
    </problemBenchmarks>
```

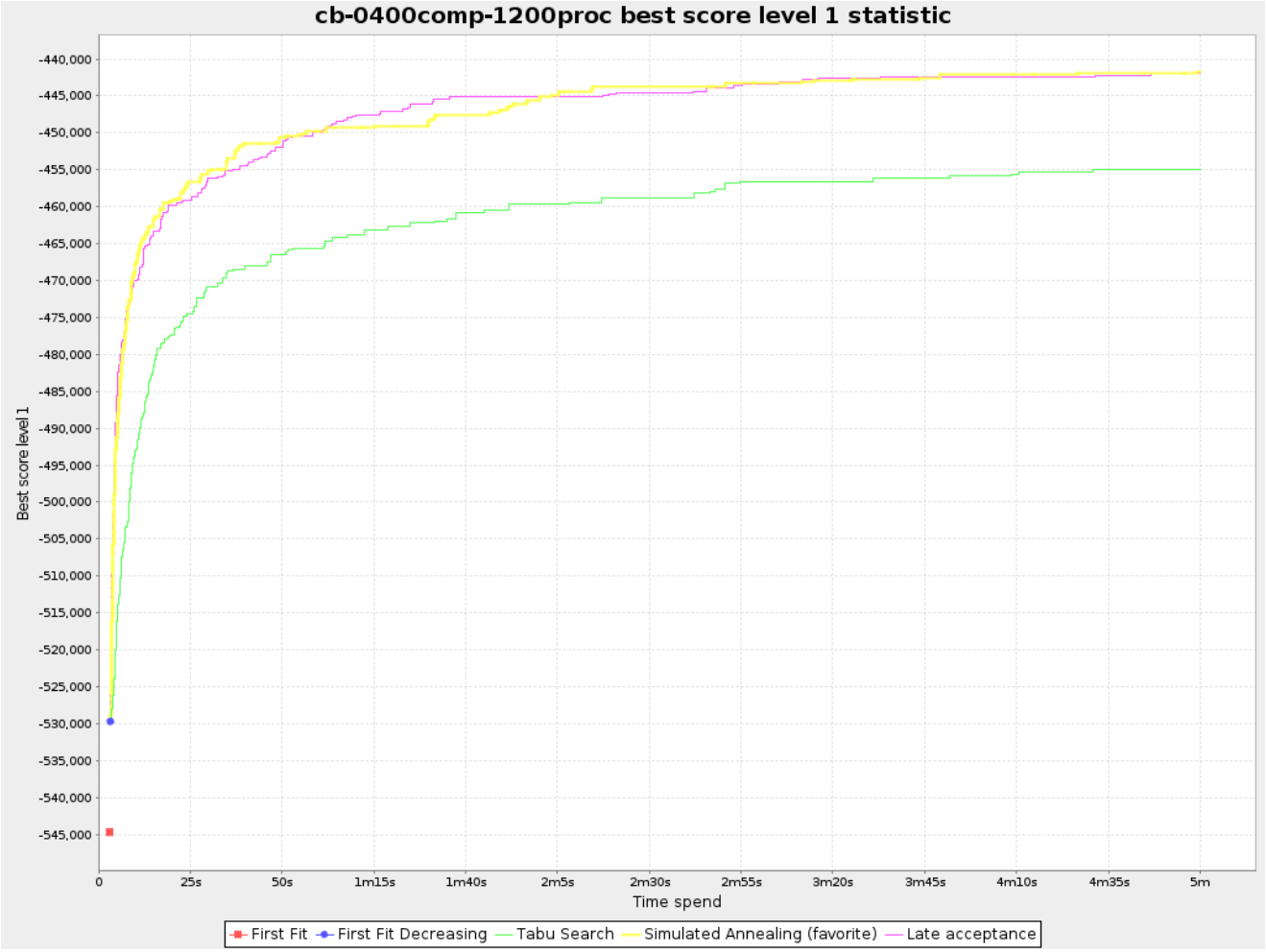


Figure 13.2. Best score over time statistic

The best score over time statistic is very useful to detect abnormalities, such as a potential *score trap*.

> **Note**
>
> A time gradient based algorithm (such as Simulated Annealing) will have a different statistic if it's run with a different time limit configuration. That's because this Simulated Annealing implementation automatically determines its velocity based on the amount of time that can be spend. On the other hand, for the Tabu Search and Late Annealing, what you see is what you'd get.

13.3.3.3. Step score over time statistic (graph and CSV)

To see how the step score evolves over time, add:

```
<problemBenchmarks>
```

```
    ...
    <problemStatisticType>STEP_SCORE</problemStatisticType>
  </problemBenchmarks>
```

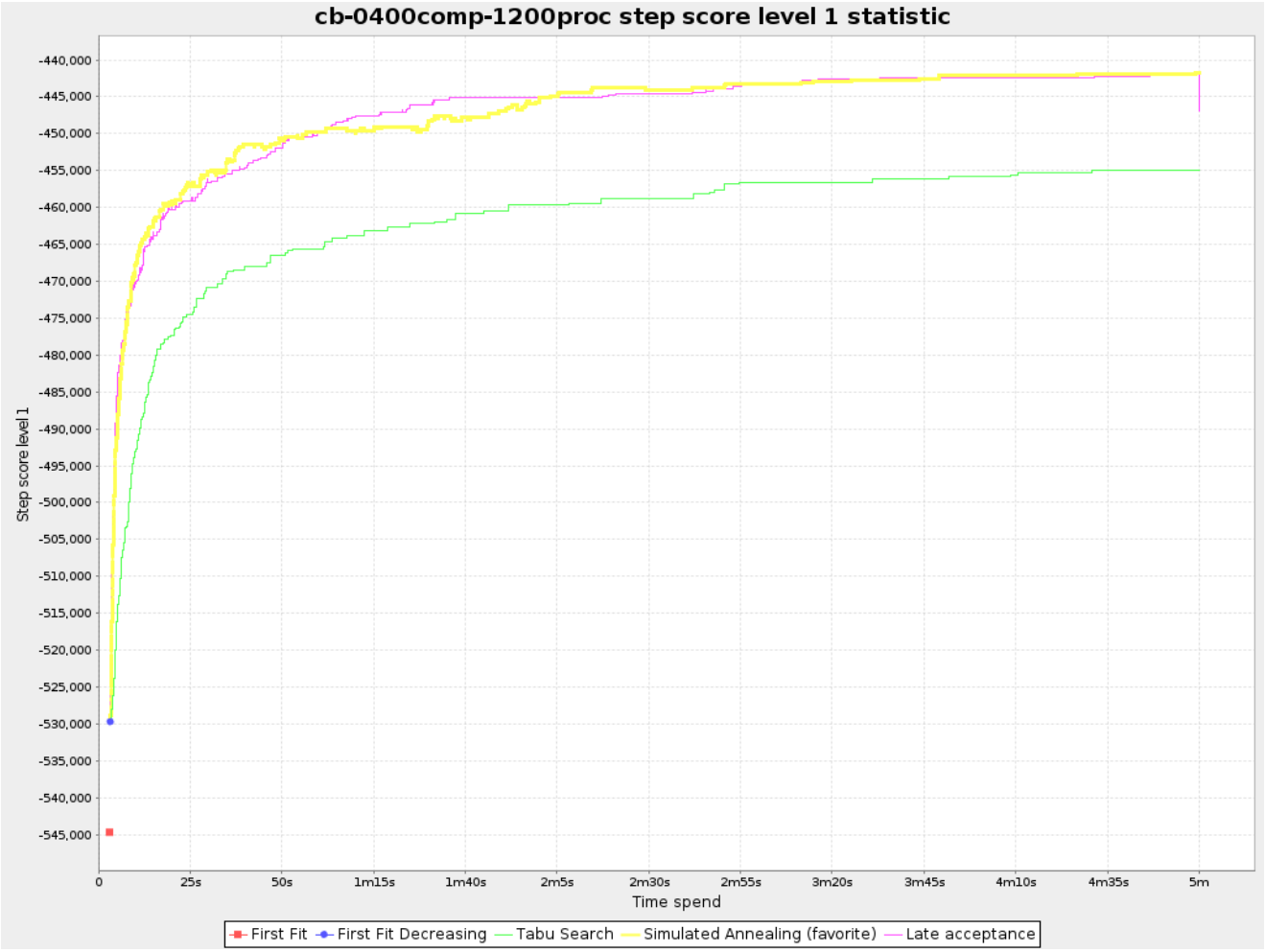


Figure 13.3. Step score over time statistic

Compare the step score statistic with the best score statistic (especially on parts for which the best score flatlines). If it hits a local optima, the solver should take deteriorating steps to escape it. But it shouldn't deteriorate too much either.

Warning

The step score statistic has been seen to slow down the solver noticeably due to GC stress, especially for fast stepping algorithms (such as Simulated Annealing and Late Acceptance).

13.3.3.4. Calculate count per second statistic (graph and CSV)

To see how fast the scores are calculated, add:

```
<problemBenchmarks>
  ...
  <problemStatisticType>CALCULATE_COUNT_PER_SECOND</problemStatisticType>
</problemBenchmarks>
```

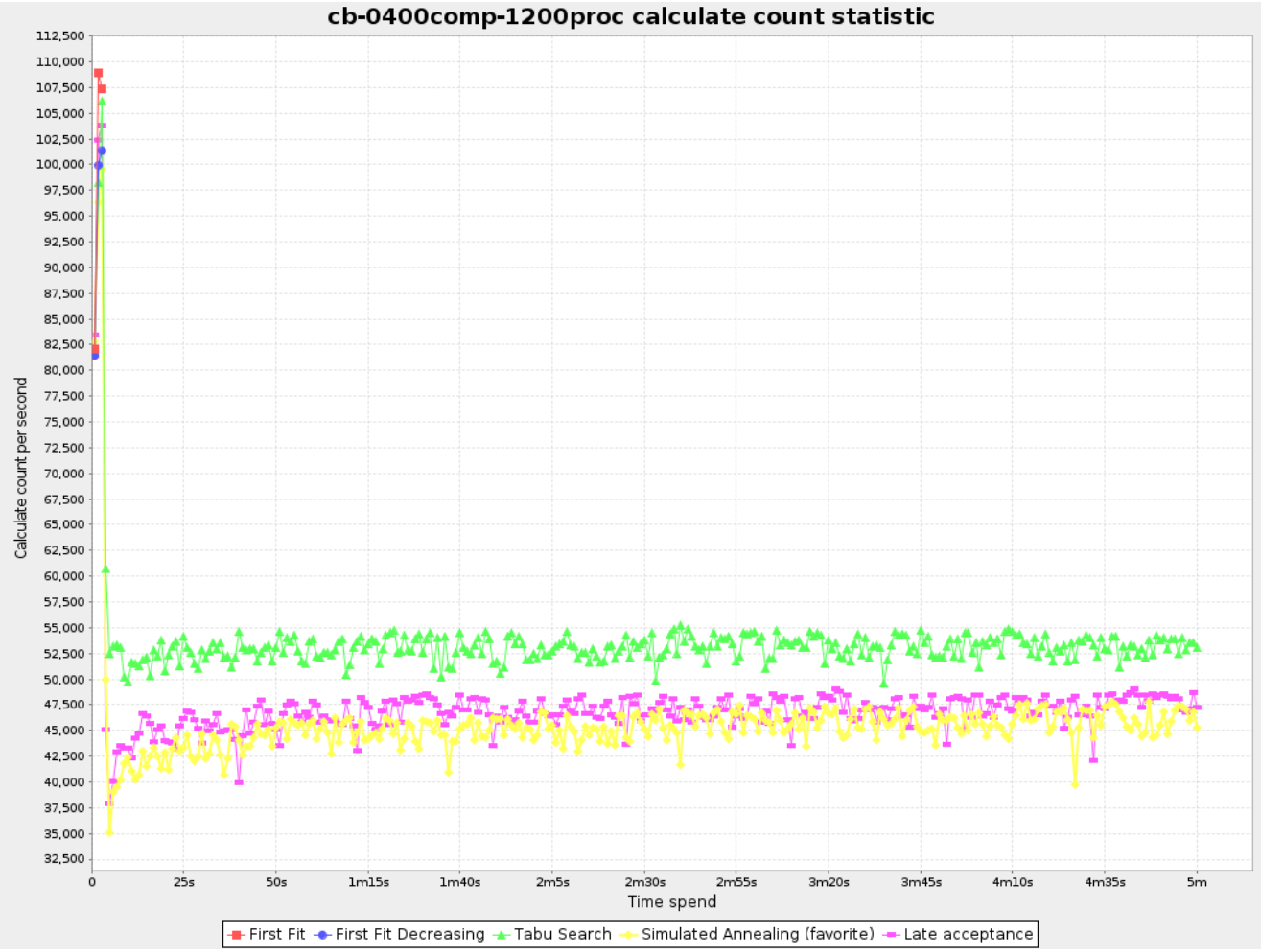


Figure 13.4. Calculate count per second statistic

> **Note**
>
> The initial high calculate count is typical during solution initialization: it's far easier to calculate the score of a solution if only a handful planning entities have been initialized, than when all the planning entities are initialized.

After those few seconds of initialization, the calculate count is relatively stable, apart from an occasional stop-the-world garbage collector disruption.

13.3.3.5. Best solution mutation over time statistic (graph and CSV)

To see how much each new best solution differs from the *previous best solution*, by counting the number of planning variables which have a different value (not including the variables that have changed multiple times but still end up with the same value), add:

```
<problemBenchmarks>
  ...
  <problemStatisticType>BEST_SOLUTION_MUTATION</problemStatisticType>
</problemBenchmarks>
```

Figure 13.5. Best solution mutation over time statistic

Use Tabu Search - an algorithm that behaves like a human - to get an estimation on how difficult it would be for a human to improve the previous best solution to that new best solution.

13.3.3.6. Move count per step statistic (graph and CSV)

To see how the selected and accepted move count per step evolves over time, add:

```
<problemBenchmarks>
  ...
  <problemStatisticType>MOVE_COUNT_PER_STEP</problemStatisticType>
</problemBenchmarks>
```

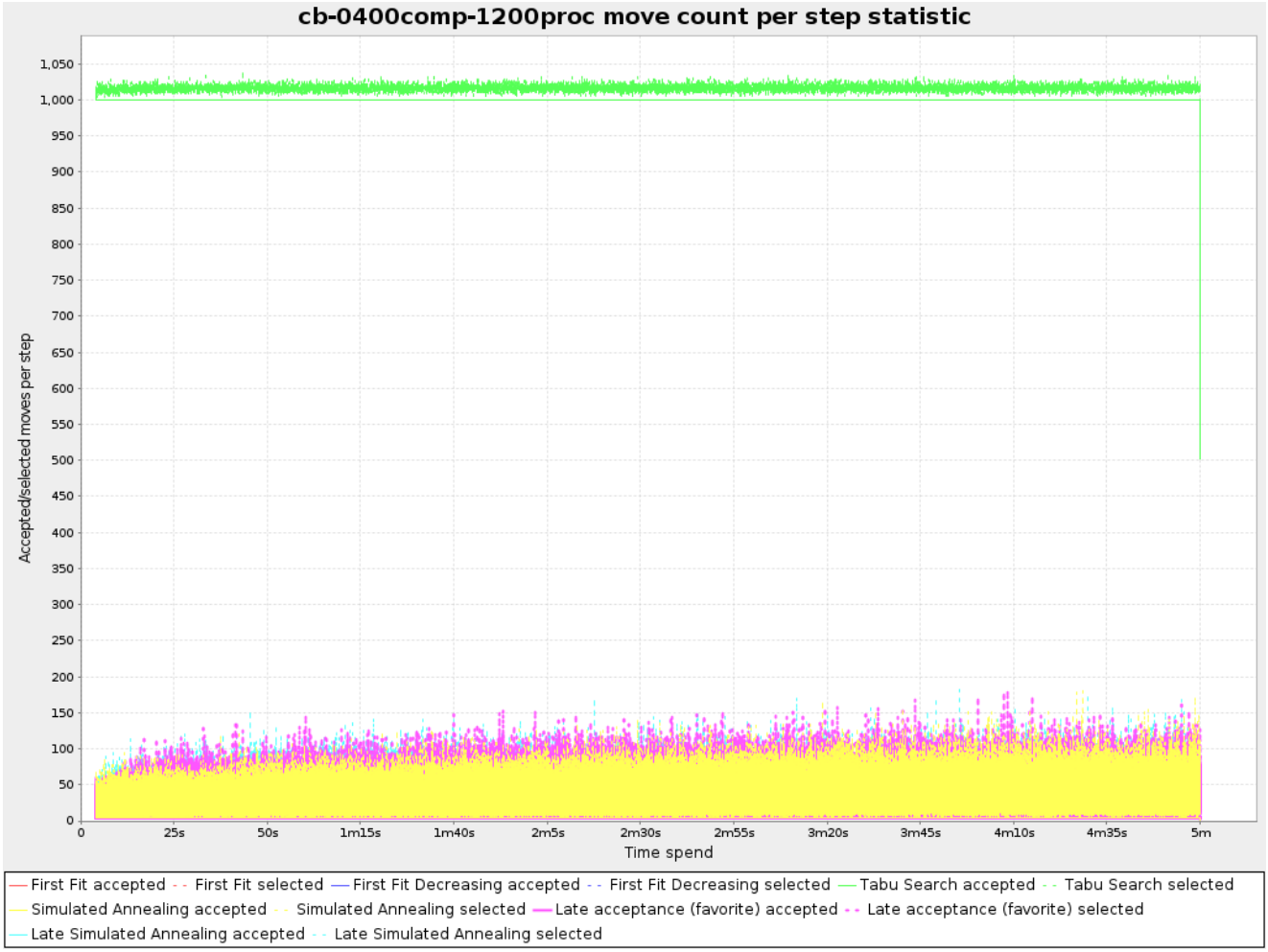


Figure 13.6. Move count per step statistic

Warning

This statistic has been seen to slow down the solver noticeably due to GC stress, especially for fast stepping algorithms (such as Simulated Annealing and Late Acceptance).

13.3.3.7. Memory use statistic (graph and CSV)

To see how much memory is used, add:

```
<problemBenchmarks>
  ...
  <problemStatisticType>MEMORY_USE</problemStatisticType>
</problemBenchmarks>
```

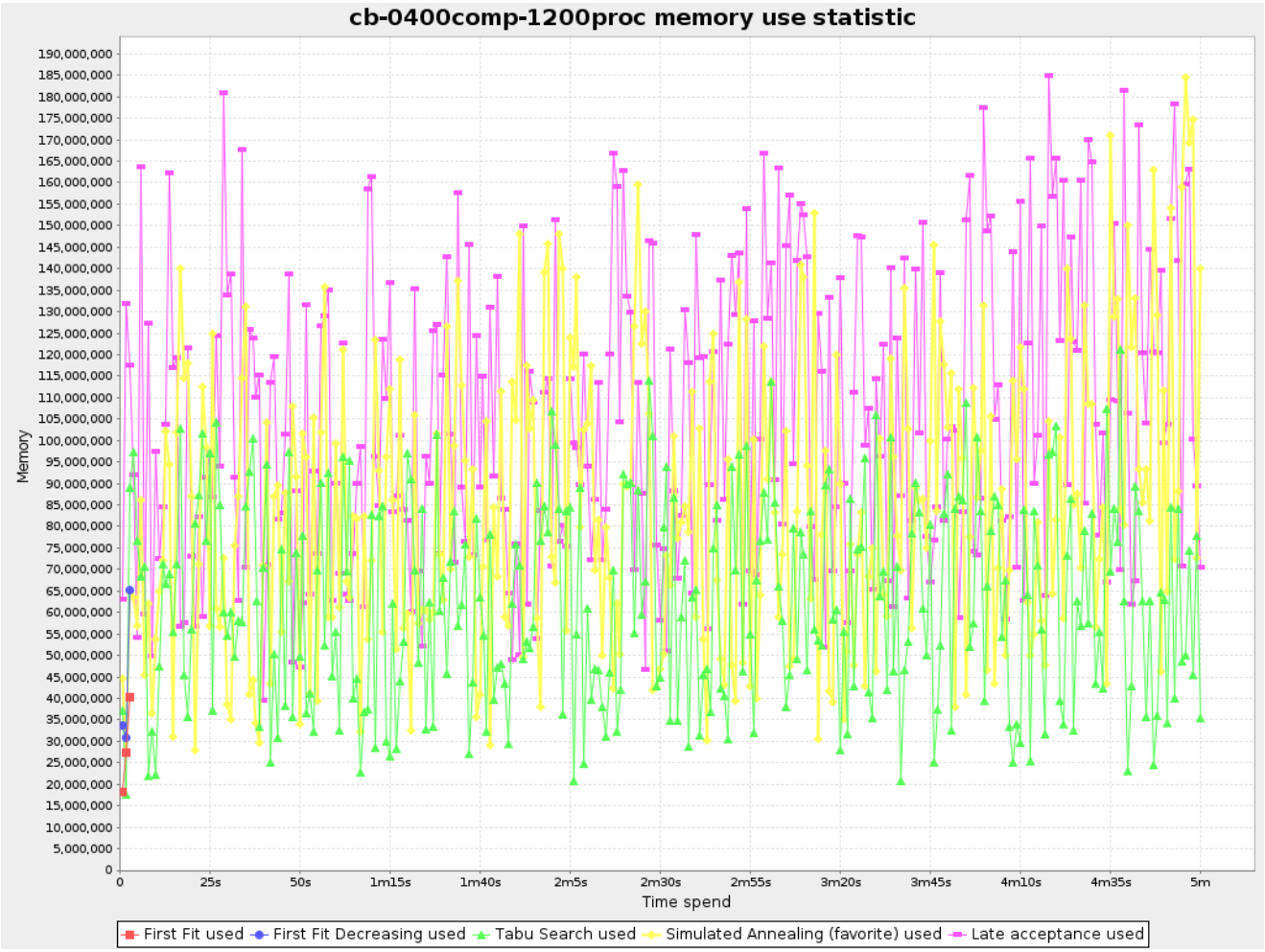


Figure 13.7. Memory use statistic

Warning

The memory use statistic has been seen to affect the solver noticeably.

13.3.4. Ranking the `Solver`s

The benchmark report automatically ranks the solvers. The `Solver` with rank `0` is called the favorite `Solver`: it performs best overall, but it might not be the best on every problem. It's recommended to use that favorite `Solver` in production.

However, there are different ways of ranking the solvers. Configure it like this:

```
<plannerBenchmark>
  ...
  <solverBenchmarkRankingType>TOTAL_SCORE</solverBenchmarkRankingType>
  ...
</plannerBenchmark>
```

The following `solverBenchmarkRankingTypes` are supported:

- `TOTAL_SCORE` (default): Maximize the overall score, so minimize the overall cost if all solutions would be executed.
- `WORST_SCORE`: Minimize the worst case scenario.
- `TOTAL_RANKING`: Maximize the overall ranking. Use this if your datasets differ greatly in size or difficulty, producing a difference in `Score` magnitude.

You can also use a custom ranking, by implementing a `Comparator`:

```
solverBenchmarkRankingComparatorClass>...TotalScoreSolverBenchmarkRankingComparator</
  solverBenchmarkRankingComparatorClass>
```

Or a weight factory:

```
hmarkRankingWeightFactoryClass>...TotalRankSolverBenchmarkRankingWeightFactory</
  solverBenchmarkRankingWeightFactoryClass>
```

13.4. Advanced benchmarking

13.4.1. Benchmarking performance tricks

13.4.1.1. Parallel benchmarking on multiple threads

If you have multiple processors available on your computer, you can run multiple benchmarks in parallel on multiple threads to get your benchmarks results faster:

```
<plannerBenchmark>
  ...
  <parallelBenchmarkCount>AUTO</parallelBenchmarkCount>
  ...
</plannerBenchmark>
```

Warning

Running too many benchmarks in parallel will affect the results of benchmarks negatively. Leave some processors unused for garbage collection and other processes.

We tweak `parallelBenchmarkCount AUTO` to maximize the reliability and efficiency of the benchmark results.

The following `parallelBenchmarkCounts` are supported:

- `1` (default): Run all benchmarks sequentially.
- `AUTO`: Let Planner decide how many benchmarks to run in parallel. This formula is based on experience. It's recommended to prefer this over the other parallel enabling options.
- Static number: The number of benchmarks to run in parallel.

```
<parallelBenchmarkCount>2</parallelBenchmarkCount>
```

- JavaScript formula: Formula for the number of benchmarks to run in parallel. It can use the variable `availableProcessorCount`. For example:

```
<parallelBenchmarkCount>(availableProcessorCount / 2) + 1</
parallelBenchmarkCount>
```

> **Note**
>
> The `parallelBenchmarkCount` is always limited to the number of available processors. If it's higher, it will be automatically decreased.

> **Note**
>
> In the future, we will also support multi-JVM benchmarking. This feature is independent of *multi-threaded solving* [https://issues.jboss.org/browse/PLANNER-76] or multi-JVM solving.

13.4.2. Template based benchmarking and matrix benchmarking

Matrix benchmarking is benchmarking a combination of value sets. For example: benchmark 4 `entityTabuSize` values (`5`, `7`, `11` and `13`) combined with 3 `acceptedCountLimit` values (`500`, `1000` and `2000`), resulting in 12 solver configurations.

To reduce the verbosity of such a benchmark configuration, you can use a *Freemarker* [http://freemarker.sourceforge.net/] template for the benchmark configuration instead:

```
<plannerBenchmark>
  ...

  <inheritedSolverBenchmark>
    ...
  </inheritedSolverBenchmark>

<#list [5, 7, 11, 13] as entityTabuSize>
<#list [500, 1000, 2000] as acceptedCountLimit>
  <solverBenchmark>

<name>entityTabuSize${entityTabuSize}acceptedCountLimit${acceptedCountLimit}</
name>
    <solver>
      <localSearch>
        <unionMoveSelector>
          <changeMoveSelector/>
          <swapMoveSelector/>
        </unionMoveSelector>
        <acceptor>
          <entityTabuSize>${entityTabuSize}</entityTabuSize>
        </acceptor>
        <forager>
```

```
          <acceptedCountLimit>${acceptedCountLimit}</acceptedCountLimit>
        </forager>
      </localSearch>
    </solver>
  </solverBenchmark>
</#list>
</#list>
</plannerBenchmark>
```

And build it with the class `FreemarkerXmlPlannerBenchmarkFactory`:

```
        PlannerBenchmarkFactory plannerBenchmarkFactory = new FreemarkerXmlPlannerBenchmarkFact
                            "/org/optaplanner/examples/cloudbalancing/benchmark/
cloudBalancingBenchmarkConfigTemplate.xml.ftl");
        PlannerBenchmark plannerBenchmark = benchmarkFactory.buildPlannerBenchmark();
```

Chapter 14. Repeated planning

14.1. Introduction to repeated planning

The world constantly changes. The planning facts used to create a solution, might change before or during the execution of that solution. There are 3 types of situations:

- *Unforeseen fact changes*: For example: an employee assigned to a shift calls in sick, an airplane scheduled to take off has a technical delay, one of the machines or vehicles break down, ... Use **backup planning**.
- *Unknown long term future facts*: For example: The hospital admissions for the next 2 weeks are reliable, but those for week 3 and 4 are less reliable and for week 5 and beyond are not worth planning yet. Use **continuous planning**.
- *Constantly changing planning facts*: Use **real-time planning**.

Waiting to start planning - to lower the risk of planning facts changing - usually isn't a good way to deal with that. More CPU time means a better planning solution. An incomplete plan is better than no plan.

Luckily, the optimization algorithms support planning a solution that's already (partially) planned, known as repeated planning.

14.2. Backup planning

Backup planning is the technique of adding extra score constraints to create space in the planning for when things go wrong. That creates a backup plan in the plan. For example: try to assign an employee as the spare employee (1 for every 10 shifts at the same time), keep 1 hospital bed open in each department, ...

Then, when things go wrong (one of the employees calls in sick), change the planning facts on the original solution (delete the sick employee leave his/her shifts unassigned) and just restart the planning, starting from that solution, which has a different score now. The construction heuristics will fill in the newly created gaps (probably with the spare employee) and the metaheuristics will even improve it further.

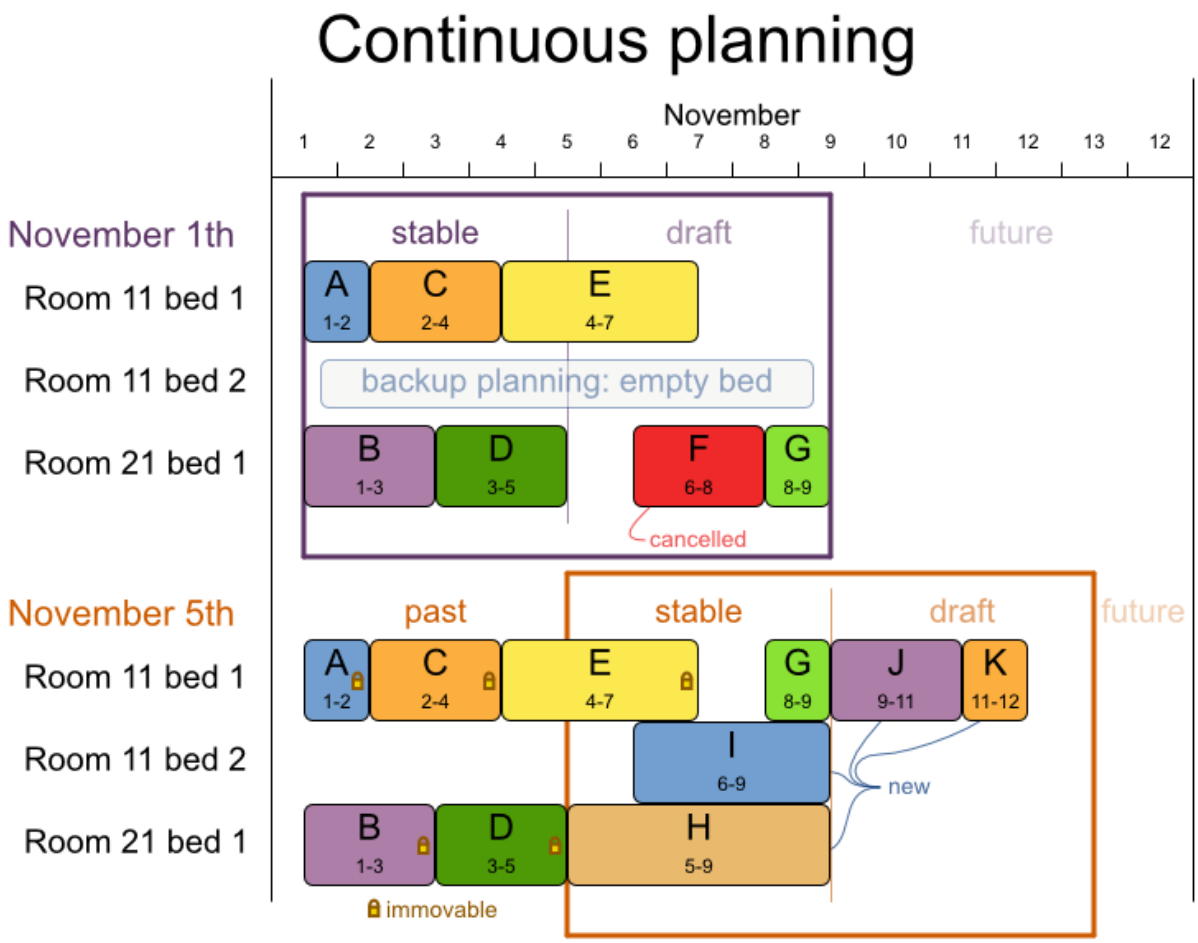
14.3. Continuous planning (windowed planning)

Continuous planning is the technique of planning one or more upcoming planning windows at the same time and repeating that process monthly, weekly, daily or hourly. Because time is infinite, there are infinite future windows, so planning all future windows is impossible. Instead, plan only a fixed number of upcoming planning windows.

Past planning windows are immutable. The first upcoming planning window is considered stable (unlikely to change), while later upcoming planning windows are considered draft (likely to change during the next planning effort). Distant future planning windows are not planned at all.

Past planning windows have only *immovable* planning entities: the planning entities can no longer be changed (they are unable to move), but some of them are still needed in the score calculation, as they might affect some of the score constraints that apply on the upcoming planning entities. For example: when an employee should not work more than 5 days in a row, he shouldn't work today and tomorrow if he worked the past 4 days already.

Sometimes some planning entities are semi-immovable: they can be changed, but occur a certain score penalty if they differ from their original place. For example: avoid rescheduling hospital beds less than 2 days before the patient arrives (unless it's really worth it), avoid changing the airplane gate during the 2 hours before boarding (unless there is no alternative), ...



Notice the difference between the original planning of November 1th and the new planning of November 5th: some planning facts (F, H, I, J, K) changed, which results in unrelated planning entities (G) changing too.

14.3.1. Immovable planning entities

To make some planning entities immovable, simply add an entity `SelectionFilter` that returns `true` if an entity is movable and `false` if it is immovable.

```
public class MovableShiftAssignmentSelectionFilter implements SelectionFilter<ShiftAssignment>

    public boolean accept(ScoreDirector scoreDirector, ShiftAssignment shiftAssignment) {
        ShiftDate shiftDate = shiftAssignment.getShift().getShiftDate();
        NurseRoster nurseRoster = (NurseRoster) scoreDirector.getWorkingSolution();
        return nurseRoster.getNurseRosterInfo().isInPlanningWindow(shiftDate);
    }

}
```

And configure it like this:

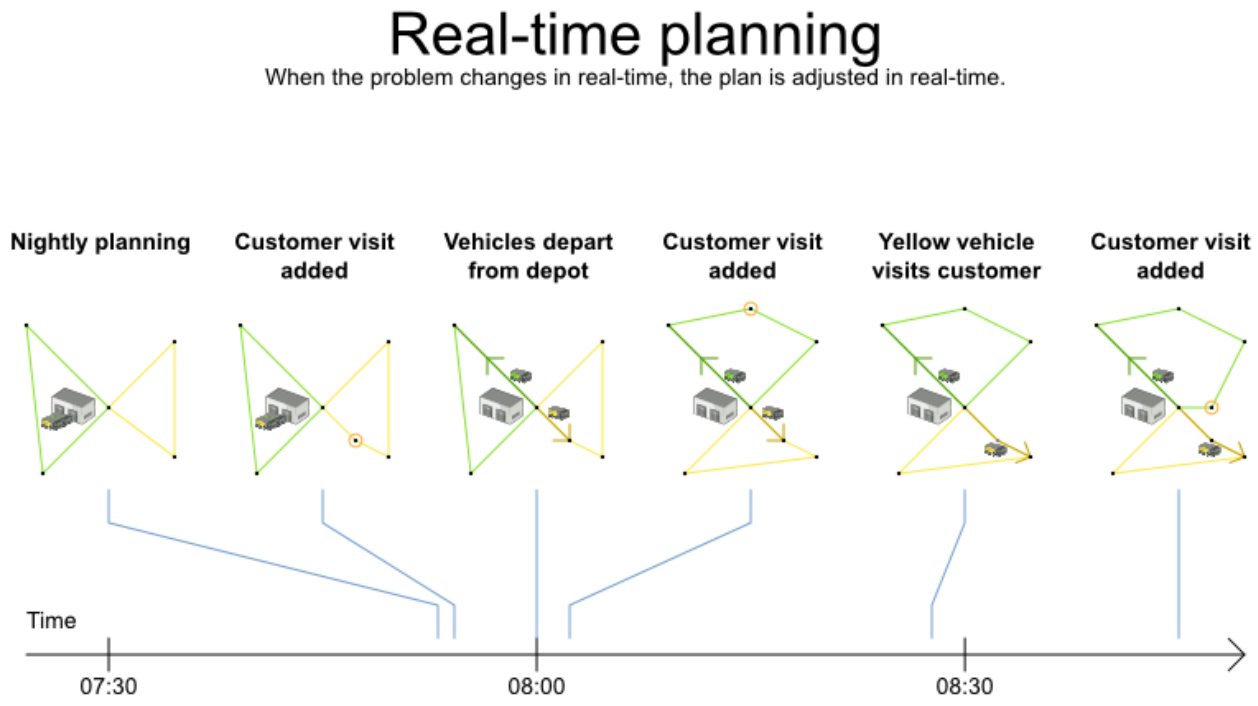
```
@PlanningEntity(movableEntitySelectionFilter = MovableShiftAssignmentSelectionFilter.class)
public class ShiftAssignment {
    ...
}
```

Warning

Custom `MoveListFactory` and `MoveIteratorFactory` implementations must make sure that they don't move immovable entities.

14.4. Real-time planning (event based planning)

To do real-time planning, first combine backup planning and continuous planning with short planning windows to lower the burden of real-time planning.



While the `Solver` is solving, an outside event might want to change one of the problem facts, for example an airplane is delayed and needs the runway at a later time. Do not change the problem fact instances used by the `Solver` while it is solving, as that will corrupt it. Instead, add a `ProblemFactChange` to the `Solver` which it will execute in the solver thread as soon as possible.

```
public interface Solver {

    ...

    boolean addProblemFactChange(ProblemFactChange problemFactChange);

    boolean isEveryProblemFactChangeProcessed();

    ...

}
```

```
public interface ProblemFactChange {
```

```
    void doChange(ScoreDirector scoreDirector);

}
```

Here's an example:

```
    public void deleteComputer(final CloudComputer computer) {
        solver.addProblemFactChange(new ProblemFactChange() {
            public void doChange(ScoreDirector scoreDirector) {
                CloudBalance cloudBalance = (CloudBalance) scoreDirector.getWorkingSolution();
                  // First remove the planning fact from all planning entities
 that use it
                for (CloudProcess process : cloudBalance.getProcessList()) {
                    if (ObjectUtils.equals(process.getComputer(), computer)) {
                      scoreDirector.beforeVariableChanged(process, "computer");
                        process.setComputer(null);
                       scoreDirector.afterVariableChanged(process, "computer");
                    }
                }
                // Next remove it the planning fact itself
                for (Iterator<CloudComputer> it = cloudBalance.getComputerList().iterator(); it
                    CloudComputer workingComputer = it.next();
                    if (ObjectUtils.equals(workingComputer, computer)) {
                       scoreDirector.beforeProblemFactRemoved(workingComputer);
                        it.remove(); // remove from list
                       scoreDirector.beforeProblemFactRemoved(workingComputer);
                        break;
                    }
                }
            }
        });
    }
```

Warning

Any change on the problem facts or planning entities in a `ProblemFactChange` must be done on the instances of the `Solution` of `scoreDirector.getWorkingSolution()`. Note that these are not the same entity instances as in the bestSolution (or therefore your user interface): they are clones.

Warning

Any change on the problem facts or planning entities in a `ProblemFactChange` must be told to the `ScoreDirector`.

Note

Many types of changes can leave a planning entity uninitialized, resulting in a partially initialized solution. That's fine, as long as the first solver phase can handle it. All construction heuristics solver phases can handle that, so it's recommended to configure such a `SolverPhase` as the first phase.

In essence, the `Solver` will stop, run the `ProblemFactChange` and restart. Each `SolverPhase` will run again. Each configured `Termination` (except `terminateEarly`) will reset. This means the construction heuristic will run again, but because little or no planning variables will be uninitialized (unless you have a *nullable planning variable*), this won't take long.

Normally, you won't configure any `Termination`, just call `Solver.terminateEarly()` when the results are needed. Alternatively, you can subscribe to the `BestSolutionChangedEvent`. A `BestSolutionChangedEvent` doesn't guarantee that every `ProblemFactChange` has been processed already, so check `Solver.isEveryProblemFactChangeProcessed()` and ignore any `BestSolutionChangedEvent` fired while that method returns `false`.