Drools Planner User Guide

Chapter 1. Planner introduction

1.1. What is Drools Planner?

Drools Planner **[http://www.jboss.org/drools/drools-planner]** optimizes automated **planning** by combining a search algorithm with the power of the Drools rule engine. Use cases of planning problems include:

- Employee shift rostering, such as nurse rostering

- Vehicle routing, freight routing and people routing, such as *The traveling salesman problem* [http://en.wikipedia.org/wiki/Travelling_salesman_problem] and *The traveling tournament problem* [http://mat.gsia.cmu.edu/TOURN/]

- Educational timetabling, such as lesson scheduling, course schedule, exam scheduling and conference scheduling

- Bin packing, stock sorting and storage organizing

- Machine queue planning

- Miss manners too (although the Drools Planner example solves this differently than the pure Drools Expert example)

A planning problem consists out of a number of constraints. Generally, there are 3 types 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*.

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

These constraints define the *score function* of a planning problem. This is where the drools rule engine comes into play: **adding constraints with score rules is easy and scalable**.

A planning problem has a number of *solutions*. Each solution has a score. There are 3 categories of solutions:

- A *possible solution* is a solution that does or does not break any number of constraints. Planning problems tend to have a incredibly large number of possible solutions. Most 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 remote case that it's not a feasible solution because there are no feasible solutions.

Drools Planner supports several search algorithms to efficiently wade through the incredibly large number of possible solutions. **It makes it easy to switch the search algorithm**, by simply changing the solver configuration.

1.2. Status of Drools Planner

Drools Planner is production ready. The API is almost stable but backward incompatible changes can occur. With the recipe called *UpgradeFromPreviousVersionRecipe.txt* [https://github.com/droolsjbpm/drools-planner/blob/master/drools-planner-distribution/src/main/assembly/filtered-resources/UpgradeFromPreviousVersionRecipe.txt] you can easily upgrade and deal with any backwards incompatible changes between versions. This recipe is included in every release.

Drools Planner, like Drools, is open source software under the Apache Software License.

1.3. Getting Drools Planner and running the examples

1.3.1. Getting the release package and running the examples

You can download a release of Drools Planner from *the drools download site* [http://www.jboss.org/drools/downloads.html]. To run an example, just open the directory `examples` and run the script (`runExamples.sh` on linux or `runExamples.bat` on windows) and pick an example:

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

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

1.3.2. Get it with maven

The Drools Planner jars are available on *the jboss maven repository* [http://repository.jboss.com/maven2/]. If you use maven 2 or 3, just add a dependency to `drools-planner-core` in your project's `pom.xml`:

```
<dependency>
    <groupId>org.drools.planner</groupId>
    <artifactId>drools-planner-core</artifactId>
```

```
        <version>5.x</version>
    </dependency>
```

You might also need to add the jboss repository in your repository manager (nexus, archiva, ...) or your `~/settings.xml`:

```
    <repositories>
        <repository>
            <id>jboss-public-repository-group</id>
            <name>JBoss Public Maven Repository Group</name>
          <url>https://repository.jboss.org/nexus/content/groups/public/</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
        ...
    </repositories>
```

1.3.3. Build it from source

You can also easily build it from source yourself. Clone drools from GitHub and do a maven 2 build:

```
$ git clone git@github.com:droolsjbpm/drools-planner.git drools-planner
...
$ cd drools-planner
$ mvn -DskipTests clean install
...
```

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

```
$ cd drools-planner-examples
$ mvn exec:exec
...
```

1.4. Questions, issues and blogs

Your questions and remarks 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://n3.nabble.com/Drools-User-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 Drools Planner code or for this manual to *the drools issue tracker* [https://jira.jboss.org/jira/browse/JBRULES]. Select the component `drools-planner`.

Pull requests (and patches) are very welcome and get priority treatment! Attach them to a JIRA issue and optionally send a mail to the dev mailing list to get the issue fixed fast. By open sourcing your improvements, you 'll benefit from our peer review, improvements made upon your improvements and maybe even a thank you on our blog.

Check *our blog* [http://blog.athico.com/search/label/planner] and twitter (*Geoffrey De Smet* [http://twitter.com/geoffreydesmet]) for news. If Drools Planner helps you, don't forget to blog or to twitter about it!

Chapter 2. Use cases and examples

2.1. Introduction

Drools Planner has several examples. In this manual we explain Drools Planner mainly using the n queens example. So it's advisable to read at least the section about that example. For advanced users, the following examples are recommended: curriculum course, examination and nurse rostering.

You can find the source code of all these examples in the drools source distribution and also in git under `drools-planner/drools-planner-examples`.

2.2. The n queens example

2.2.1. Screenshot

Here is a screenshot of the example:
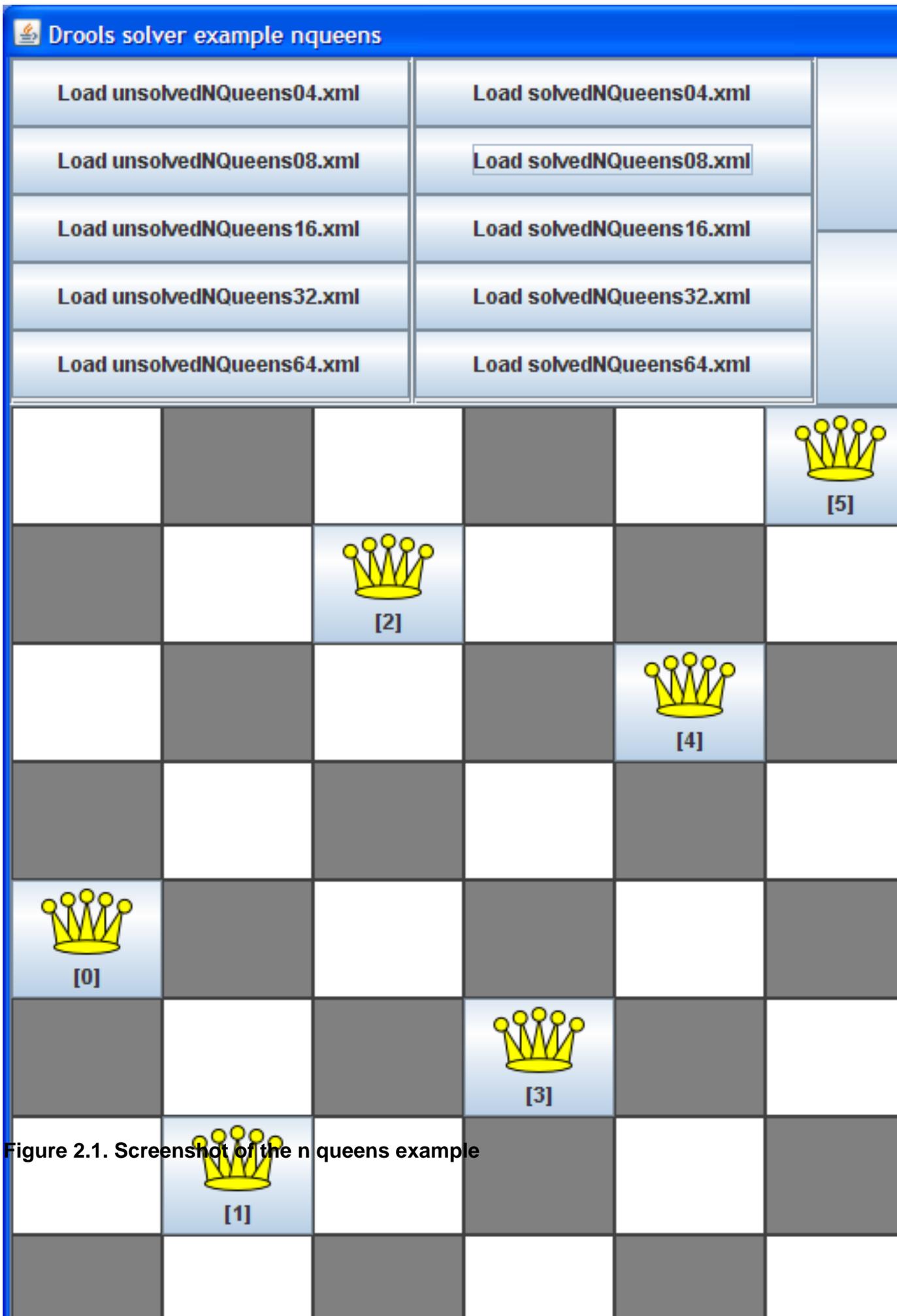
Figure 2.1. Screenshot of the n queens example

2.2.2. Problem statement

The *n queens puzzle* is a puzzle with the follow constraints:

- Use a chessboard of n rows and n columns.

- Place n queens on the chessboard.

- No 2 queens can attack each other. Note that a queen can attack any other queen on the same horizontal, vertical or diagonal line.

The most common n queens puzzle is the 8 queens puzzle, with $n = 8$. We 'll explain Drools Planner using the 4 queens puzzle as the primary example.

A proposed solution could be:
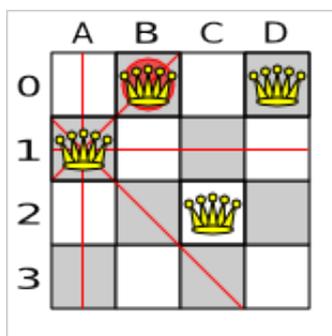
Figure 2.2. A wrong solution for the 4 queens puzzle

The above solution is wrong because queens A1 and B0 can attack each other (as 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.

2.2.3. Solution(s)

Below is a correct solution:
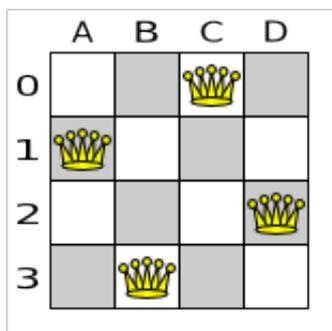
Figure 2.3. A correct solution for the 4 queens puzzle

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.

2.2.4. Problem size

These numbers might give you some insight on the size of this problem.

Table 2.1. NQueens problem size

# queens (n)	# possible solutions (each queen its own column)	# feasible solutions (distinct)	# optimal solutions (distinct)	# possible / # optimal
4	256	2	2	128
8	16777216	64	64	262144
16	18446744073709551616	14772512	14772512	1248720872503
32	1.4615016373309029182036848e+48	?	?	?
64	3.9402006196394479212279040 1e+115	?	?	?
n	n ^ n	?	# feasible solutions	?

The Drools Planner implementation has not been optimized because it functions as a beginner example. Nevertheless, it can easily handle 64 queens.

2.2.5. Domain class diagram

Use a good domain model and it will be easier to understand and solve your problem with Drools Planner. We 'll use this domain model for the n queens example:
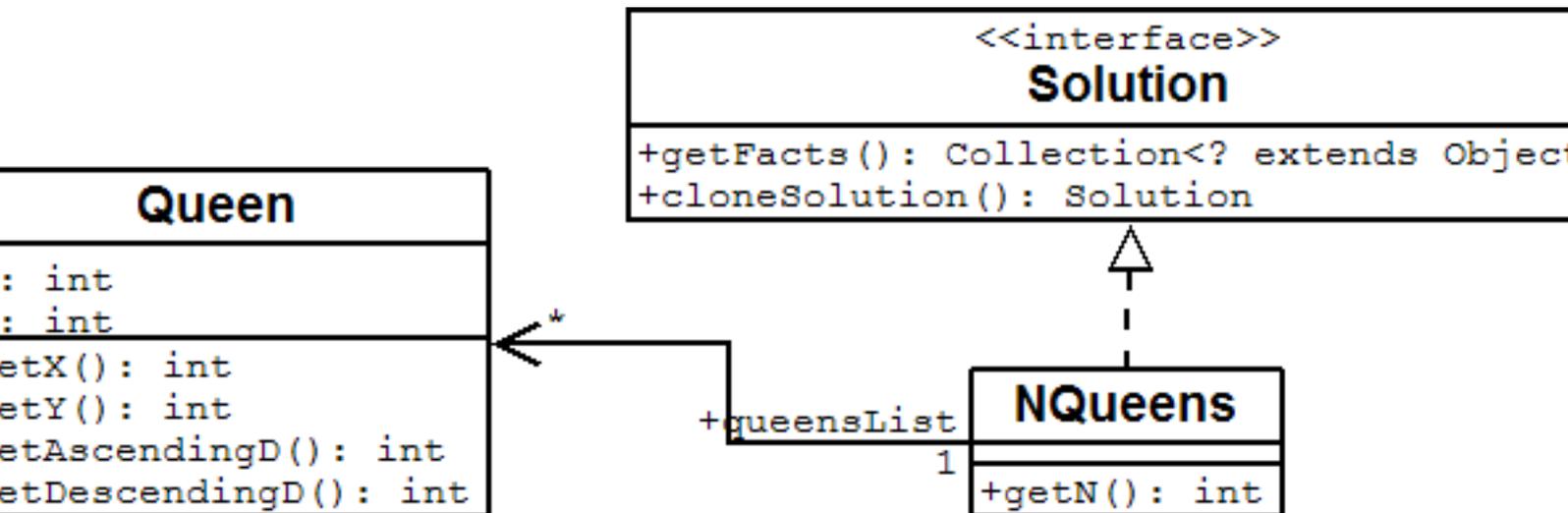
Figure 2.4. NQueens domain class diagram

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

Table 2.2. A solution for the 4 queens puzzle shown in the domain model

A solution	Queen	x	y	ascendingD (x + y)	descendingD (x - y)
		0	1	1 (**)	-1
		1	0 (*)	1 (**)	1
		2	2	4	0
		3	0 (*)	3	3

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

2.3. The Manners 2009 example

2.3.1. 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.

2.4. The Traveling Salesman Problem (TSP) example

2.4.1. Problem statement

Given a list of cities, find the shortest tour for a salesman that visits each city exactly once. *See the wikipedia definition of the traveling Salesman Problem.* [Traveling Salesman Problem]

It is *one of the most intensively studied problems* [one of the most intensively studied problems] in computational mathematics. In the real world, it's often part of a planning problem, along with other constraints such as employee shift time constraints.

2.5. The Traveling Tournament Problem (TTP) example

2.5.1. Screenshot

Here is a screenshot of the example:

Figure 2.5. Screenshot of the traveling tournament example

2.5.2. Problem statement

Schedule matches between N teams with the following 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.

and the following soft constraint:

• Minimize the total distance traveled by all teams.

You can find a detailed description as well as several records of this problem here. [http://mat.gsia.cmu.edu/TOURN/]

2.5.3. Simple and smart implementation

There are 2 implementations (simple and smart) to demonstrate the importance of some performance tips. The `DroolsPlannerExamplesApp` always runs the smart implementation, but with these commands you can compare the 2 implementations yourself:

```
$                    mvn                    exec:exec                    -
Dexec.mainClass="org.drools.planner.examples.travelingtournament.app.simple.SimpleTravelingTou
...
$                    mvn                    exec:exec                    -
Dexec.mainClass="org.drools.planner.examples.travelingtournament.app.smart.SmartTravelingTourna
...
```

The smart implementation performs and scales exponentially better than the simple implementation.

2.5.4. Problem size

These numbers might give you some insight on the size of this problem.

Table 2.3. Traveling tournament problem size

# teams	# days	# matches	# possible solutions (simple)	# possible solutions (smart)	# feasible solutions	# optimal solutions
4	6	12	2176782336	<= 518400	?	1?

# teams	# days	# matches	# possible solutions (simple)	# possible solutions (smart)	# feasible solutions	# optimal solutions
6	10	30	1000000000000000000000000000000	47784725839872000000	0	1?
8	14	56	1.524649437882904656061736043e+64	5.77608277425558771434498864e+43	?	1?
10	18	90	9.43029892325559280477052413e+112	1.07573451027871200629339068e+79	?	1?
12	22	132	1.584141124784195320415735060e+177	2.01650616733413376416949843e+126	?	1?
14	26	182	3.350806356954103223315789511e+257	1.73513467024013808570420241e+186	?	1?
16	30	240	3.229246017998554007515522483e+354	2.45064610271441678267620602e+259	?	1?
n	2 * (n - 1)	n * (n - 1)	(2 * (n - 1)) ^ (n * (n - 1))	<= (((2 * (n - 1))!) ^ (n / 2))	?	1?

2.6. Cloud balancing

2.6.1. Problem statement

There are a number of computers available. Assign a list of processes on those computers.

Hard constraints:

- Every computer should be able to handle the sum of each of the minimal hardware requirements (CPR, RAM, network bandwith) of all its processes.

Soft constraints:

- Each computer that has one or more processes assigned, has a fixed cost. Minimize the total cost.

This is a form of bin packing.

2.7. The ITC 2007 curriculum course example

2.7.1. Problem statement

Schedule lectures into rooms and time periods.

You can find a more detailed description of this problem here. [http://www.cs.qub.ac.uk/itc2007/ curriculmcourse/course_curriculm_index.htm]

2.8. The ITC 2007 examination example

2.8.1. Screenshot

Here is a screenshot of the example:

Period \ Room	0 {260}	1 {100}	2 {129}
15:04:2005 09:30:00 {210}	0 {195, 252}		5
15:04:2005 14:00:00 {210}		1 {135, 85} / 595 {180, 56}	4
18:04:2005 09:30:00 {210}	56 {180, 54}		2 / 5

Figure 2.6. Screenshot of the examination example

Score = -258.000.000

2.8.2. Problem statement

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

There are a number of hard constraints that cannot be broken:

- Exam conflict: 2 exams that share students should not occur in the same period.

- Room capacity: A room's seating capacity should suffice at all times.

- Period duration: A period's duration should suffice for all of its exams.

- Period related hard constraints should be fulfilled:

    - Coincidence: 2 exams should use the same period (but possibly another room).

    - Exclusion: 2 exams should not use the same period.

    - After: 1 exam should occur in a period after another exam's period.

- Room related hard constraints should be fulfilled:

    - Exclusive: 1 exam should not have to share its room with any other exam.

There are also a number of soft constraints that should be minimized (each of which has parameterized penalty's):

- 2 exams in a row.

- 2 exams in a 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: Some periods have a penalty when used.

- Room penalty: Some rooms have a penalty when used.

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

You can find a more detailed description of this problem here. [http://www.cs.qub.ac.uk/itc2007/examtrack/exam_track_index.htm]

2.8.3. Problem size

These numbers might give you some insight on the size of this problem.

Table 2.4. Examination problem size

Set	# students	# exams/ topics	# periods	# rooms	# possible solutions	# feasible solutions	# optimal solutions
exam_comp7_8831		607	54	7	10^1564	?	1?
exam_comp12_484		870	40	49	10^2864	?	1?
exam_comp16_365		934	36	48	10^3023	?	1?
exam_comp4_4214		273	21	1	10^360	?	1?
exam_comp8_7695		1018	42	3	10^2138	?	1?
exam_comp7_9096		242	16	8	10^509	?	1?
exam_comp13_795		1096	80	28	10^3671	?	1?
exam_comp7_5468		598	80	8	10^1678	?	1?
?	s	t	p	r	(p * r) ^ e	?	1?

Geoffrey De Smet (the Drools Planner lead) finished 4th in the International Timetabling Competition 2007's examination track with a very early version of Drools Planner. Many improvements have been made since then.

2.8.4. Domain class diagram

Below you can see the main examination domain classes:

<<interface>>
Solution

<<singleton>>
Examination

1 – examList

An exam changes during solving:
The exam.period and/or exam.
room reference change.
The exam.id and exam.topic
reference do not change.

1..*

Exam

– id : long

– room

1 – topic   0..* – period   0..*

Calculated before solving

1

TopicConflict

– studentSize : int

0..* – leftTopic   0..* – rightTopic

Period

– id : long
– startDateTimeString : String
– periodIndex : int
– dayIndex : int
– duration : int
– penalty : int
– frontLoadLast : boolean

1   1   1

Topic

– id : long
– duration : int
– frontLoadLarge : boolean

1   1

– topic

PeriodHardConstraint

– id : long

– leftTopic

– rightTopic

1..* – studentList

0..*

1

Figure 2.7. Examination domain class diagram

Student

– id : long

Not asserted into the working
memory

<<enum>>
PeriodHardConstraintType

– COINCIDENCE : int
– EXCLUSION : int
– AFTER : int

Notice that we've split up the exam concept into an `Exam` class and a `Topic` class. The `Exam` instances change during solving, when they get another period or room property. The `Topic`, `Period` and `Room` instances never change during solving.

2.9. The patient admission scheduling (PAS) example

In this problem, we have to assign each patient (that will come to the hospital) 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.

There are a couple of hard constraints:

- 2 patients shouldn't 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).

And of course, there are also some 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.

2.10. The INRC 2010 nurse rostering example

2.10.1. Problem statement

Schedule nurses into shifts.

You can find a more detailed description of this problem here. [http://www.kuleuven-kortrijk.be/nrpcompetition]

Chapter 3. Planner configuration

3.1. Types of solvers

Different solvers solve problems in different ways. Each type has advantages and disadvantages. We 'll roughly discuss a few of the solver types here. You can safely skip this section.

3.1.1. Brute force

Brute force creates and evaluates every possible solution, usually by creating a search tree.

Advantages:

- It knows when it has found an optimal solution. If there is more then 1 optimal solution, it finds all optimal solutions.

- It is straightforward and simple to implement.

Disadvantages:

- It performs and scales horribly. Mostly unusable for a real-world problem due to time limitations.

Brute force is currently not implemented in Drools Planner. But we have plans to implement it in the future, as a reference for validating the output of the other solver types.

3.1.2. Branch and bound

Branch and bound is an improvement over brute force, as it prunes away subsets of solutions which cannot have a better solution than the best solution already found at that point.

Advantages:

- It knows when it has found an optimal solution. If there is more then 1 optimal solution, it can find all optimal solutions if needed.

- It can determine the bounds of a problem, which gives an indication of the quality of a solution.

Disadvantages:

- It still scales very badly. Mostly unusable for a real-world problem due to time limitations.

Branch and bound is currently not implemented in Drools Planner. But we have plans to implement it in the future, as a reference for validating the output of the other solver types.

3.1.3. Simplex

Simplex is an algorithm to find the numerical solution of a linear programming problem.

Advantages:

- It knows when it has found an optimal solution.

Disadvantages:

- It's complex and mathematical to implement constraints.

Drools Planner does not implement simplex.

3.1.4. Genetic algorithms

Advantages:

- It's scalable.

- Given a limited time, it can still deliver a pretty decent solution.

Disadvantages:

- It does not know when it has found an optimal solution.

- If the optimal score is unknown (which is usually the case), it must be told when to stop looking (for example based on time spend, user input, ...).

The genetic algorithm is currently not implemented in Drools Planner.

3.1.5. Local search (tabu search, simulated annealing, ...)

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.

Local search works a lot like a human planner: it uses a single search path and moves facts around to find a good feasible solution.

A simple local search can easily get stuck in a local optima, but improvements (such as tabu search and simulated annealing) address this problem.

Advantages:

- It's relatively simple and natural to implement constraints (at least in Drools Planner's implementation).

- It's very scalable, even when adding extra constraints (at least in Drools Planner's implementation).

- Given a limited time, it can still deliver a pretty decent solution.

Disadvantages:

- It does not know when it has found an optimal solution.

- If the optimal score is unknown (which is usually the case), it must be told when to stop looking (for example based on time spend, user input, ...).

Drools Planner implements local search, including tabu search and simulated annealing.

3.2. The size of real world problems

As a planning problem gets bigger, the search space tends to blow up really fast. It's not uncommon to see that it's possible to optimally plan 5 people in less then a second, while planning 6 people optimally would take years. Take a look at the problem size of the examples: many instances have a lot more possible solutions than the minimal number of atoms in the known universe (10^80).

The cold, hard reality is that **for most real-world planning problems we will not find the optimal solution in our lifetimes.** But that's OK, as long as we improve upon the solutions created by human planners (which is easy) or other systems.

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.

3.3. The Solver interface

Solving a planning problem with Drools Planner consists out of 4 steps:

1. Build a solver, for example a tabu search solver for any NQueens puzzle.

2. Set a starting solution on the solver, for example a 4 Queens puzzle instance.

3. Solve it.

4. Get the best solution found by the solver.

Every build-in solver implemented in Drools Planner implements the `Solver` interface:

```java
public interface Solver {

    void setStartingSolution(Solution solution);

    Solution getBestSolution();

    void solve();

    // ...

}
```

There is normally no need to implement the `Solver` interface yourself.

A `Solver` should only be accessed from a single thread, except for the methods that are specifically javadocced as thread-safe.

3.4. Building a Solver

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

```
XmlSolverConfigurer configurer = new XmlSolverConfigurer();
        configurer.configure("/org/drools/planner/examples/nqueens/solver/
nqueensSolverConfig.xml");
    Solver solver = configurer.buildSolver();
```

A basic solver configuration file looks something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<localSearchSolver>
                        <scoreDrl>/org/drools/planner/examples/nqueens/solver/
nQueensScoreRules.drl</scoreDrl>
    <scoreDefinition>
        <scoreDefinitionType>SIMPLE</scoreDefinitionType>
    </scoreDefinition>
    <termination>
        <scoreAttained>0</scoreAttained>
    </termination>
    <selector>

<moveFactoryClass>org.drools.planner.examples.nqueens.solver.NQueensMoveFactory</
    moveFactoryClass>
    </selector>
    <acceptor>
        <completeSolutionTabuSize>1000</completeSolutionTabuSize>
    </acceptor>
    <forager>
        <pickEarlyType>NEVER</pickEarlyType>
    </forager>
</localSearchSolver>
```

This is a tabu search configuration for n queens. We 'll explain the various parts of a configuration later in this manual.

Drools Planner makes it relatively easy to switch a solver type 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.

3.4.1. Environment mode

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.

The environment mode also allows you to detect common bugs in your implementation.

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

```xml
<localSearchSolver>
    <environmentMode>DEBUG</environmentMode>
    ...
</localSearchSolver>
```

There are 3 environment modes:

3.4.1.1. DEBUG

The debug mode is reproducible (see the reproducible mode) and also turns on assertions (such as assert that the undo Move is uncorrupted) to fail-fast on a bug in your Move implementation, your score rule, ...

The debug mode is slow.

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

3.4.1.2. REPRODUCIBLE (default)

The reproducible mode is the default mode because it is recommended during development. In this mode, 2 runs on the same computer will execute the same code in the same order. They will also yield the same result, except if they use a time based termination and they have a sufficiently large difference in allocated CPU time. This allows you to benchmark new optimizations (such as a new move implementation or a different minimalAcceptedSelection setting) fairly.

The reproducible mode is not much slower than the production mode.

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

3.4.1.3. PRODUCTION

The production mode is the fastest and the most robust, but not reproducible. It is recommended for a production environment.

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 caused by another issue (such as a score trap).

3.5. The Solution interface

A Solver can only solve 1 problem instance at a time.

You need to present the problem as a starting `Solution` instance to the solver.

You need to implement the `Solution` interface:

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

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

    Collection<? extends Object> getFacts();

    Solution<S> cloneSolution();

}
```

For example, an `NQueens` instance just holds a list of all its queens:

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

    private List<Queen> queenList;

    // ...

}
```

3.5.1. 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`:

```
    private SimpleScore score;

    public SimpleScore getScore() {
```

```
        return score;
    }

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

Most use cases use a `HardAndSoftScore` instead.

3.5.2. The getFacts method

All objects returned by the `getFacts()` method will be asserted into the drools working memory. Those facts can be used by the score rules. For example, `NQueens` just returns all `Queen` instances.

```
    public Collection<? extends Object> getFacts() {
        return queenList;
    }
```

3.5.3. The cloneSolution method

Most solvers use the `cloneSolution()` method to clone the solution each time they encounter a new best solution. The `NQueens` implementation just clones all `Queen` instances:

```
    public NQueens cloneSolution() {
        NQueens clone = new NQueens();
        List<Queen> clonedQueenList = new ArrayList<Queen>(queenList.size());
        for (Queen queen : queenList) {
            clonedQueenList.add(queen.clone());
        }
        clone.queenList = clonedQueenList;
        clone.score = score;
        return clone;
    }
```

The `cloneSolution()` method should clone no more and no less than the parts of the `Solution` that can change during planning. For example, in the curriculum course schedule example the lectures are cloned, but teachers, courses, timeslots, periods, rooms, ... are not cloned because only a lecture's appointed period or room changes during solving:

```
    /**
     * Clone will only deep copy the {@link #lectureList}.
     */
    public CurriculumCourseSchedule cloneSolution() {
```

```
        CurriculumCourseSchedule clone = new CurriculumCourseSchedule();
        ...
        clone.teacherList = teacherList;
        clone.curriculumList = curriculumList;
        clone.courseList = courseList;
        clone.dayList = dayList;
        clone.timeslotList = timeslotList;
        clone.periodList = periodList;
        clone.roomList = roomList;
      clone.unavailablePeriodConstraintList = unavailablePeriodConstraintList;
       List<Lecture> clonedLectureList = new ArrayList<Lecture>(lectureList.size());
       for (Lecture lecture : lectureList) {
           Lecture clonedLecture = lecture.clone();
           clonedLectureList.add(clonedLecture);
       }
       clone.lectureList = clonedLectureList;
       clone.score = score;
       return clone;
    }
```

3.6. The starting solution

First, you will need to make a starting solution and set that on the solver:

```
solver.setStartingSolution(startingSolution);
```

3.6.1. A simple filler algorithm

For 4 queens we use a simple filler algorithm that creates a starting solution with all queens on a different x and on the same y (with y = 0).
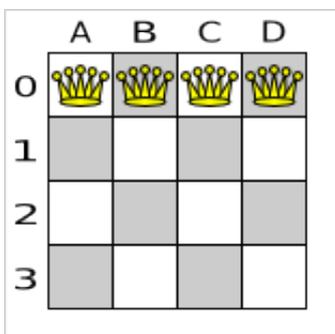

Figure 3.1. Starting solution for the 4 queens puzzle

Here's how we generate it:

```java
private NQueens createNQueens(int n) {
    NQueens nQueens = new NQueens();
    nQueens.setId(0L);
    List<Queen> queenList = new ArrayList<Queen>(n);
    for (int i = 0; i < n; i++) {
        Queen queen = new Queen();
        queen.setId((long) i);
        queen.setX(i); // Different column
        queen.setY(0); // Same row
        queenList.add(queen);
    }
    nQueens.setQueenList(queenList);
    return nQueens;
}
```

The starting solution will probably be far from optimal (or even feasible). Here, it's actually the worst possible solution. However, we 'll let the solver find a much better solution for us anyway.

3.6.2. StartingSolutionInitializer

For large problems, a simple filler algorithm like `createNQueens(int)` doesn't suffice. A (local search) solver starting from a bad starting solution wastes a lot of time to reach a solution which an initializer algorithm can generate in a fraction of that time.

An initializer algorithm usually works something like this:

- It sorts the unplanned elements in a queue according to some general rules, for example by exam student size.

- Next, it plans them in the order they come from the queue. Each element is put the best, still available spot.

- It doesn't change an already planned element. It exits when the queue is empty and all elements are planned.

Such an algorithm is very deterministic: it's really fast, but you can't give it more time to generate an even better solution. In some cases the solution it generates will be feasible, but in most cases it won't. You 'll need a real solver to get to a feasible or more optimal solution. Nevertheless you 'll want to such an initializer to give the real solver a serious head start. You can do this by implementing the `StartingSolutionInitializer` interface:

```java
public interface StartingSolutionInitializer extends SolverAware {

    boolean isSolutionInitialized(Solution solution);

    void initializeSolution(Solution solution);
```

```
}
```

You'll need to set a (uninitialized) solution on the solver. Once the solver starts, it will first call the `StartingSolutionInitializer` to initialize the solution. If the `StartingSolutionInitializer` adds, edits or removes facts it needs to notify the workingMemory about this. It can use score calculation during its initialization process.

Here's an example on how you add the `StartingSolutionInitializer` to the configuration:

```
<localSearchSolver>
    ...
.StartingSolutionInitializer</
    startingSolutionInitializerClass>
    ...
</localSearchSolver>
```

3.7. Solving a problem

Solving a problem is quite easy once you have a solver and the starting solution:

```
solver.setStartingSolution(startingSolution);
solver.solve();
Solution bestSolution = solver.getBestSolution();
```

The `solve()` method will 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 long you allow the solver to work, which solver type you use, ...), that best solution will be a feasible or even an optimal solution.
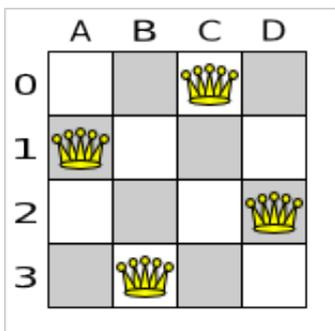

Figure 3.2. Best solution for the 4 queens puzzle (also an optimal solution)

After a problem is solved, you can reuse the same solver instance to solve another problem (of the same problem type).

Chapter 4. Score calculation with a rule engine

4.1. Rule based score calculation

The score calculation (or fitness function) of a planning problem is based on constraints (such as hard constraints, soft constraints, rewards, ...). A rule engine, such as Drools, makes it easy to implement those constraints as *score rules*.

Adding more constraints is easy and scalable (once you understand the DRL syntax). This allows you to add a bunch of soft constraint score rules on top of the hard constraints score rules with little effort and at a reasonable performance cost. For example, for a freight routing problem you could add a soft constraint to avoid the certain flagged highways during rush hour.

4.2. Defining the score rules source

There are 2 ways to define where your score rules live.

4.2.1. A scoreDrl resource on the classpath

This is the simplest 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, for example:

```
                    <scoreDrl>/org/drools/planner/examples/nqueens/solver/
nQueensScoreRules.drl</scoreDrl>
```

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

4.2.2. A RuleBase (possibly defined by Guvnor)

If you prefer to build the `RuleBase` yourself or if you're combining Planner with Guvnor, you can set the `RuleBase` on the `XmlSolverConfigurer` before building the `Solver`:

```
    xmlSolverConfigurer.getConfig().setRuleBase(ruleBase);
```

4.3. Implementing a score rule

The score calculation of a planning problem is based on constraints (such as hard constraints, soft constraints, rewards, ...). A rule engine, such as Drools, makes it easy to implement those constraints as *score rules*.

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

```
rule "multipleQueensHorizontal"
    when
        $q1 : Queen($id : id, $y : y);
        $q2 : Queen(id > $id, y == $y);
    then
                                                insertLogical(new
 UnweightedConstraintOccurrence("multipleQueensHorizontal", $q1, $q2));
end
```

This score rule will fire once for every 2 queens with the same `y`. 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 the starting solution of 4 queens:

Figure 4.1. Starting solution for the 4 queens puzzle

In this starting 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 starting solution will have a score of `-6`. An optimal solution of 4 queens has a score of `0`.

4.4. Delta based score calculation

It's recommended to use Drools in forward-chaining mode (which is the default behaviour), as for score implementations this will create the effect of a *delta based score calculation* instead of a full score calculation on each solution evaluation. For example, if a single queen A moves from y `0` to `3`, it won't bother to recalculate the "multiple queens on the same horizontal line" constraint between 2 queens if neither of those queens is queen A. This is a huge performance gain. **Drools Planner gives you this huge performance gain without forcing you to write a very complicated delta based score calculation algorithm.** Just let the Drools rule engine do the hard work.
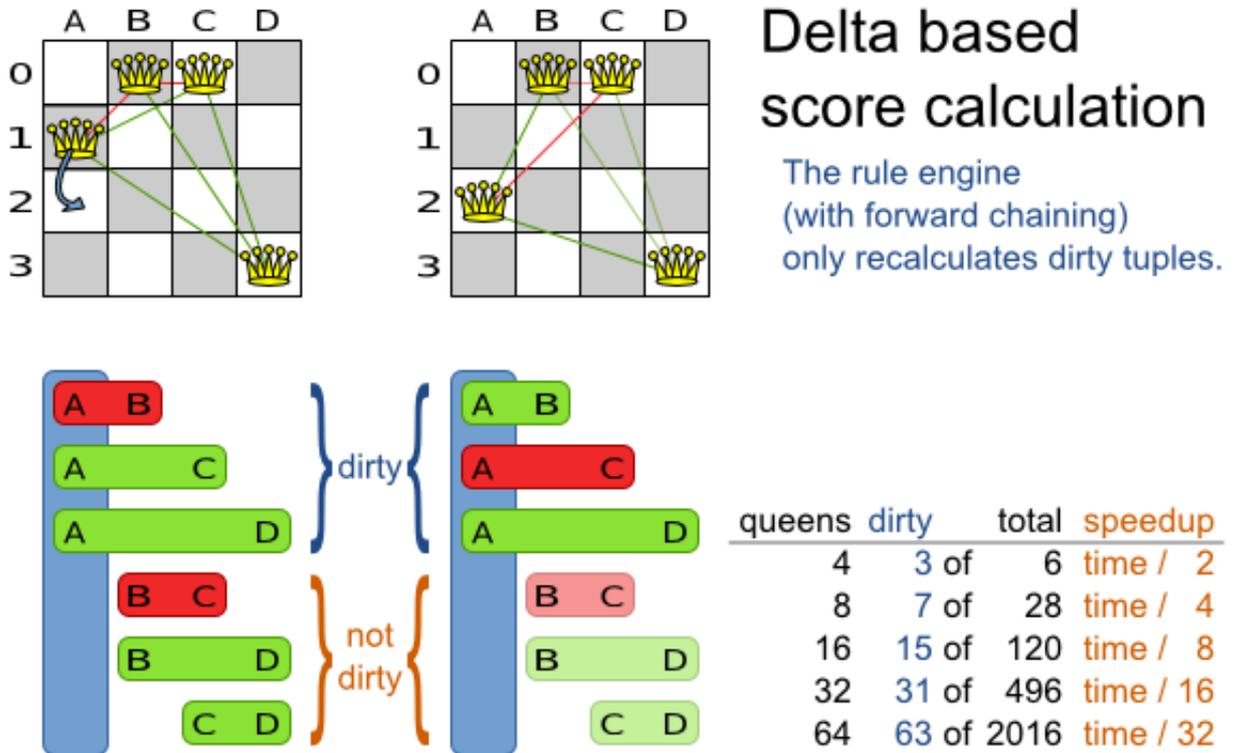
Figure 4.2. Delta based score calculation for the 4 queens puzzle

The speedup due to delta based score calculation is huge, because the speedup is relative to the size of your planning problem (your *n*). By using score rules, you get that speedup without writing any delta code.

4.5. The ScoreDefinition interface

The `ScoreDefinition` interface defines the score representation. The score must a `Score` instance and the instance type (for example `DefaultHardAndSoftScore`) must be stable throughout the solver runtime.

The solver aims to find the solution with the highest score. *The best solution* is the solution with the highest score that it has encountered during its solving.

Most planning problems tend to use negative scores (the amount of negative constraints being broken) with an impossible perfect score of 0. This explains why the score of a solution of 4 queens is the negative of the number of queen couples which can attack each other.

A `ScoreDefinition` instance is configured in the solver configuration:

```
<scoreDefinition>
    <scoreDefinitionType>SIMPLE</scoreDefinitionType>
```

```
    </scoreDefinition>
```

There are a couple of build-in `ScoreDefinition` implementations:

- SIMPLE: The `SimpleScoreDefinition` defines the `Score` as a `SimpleScore` which has a single int value, for example `-123`.

- HARD_AND_SOFT: The `HardAndSoftScoreDefinition` defines the `Score` as a `HardAndSoftScore` which has a hard int value and a soft int value, for example `-123hard/-456soft`.

You can implement your own `ScoreDefinition`, although the build-in score definitions should suffice for most needs.

A `ScoreCalculator` instance is asserted into the working memory as a global called `scoreCalculator`. Your score rules need to (indirectly) update that instance. Usually you 'll make a single rule as an aggregation of the other rules to update the score:

```
global SimpleScoreCalculator scoreCalculator;

rule "multipleQueensHorizontal"
    when
        $q1 : Queen($id : id, $y : y);
        $q2 : Queen(id > $id, y == $y);
    then
                                                      insertLogical(new
 UnweightedConstraintOccurrence("multipleQueensHorizontal", $q1, $q2));
end

// multipleQueensVertical is obsolete because it is always 0

rule "multipleQueensAscendingDiagonal"
    when
        $q1 : Queen($id : id, $ascendingD : ascendingD);
        $q2 : Queen(id > $id, ascendingD == $ascendingD);
    then
                                                      insertLogical(new
 UnweightedConstraintOccurrence("multipleQueensAscendingDiagonal", $q1, $q2));
end

rule "multipleQueensDescendingDiagonal"
    when
        $q1 : Queen($id : id, $descendingD : descendingD);
        $q2 : Queen(id > $id, descendingD == $descendingD);
    then
                                                      insertLogical(new
 UnweightedConstraintOccurrence("multipleQueensDescendingDiagonal", $q1, $q2));
end
```

```
rule "hardConstraintsBroken"
    when
        $occurrenceCount : Number() from accumulate(
            $unweightedConstraintOccurrence : UnweightedConstraintOccurrence(),
            count($unweightedConstraintOccurrence)
        );
    then
        scoreCalculator.setScore(- $occurrenceCount.intValue());
end
```

Optionally, you can also weigh your constraints differently, by multiplying the count of each score rule with its weight. For example in freight routing, you can make 5 broken "avoid crossroads" soft constraints count as much as 1 broken "avoid highways at rush hour" soft constraint. This allows your business analysts to easily tweak the score function as they see fit.

Here's an example of all the NQueens constraints written as a single rule, using multi pattern accumulates and making multipleQueensHorizontal constraint outweigh the other constraints 5 times:

```
// Warning: This currently triggers backwards chaining instead of forward chaining
 and seriously hurts performance and scalability.
rule "constraintsBroken"
    when
        $multipleQueensHorizontal : Long()
        from accumulate(
            $q1 : Queen($id : id, $y : y)
            and Queen(id > $id, y == $y),
          count($q1)
        );
        $multipleQueensAscendingDiagonal : Long()
        from accumulate(
            $q2 : Queen($id : id, $ascendingD : ascendingD)
            and Queen(id > $id, ascendingD == $ascendingD),
          count($q2)
        );
        $multipleQueensDescendingDiagonal : Long()
        from accumulate(
            $q3 : Queen($id : id, $descendingD : descendingD)
            and Queen(id > $id, descendingD == $descendingD),
          count($q3)
        );
    then
            scoreCalculator.setScore(-  (5  *  $multipleQueensHorizontal)  -
 $multipleQueensAscendingDiagonal - $multipleQueensDescendingDiagonal);
end
```

4.6. Tips and tricks

- If you know a certain constraint can never be broken, don't bother writing a score rule for it. For example, the n queens example doesn't have a "multipleQueensVertical" rule because a queen's x never changes and the starting solution puts each queen on a different x. This tends to give a huge performance gain, not just because the score function is faster, but mainly because most solver implementations will spend less time evaluating unfeasible solutions.

- Be watchfull for score traps. A score trap is a state in which several moves need to be done to resolve or lower the weight of a single constraint occurrence. Some examples of score traps:

  - If you need 2 doctors at each table, but you're only moving 1 doctor at a time, then the solver has no insentive 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 your score function.

  - If you only add the table as a cause of the ConstraintOccurrence and forget the jobType (which is doctor or politician), then the solver has no insentive to move a docter to table which is short of a doctor and a politician.

- If you use tabu search, combine it with a `minimalAcceptedSelection` selector. Take some time to tweak the value of `minimalAcceptedSelection`.

- Verify that your score calculation happens in the correct `Number` type. If you're making the sum of integer values, don't let drools use Double's or your performance will hurt. Solver implementations will usually spend most of their execution time running the score function.

- Always remember that premature optimization is the root of all evil. Make sure your design is flexible enough to allow configuration based tweaking.

- Currently, don't allow drools to backward chain instead of forward chain, so avoid query's. It kills scalibilty.

- Currently, don't allow drools to switch to MVEL mode, for performance. You can avoid this by using `eval` in the score rules, for example: `eval(day.getIndex() == $day1.getIndex() + 3)`.

- For optimal performance, use at least java 1.6 and always use server mode (`java -server`). We have seen performance increases of 30% by switching from java 1.5 to 1.6 and 50% by turning on server mode.

- If you're doing performance tests, always remember that the JVM needs to warm up. First load your solver and do a short run, before you start benchmarking it.

In case you haven't figured it out yet: performance (and scalability) is very important for solving planning problems. What good is a real-time freight routing solver that takes a day to find a feasible solution? Even small and innocent looking problems can hide an enormous problem size. For example, they probably still don't know the optimal solution of the traveling tournament problem for as little as 10 traveling teams.

Chapter 5. Local search solver

5.1. Overview

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

- 4 queens has 256 possible solutions (`n ^ n`) and 2 optimal solutions.

- 5 queens has 3125 possible solutions (`n ^ n`) and 1 optimal solution.

- 8 queens has 16777216 possible solutions (`n ^ n`) and 92 optimal solutions.

- Most real-life planning problems have an incredible number of possible solutions and only 1 or a few optimal solutions.

An algorithm that checks every possible solution (even with pruning) can easily run for billions of years on a single real-life planning problem. Most of the time, we are happy with a feasible solution found in a limited amount of time. Local search tends to find a feasible solution relatively fast. Because it acts very much like a human, it is also pretty natural to program.

Local search solves a problem by making a move on the current solution which changes it into a better solution. It does that high number of iterations untill its time runs out and it is satisfied with the solution. It starts with the starting solution.

A local search algorithm and the drools rule engine turn out to be a really nice combination, because:

- A rule engine such as Drools Expert is **great for calculating the score** of a solution of a planning problem. It make it easy to add additional soft or hard constraints such as "a teacher shouldn't teach more then 7 hours a day". However it tends to be too complex to use to actually find new solutions.

- A local search algorithm is **great at finding new improving solutions** for a planning problem, without brute-forcing every possibility. However it needs to know the score of a solution and normally offers no support in calculating that score.

Drools Planner's local search implementation combines both. On top of that, it also offers additional support for benchmarking, etc.

5.2. A move

A move is the change from a solution A to a solution B. For example, below you can see a single move on the starting solution of 4 queens that moves a single queen to another row:
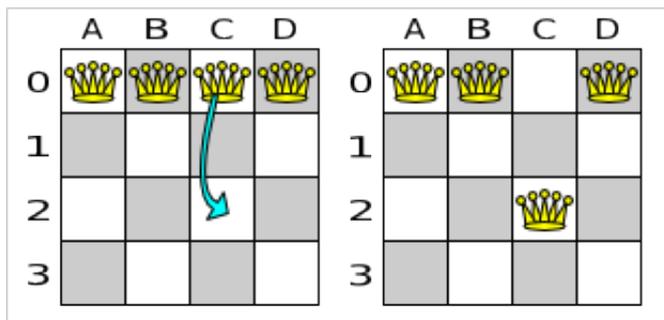
Figure 5.1. A single move (4 queens example)

A move can have a small or large impact. In the above example, the move of queen *C0 to C2* is a small move. Some moves are the same move type. These are some possibilities for move types in n queens:

- Move a single queen to another row. This is a small move. For example, move queen *C0 to C2*.

- Move all queens a number of rows down or up. This a big move.

- Move a single queen to another column. This is a small move. For example, move queen *C2 to A0* (placing it on top of queen A0).

- Add a queen to the board at a certain row and column.

- Remove a queen from the board.

Because we have decided that all queens will be on the board at all times and each queen has an appointed column (for performance reasons), only the first 2 move types are usable in our example. Furthermore, we 're only using the first move type in the example because we think it gives the best performance, but you are welcome to prove us wrong.

Each of your move types will be an implementation of the `Move` interface:

```
public interface Move {

    boolean isMoveDoable(EvaluationHandler evaluationHandler);

    Move createUndoMove(EvaluationHandler evaluationHandler);

    void doMove(EvaluationHandler evaluationHandler);

}
```

Let's take a look at the `Move` implementation for 4 queens which moves a queen to a different row:

```
public class YChangeMove implements Move {
```

```
    private Queen queen;
    private int toY;

    public YChangeMove(Queen queen, int toY) {
        this.queen = queen;
        this.toY = toY;
    }

    // ... see below

}
```

An instance of `YChangeMove` moves a queen from its current y to a different y.

Drools Planner calls the `doMove(WorkingMemory)` method to do a move. The `Move` implementation must notify the working memory of any changes it does on the solution facts:

```
    public void doMove(WorkingMemory workingMemory) {
        FactHandle queenHandle = workingMemory.getFactHandle(queen);
        queen.setY(toY);
        workingMemory.update(queenHandle, queen); // after changes are made
    }
```

You need to call the `workingMemory.update(FactHandle, Object)` method after modifying the fact. Note that you can alter multiple facts in a single move and effectively create a big move (also known as a coarse-grained move).

Drools Planner automatically filters out *non doable moves* by calling the `isDoable(WorkingMemory)` 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.

- 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(WorkingMemory workingMemory) {
        int fromY = queen.getY();
        return fromY != toY;
    }
```

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 can become doable on a later solution.

Each move has an *undo move*: a move (usually of the same type) which does the exact opposite. In the above example the undo move of *C0 to C2* would be the move *C2 to C0*. An undo move can be created from a move, but only before the move has been done on the current solution.

```
public Move createUndoMove(WorkingMemory workingMemory) {
    return new YChangeMove(queen, queen.getY());
}
```

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*.

The local search solver can do and undo a move more than once, even on different (successive) solutions.

A move must implement the `equals()` and `hashcode()` methods. 2 moves which make the same change on a solution, must be equal.

```
public boolean equals(Object o) {
    if (this == o) {
        return true;
    } else if (o instanceof YChangeMove) {
        YChangeMove other = (YChangeMove) o;
        return new EqualsBuilder()
                .append(queen, other.queen)
                .append(toY, other.toY)
                .isEquals();
    } else {
        return false;
    }
}

public int hashCode() {
    return new HashCodeBuilder()
            .append(queen)
            .append(toY)
            .toHashCode();
}
```

In the above example, the `Queen` class uses the default `Object equal()` and `hashcode()` implementations. Notice that it checks if the other move is an instance of the same move type. This 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 Drools Planner's logging more easily:

```java
public String toString() {
    return queen + " => " + toY;
}
```

Now that we can make a single move, let's take a look at generating moves.

5.3. Move generation

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:
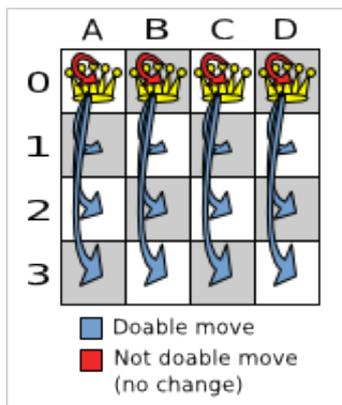

Figure 5.2. Possible moves at step 0 (4 queens example)

As you can see, not all the moves are doable. At the starting solution we have 12 doable moves (`n * (n - 1)`), one of which will be move which changes the starting solution into the next solution. Notice that the number of possible solutions is 256 (`n ^ n`), much more that the amount of doable moves. Don't create a move to every possible solution. Instead use moves which can be sequentially combined to reach every possible solution.

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. Try it yourself: run both the simple

(small moves) and the smart (big moves) version of the traveling tournament example. The smart version evaluates a lot less unfeasible solutions, which enables it to outperform and outscale the simple version.

Move generation currently happens with a `MoveFactory`:

```java
public class NQueensMoveFactory extends CachedMoveListMoveFactory {

    public List<Move> createMoveList(Solution solution) {
        NQueens nQueens = (NQueens) solution;
        List<Move> moveList = new ArrayList<Move>();
        for (Queen queen : nQueens.getQueenList()) {
            for (int n : nQueens.createNList()) {
                moveList.add(new YChangeMove(queen, n));
            }
        }
        return moveList;
    }

}
```

But we might be making move generation part of the DRL's in the future.

5.4. 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:
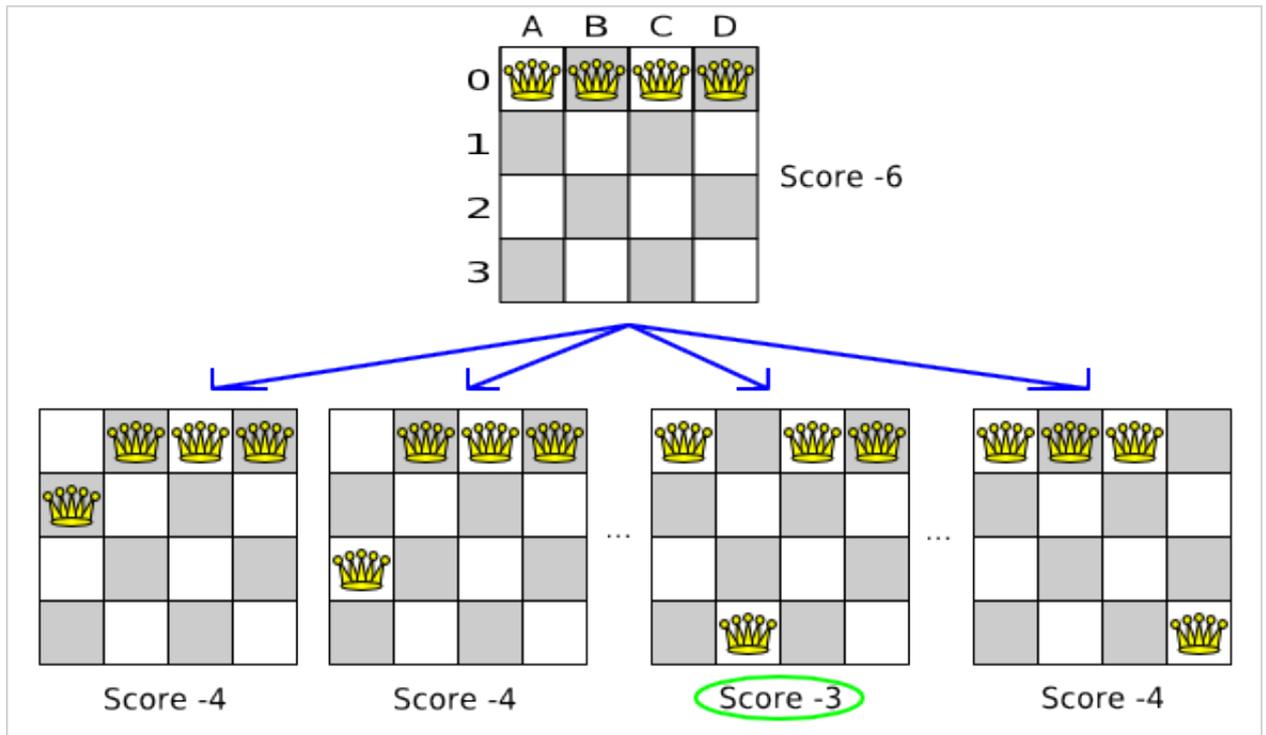
Figure 5.3. Decide the next step at step 0 (4 queens example)

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 5.4. 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
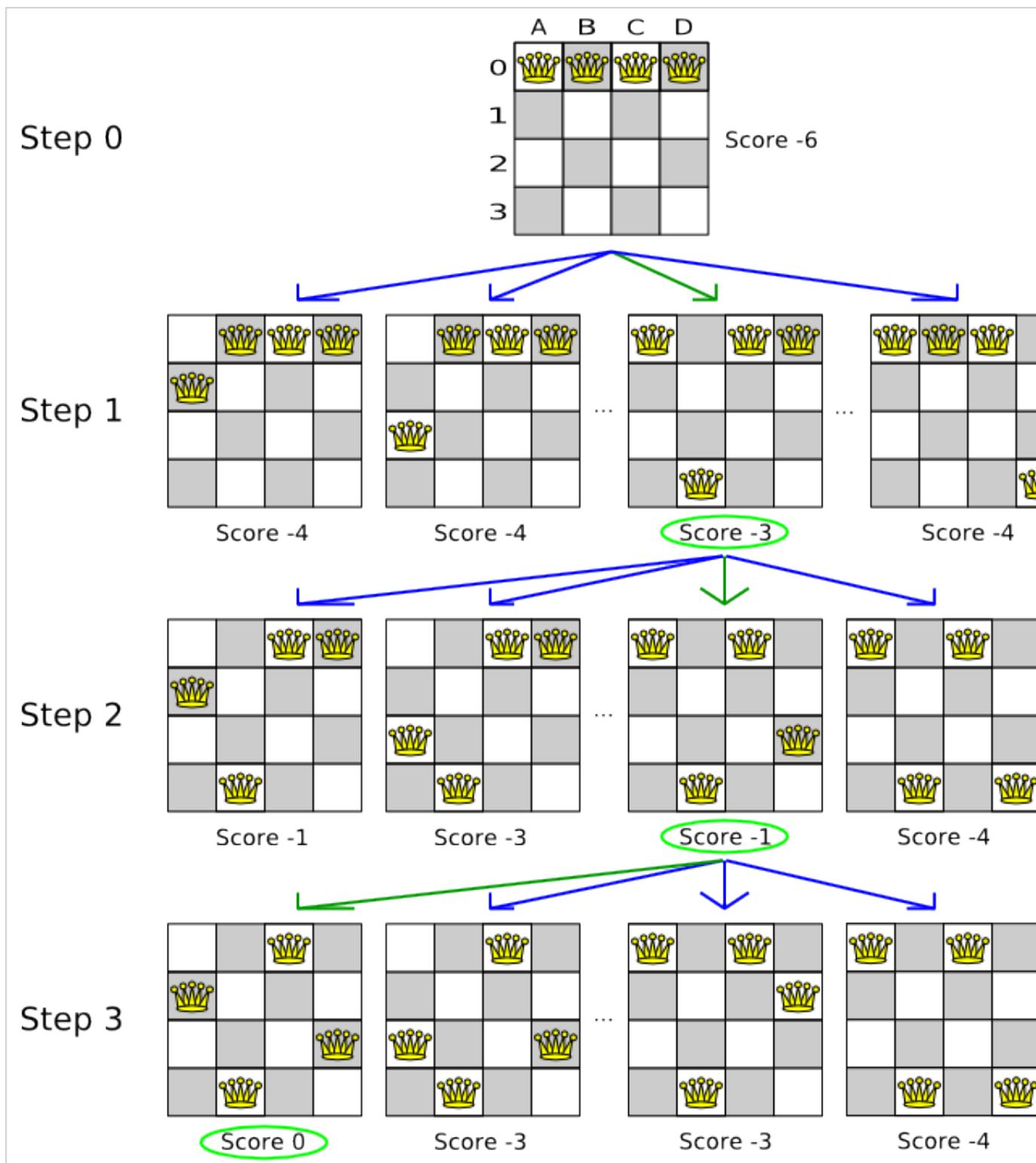
step, it doesn't investigate that solution further. This is one of the reasons why local search is very scalable.

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 INFO logging, this is reflected into the logging:

```
INFO  Solving with random seed (0).
INFO  Starting with time spend (0), score (-6), new best score (-6).
INFO  Step index (0), time spend (4), score (-3), new best score (-3), accepted
 move size (12) for picked step ([Queen-1] 1 @ 0 => 3).
INFO  Step index (1), time spend (7), score (-1), new best score (-1), accepted
 move size (12) for picked step ([Queen-0] 0 @ 0 => 1).
INFO  Step index (2), time spend (10), score (0), new best score (0), accepted
 move size (12) for picked step ([Queen-3] 3 @ 0 => 2).
INFO  Solved at step index (2) with time spend (10) for best score (0) with
 average calculate count per second (7300).
```

Notice that the logging uses the `toString()` method of our `Move` implementation: `[Queen-1] 1 @ 0 => 3`.

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.

5.5. Getting stuck in local optima

A *simple local search* 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 Drools Planner implements better local searches, such as *tabu search* and *simulated annealing* which can avoid these problems. We recommend to never use a simple local search, unless you're absolutely sure there are no local optima in your planning problem.

5.6. Deciding the next step

The local search solver decides the next step with the aid of 3 configurable components:

- A *selector* which selects (or generates) the possible moves of the current solution.

- 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.
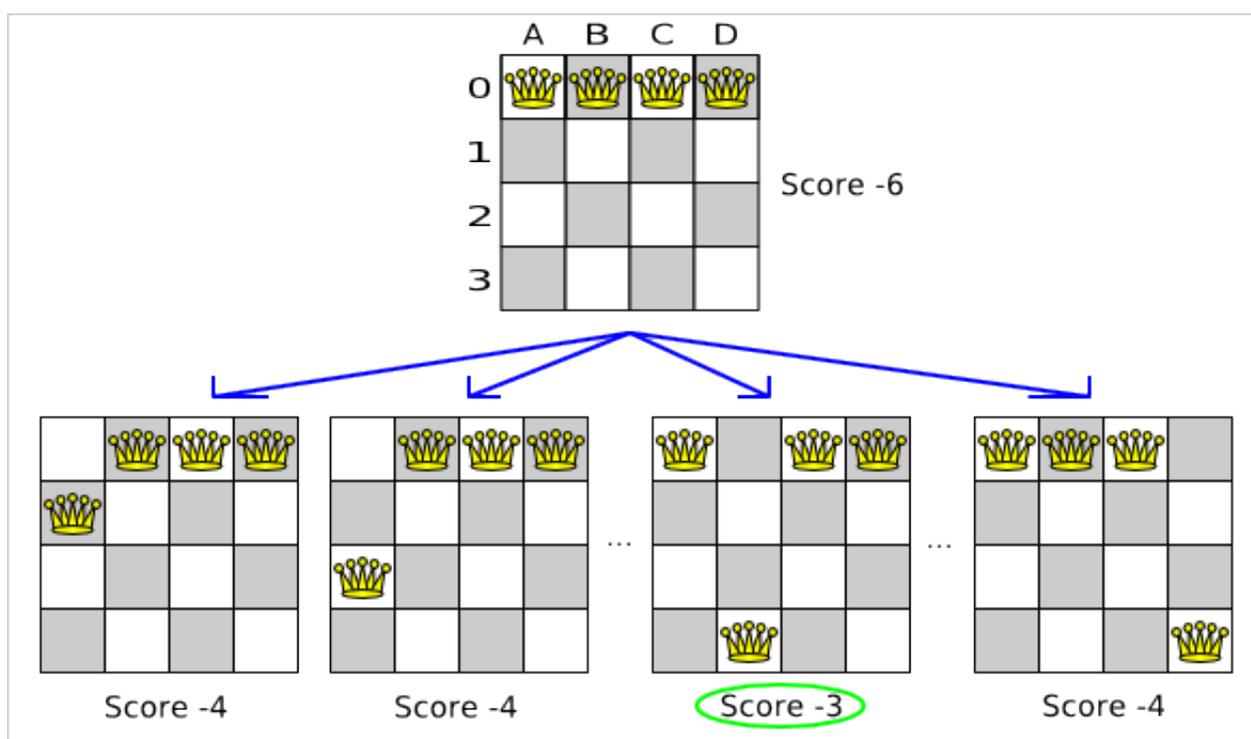
Figure 5.5. Decide the next step at step 0 (4 queens example)

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 DEBUG logging, we can see the decision making in the log:

```
INFO   Solving with random seed (0).
INFO   Starting with time spend (0), score (-6), new best score (-6).
DEBUG      Ignoring not doable move ([Queen-0] 0 @ 0 => 0).
DEBUG      Move score (-4), accept chance (1.0) for move ([Queen-0] 0 @ 0 => 1).
DEBUG      Move score (-4), accept chance (1.0) for move ([Queen-0] 0 @ 0 => 2).
DEBUG      Move score (-4), accept chance (1.0) for move ([Queen-0] 0 @ 0 => 3).
...
DEBUG      Move score (-3), accept chance (1.0) for move ([Queen-1] 1 @ 0 => 3).
```

```
...
DEBUG     Move score (-3), accept chance (1.0) for move ([Queen-2] 2 @ 0 => 3).
...
DEBUG        Move score (-4), accept chance (1.0) for move ([Queen-3] 3 @ 0 => 3).
INFO  Step index (0), time spend (6), score (-3), new best score (-3), accepted
 move size (12) for picked step ([Queen-1] 1 @ 0 => 3).
...
```

5.6.1. Selector

A selector is currently based on a `MoveFactory`.

```
   <selector>

<moveFactoryClass>org.drools.planner.examples.nqueens.solver.NQueensMoveFactory</
      moveFactoryClass>
      </selector>
```

You're not obligated to generate the same set of moves at each step. It's generally a good idea to use several selectors, mixing fine grained moves and course grained moves:

```
   <selector>
     <selector>
.nurserostering.solver.move.factory.EmployeeChangeMoveFactory</
      moveFactoryClass>
        </selector>
        <selector>
rserostering.solver.move.factory.AssignmentSwitchMoveFactory</
      moveFactoryClass>
        </selector>
        <selector>
ove.factory.AssignmentPillarPartSwitchMoveFactory</
      moveFactoryClass>
        </selector>
     </selector>
```

5.6.2. Acceptor

An acceptor is used (together with a forager) to active tabu search, simulated annealing, great deluge, ... For each move it generates an accept chance. If a move is rejected it is given an accept chance of `0.0`.

You can implement your own `Acceptor`, although the build-in acceptors should suffice for most needs. You can also combine multiple acceptors.

5.6.2.1. 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. Drools Planner 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. We recommend this type of tabu because it tends to give the best results and requires little or no tweaking.

```
<acceptor>
    <completeSolutionTabuSize>1000</completeSolutionTabuSize>
</acceptor>
```

- *Move tabu* makes recent steps tabu. It does not accept a move equal to one of those steps.

```
<acceptor>
    <completeMoveTabuSize>7</completeMoveTabuSize>
</acceptor>
```

- *Undo move tabu* makes the undo move of recent steps tabu.

```
<acceptor>
    <completeUndoMoveTabuSize>7</completeUndoMoveTabuSize>
</acceptor>
```

- *Property tabu* makes a property of recent steps tabu. For example, it can make the queen tabu, so that a recently moved queen can't be moved.

```
<acceptor>
    <completePropertyTabuSize>5</completePropertyTabuSize>
</acceptor>
```

To use property tabu, your moves must implement the `TabuPropertyEnabled` interface, for example:

```
public class YChangeMove implements Move, TabuPropertyEnabled {

    private Queen queen;
```

```
    private int toY;

    // ...

    public List<? extends Object> getTabuPropertyList() {
        return Collections.singletonList(queen);
    }

}
```

You can also make multiple properties tabu (with OR or AND semantics):

```
    public List<? extends Object> getTabuPropertyList() {
        // tabu with other moves that contain the same leftExam OR the
    same rightExam
        return Arrays.asList(leftExam, rightExam);
    }
```

```
    public List<? extends Object> getTabuPropertyList() {
        // tabu with other moves that contain the same exam AND the same
    toPeriod (but not necessary the same toRoom)
        return Collections.singletonList(Arrays.asList(exam, toPeriod));
    }
```

You can even combine tabu types:

```
    <acceptor>
        <completeSolutionTabuSize>1000</completeSolutionTabuSize>
        <completeMoveTabuSize>7</completeMoveTabuSize>
    </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. Experiments teach us that it is generally best to use a prime number for the move tabu, undo move tabu or property tabu size.

A tabu search acceptor should be combined with a high or no subset selection.

5.6.2.2. 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 unimproving 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 simple local search, 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>
        <minimalAcceptedSelection>4</minimalAcceptedSelection>
    </forager>
```

A simulated annealing acceptor should be combined with a low subset selection. The classic algorithm uses a `minimalAcceptedSelection` of `1`, but usually `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>
        <completePropertyTabuSize>5</completePropertyTabuSize>
    </acceptor>
    <forager>
        <minimalAcceptedSelection>4</minimalAcceptedSelection>
    </forager>
```

This differs from phasing, another powerful technique, where first simulated annealing is used, followed by tabu search.

5.6.3. 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, weighted on their accept chance.

You can implement your own `Forager`, although the build-in forager should suffice for most needs.

5.6.3.1. Subset selection

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 `minimalAcceptedSelection` integer, which specifies how many accepted moves should have be evaluated during each step. By default it is positive infinity, so all accepted moves are evaluated at every step.

```
<forager>
  <minimalAcceptedSelection>1000</minimalAcceptedSelection>
</forager>
```

Unlike the n queens problem, real world problems require the use of subset selection. Start from an `minimalAcceptedSelection` 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.

5.6.3.2. 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>
```

5.7. Best solution

Because the current solution can degrade (especially in tabu search and simulated annealing), the local search 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.

You can listen to solver events, including when the best solution changes during solving, by adding a `SolverEventListener` to the `Solver`:

```java
public interface Solver {

    // ...

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

}
```

5.8. Termination

Sooner or later the local search solver will have 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 local search algorithm 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 would take billions of years, so you 'll want to terminate sooner anyway.

You can configure when a local search solver needs to stop by configuring a `Termination`. A `Termination` can calculate a time gradient, which is a ratio between the time already spend solving and the expected entire solving time.

You can implement your own `Termination`, although the build-in Terminations should suffice for most needs.

5.8.1. TimeMillisSpendTermination

Terminates when an amount of time has been reached:

```
<termination>
    <maximumMinutesSpend>2</maximumMinutesSpend>
```

```
    </termination>
```

or

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

Note that the time taken by a `StartingSolutionInitializer` also is taken into account by this Termination. So if you give the solver 2 minutes to solve something, but the initializer takes 1 minute, the local search solver will only have a minute left.

Note that if you use this Termination, you will most likely sacrifice reproducibility. The best solution will depend on available CPU time, not only because it influences the amount of steps taken, but also because time gradient based algorithms (such as simulated annealing) will probably act differently on each run.

5.8.2. StepCountTermination

Terminates when an amount of steps has been reached:

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

5.8.3. 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>
```

You can also use this Termination to terminate once it reaches a feasible solution. For a solver problem with hard and soft constraints, it could look like this:

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

5.8.4. UnimprovedStepCountTermination

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

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

If it 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 step tend to improve the best solution too.

5.8.5. Combining 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 ensures it doesn't just terminate after finding a feasible solution, but also makes any obvious improvements on that solution before terminating.

5.8.6. Another thread can ask a Solver to terminate early

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

```
public interface Solver {
```

```
    // ...

    boolean terminateEarly();
    boolean isTerminatedEarly();


}
```

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.

5.9. Using a custom Selector, Acceptor, Forager or Termination

It is easy to plug in a custom `Selector`, `Acceptor`, `Forager` or `Termination` by extending the abstract class and also the config class.

For example, to use a custom `Selector`, extend the `AbstractSelector` class (see `AllMovesOfOneExamSelector`), extend the `SelectorConfig` class (see `AllMovesOfOneExamSelectorConfig`) and configure it in the configuration XML:

```
s.planner.examples.examination.solver.selector.AllMovesOfOneExamSelectorConfig"/
    >
```

If you build a better implementation that's not domain specific, consider adding it as a patch in our issue tracker and we'll take it along in future refactors and optimize it.

Chapter 6. Benchmarking and tweaking

6.1. Finding the best configuration

Drools Planner supports several solver types, but you're probably wondering which is the best one? Although some solver types generally perform better then others, it really depends on your problem domain. Most solver types also have settings which can be tweaked. Those settings can influence the results of a solver a lot, although most settings perform pretty good out-of-the-box.

Luckily, Drools Planner includes a benchmarker, which allows you to play out different solver types and different settings against each other, so you can pick the best configuration for your problem domain.

6.2. Building a Benchmarker

You can build a `Benchmarker` instance with the `XmlSolverBenchmarker`. Configure it with a benchmarker configuration xml file:

```
XmlSolverBenchmarker benchmarker = new XmlSolverBenchmarker();
    benchmarker.configure("/org/drools/planner/examples/nqueens/benchmark/
nqueensSolverBenchmarkConfig.xml");
benchmarker.benchmark();
benchmarker.writeResults(resultFile);
```

A basic benchmarker configuration file looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<solverBenchmarkSuite>
    <benchmarkDirectory>local/data/nqueens</benchmarkDirectory>

    <inheritedSolverBenchmark>
            <unsolvedSolutionFile>data/nqueens/unsolved/unsolvedNQueens32.xml</
unsolvedSolutionFile>
            <unsolvedSolutionFile>data/nqueens/unsolved/unsolvedNQueens64.xml</
unsolvedSolutionFile>
        <localSearchSolver>
                        <scoreDrl>/org/drools/planner/examples/nqueens/solver/
nQueensScoreRules.drl</scoreDrl>
            <scoreDefinition>
                <scoreDefinitionType>SIMPLE</scoreDefinitionType>
            </scoreDefinition>
            <termination>
```

```xml
                    <maximumSecondsSpend>20</maximumSecondsSpend>
                </termination>
                <selector>
>org.drools.planner.examples.nqueens.solver.move.factory.NQueensMoveFactory</
moveFactoryClass>
                </selector>
                <forager>
                    <pickEarlyType>NEVER</pickEarlyType>
                </forager>
            </localSearchSolver>
        </inheritedSolverBenchmark>

        <solverBenchmark>
            <name>Solution tabu</name>
            <localSearchSolver>
                <acceptor>
                <completeSolutionTabuSize>1000</completeSolutionTabuSize>
                </acceptor>
            </localSearchSolver>
        </solverBenchmark>
        <solverBenchmark>
            <name>Move tabu 5</name>
            <localSearchSolver>
                <acceptor>
                <completeMoveTabuSize>5</completeMoveTabuSize>
                </acceptor>
            </localSearchSolver>
        </solverBenchmark>
        <solverBenchmark>
            <name>Move tabu 7</name>
            <localSearchSolver>
                <acceptor>
                <completeMoveTabuSize>7</completeMoveTabuSize>
                </acceptor>
            </localSearchSolver>
        </solverBenchmark>
        <solverBenchmark>
            <name>Solution tabu and move tabu 7</name>
            <localSearchSolver>
                <acceptor>
                <completeSolutionTabuSize>1000</completeSolutionTabuSize>
                <completeMoveTabuSize>7</completeMoveTabuSize>
                </acceptor>
            </localSearchSolver>
        </solverBenchmark>
    </solverBenchmarkSuite>
```

This benchmarker will try 4 configurations (1 solution tabu, 2 move tabu's and 1 solution-move tabu) on 2 data sets (32 and 64 queens), so it will run 8 solvers.

Every `solverBenchmark` entity contains a solver configuration (for example a local search solver) and one or more `unsolvedSolutionFile` entities. It will run the solver configuration on each of those unsolved solution files. A `name` is optional and generated if absent. The common part of multiple `solverBenchmark` entities can be extracted to the `inheritedSolverBenchmark` entity, but they can still be overwritten per `solverBenchmark` entity.

You need to specify a `benchmarkDirectory` (relative to the working directory). A summary statistic and the best solution of each solver run will be written in that directory.

6.3. Best score over time statistic (graph and CSV)

The benchmarker supports outputting the best score over time statistic as a graph and a CSV (comma separated values) file. Here's an example of a graph:
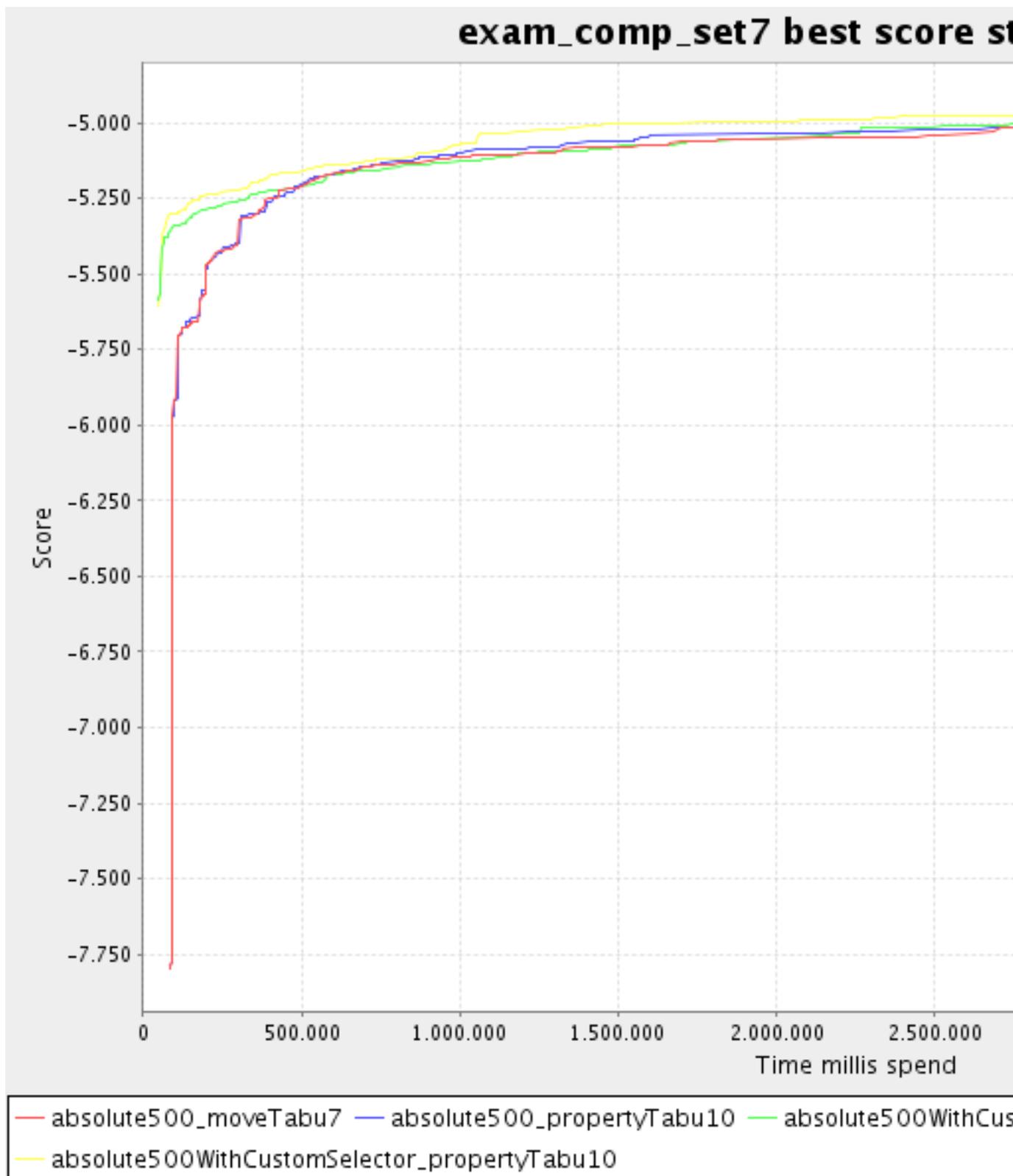
Figure 6.1. Best score over time statistic

To configure graph and CSV output of the best score over time, just add a `solverStatisticType` line:

```
<solverBenchmarkSuite>
    <benchmarkDirectory>local/data/nqueens/solved</benchmarkDirectory>
    <solverStatisticType>BEST_SOLUTION_CHANGED</solverStatisticType>
    ...
</solverBenchmarkSuite>
```

It will output all graphs and CSV files in the `benchmarkDirectory`.

Index