Chapter 1. User Guide

1.1. Solver introduction

1.1.1. What is a Solver?

Drools-solver combines a search algorithm with the power of the drools rule engine to solve planning problems. Good examples of such planning problems include:

- Employee shift rostering
- Freight routing
- Supply sorting
- Lesson scheduling
- Exam scheduling
- *The traveling salesman problem* [http://en.wikipedia.org/wiki/Travelling_salesman_problem]
- *The traveling tournament problem* [http://mat.gsia.cmu.edu/TOURN/]
- Miss manners too (although drools-solver would solve this differently than the pure drools rule engine 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. We can break down the solutions of a planning problem into 3 categories:

- 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.

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

1.1.2. Status of drools-solver

Drools-solver is an **experimental** module of Drools. The API is far from stable and backward incompatible changes occur now and then. A recipe to upgrade and apply those API changes between versions will be maintained soon.

You can download an alfa release of Drools-solver from *the drools download site* [http://www.jboss.org/drools/downloads.html].

1.1.3. Building drools-solver and running an example

You can also easily build it from source yourself. Check out drools from subversion and do a maven 2 build with the solver profile:

```
$ svn checkout http://anonsvn.jboss.org/repos/labs/labs/jbossrules/trunk/
 drools
...
$ cd drools
$ mvn -Dmaven.test.skip clean install
...
```

After that, you can run any example directly from the command line, for example to run the n queens example, run:

```
$ cd drools-solver/drools-solver-examples/
$ mvn exec:exec
 -Dexec.mainClass="org.drools.solver.examples.nqueens.app.NQueensApp"
...
```

You will use drools-solver with the latest, unstable snapshot of the drools rule engine. If you would rather use a stable version of the drools rule engine, edit `/drools-solver/pom.xml` and overwrite the drools jar versions, before building and running the examples:

```
    <dependencyManagement>
```

```
        <dependencies>
            <dependency>
                <groupId>org.drools</groupId>
                <artifactId>drools-core</artifactId>
                <version>4.0.3</version>
            </dependency>
            <dependency>
                <groupId>org.drools</groupId>
                <artifactId>drools-compiler</artifactId>
                <version>4.0.3</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

1.2. Solver examples

1.2.1. Introduction

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

1.2.2. The n queens example

1.2.2.1. Running the example

In the directory `/drools-solver/drools-solver-examples/` run the following command:

```
$ mvn exec:exec
 -Dexec.mainClass="org.drools.solver.examples.nqueens.app.NQueensApp"
...
```

1.2.2.2. Screenshot

Here is a screenshot of the example:

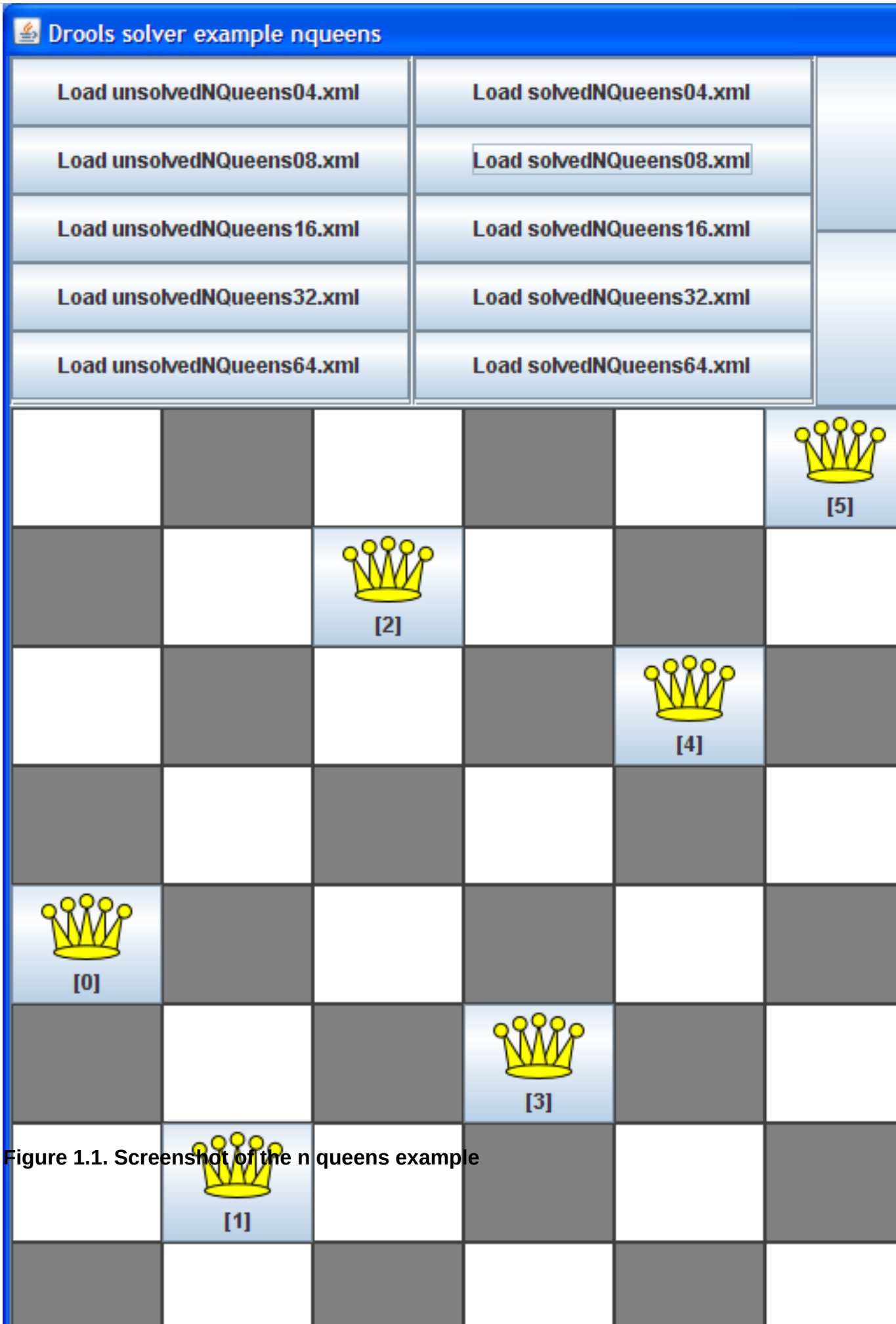


Figure 1.1. Screenshot of the n queens example

1.2.2.3. 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-solver using the 4 queens puzzle as the primary example.

A proposed solution could be:

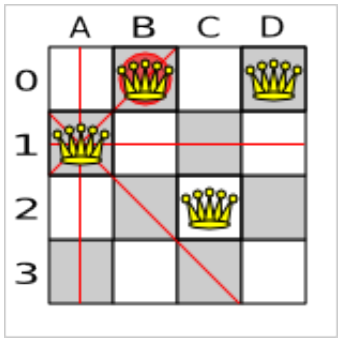


Figure 1.2. A wrong solution for the 4 queens puzzle

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.

1.2.2.4. Solution(s)

Below is a correct solution:

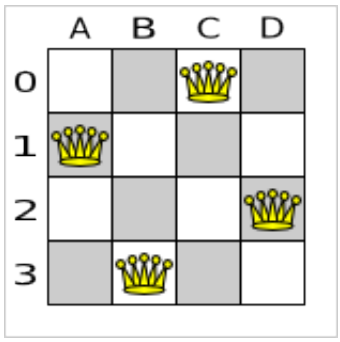


Figure 1.3. A correct solution for the 4 queens puzzle

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.

1.2.2.5. Problem size

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

Table 1.1. NQueens problem size

# queens (n)	# possible solutions (each queen it's 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	1.461501637330902918203684832e+48	?	?	?
64	3.940200619639447921227904011e+115	?	?	?
n	n ^ n	?	# feasible solutions	?

1.2.2.6. Domain class diagram

Use a good domain model and it will be easier to understand and solve your problem with drools-solver. We 'll use this domain model for the n queens example:

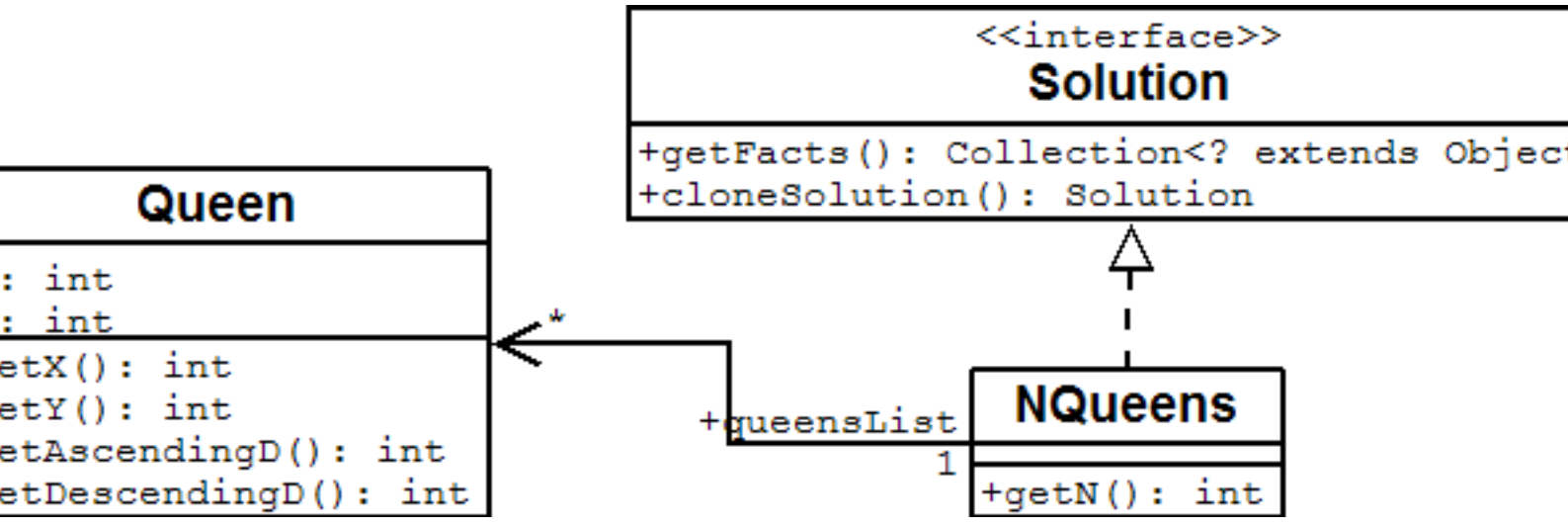
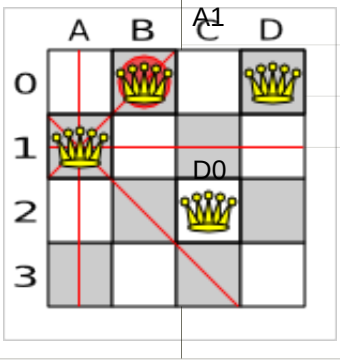


Figure 1.4. NQueens domain class diagram

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

A solution	Queen	x	y	ascendingD (x + y)	descendingD (x - y)
	A1	0	1	1 (**)	-1
		1	0 (*)	1 (**)	1
		2	2	4	0
	D0	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 drools-solver. Notice that in the 4 queens example, NQueens's `getN()` method will always return 4.

You can find the source code of this example (as well as well as several other examples) in the drools-solver-examples src distribution.

1.2.3. The lesson schedule example

1.2.3.1. Running the example

In the directory `/drools-solver/drools-solver-examples/` run the following command:

```
$ mvn exec:exec
 -
Dexec.mainClass="org.drools.solver.examples.lessonschedule.app.LessonScheduleApp"
...
```

1.2.3.2. Screenshot

Here is a screenshot of the example:

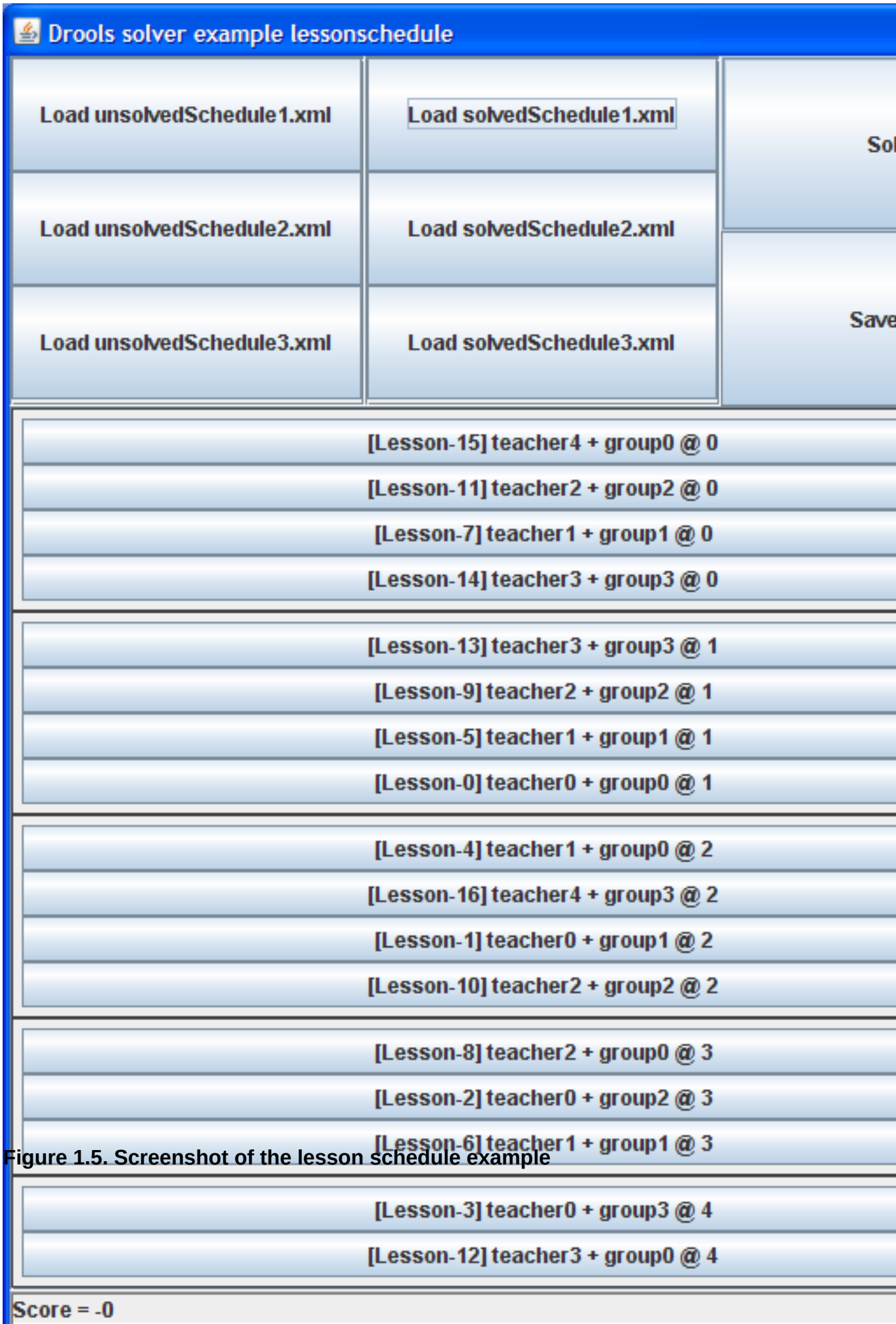


Figure 1.5. Screenshot of the lesson schedule example

1.2.3.3. Problem statement

Schedule lessons with the follow constraints:

- No teacher with 2 lessons in the same timeslot
- No group with 2 lessons in the same timeslot

1.2.4. The traveling tournament example

1.2.4.1. Running the example

In the directory `/drools-solver/drools-solver-examples/` run one of the the following commands:

```
$ mvn exec:exec
 -
Dexec.mainClass="org.drools.solver.examples.travelingtournament.app.simple.SimpleTravelingTo
...
$ mvn exec:exec
 -
Dexec.mainClass="org.drools.solver.examples.travelingtournament.app.smart.SmartTravelingTourn
...
```

The smart implementation performs and scales a lot better than the simple implementation.

1.2.4.2. Screenshot

Here is a screenshot of the example:

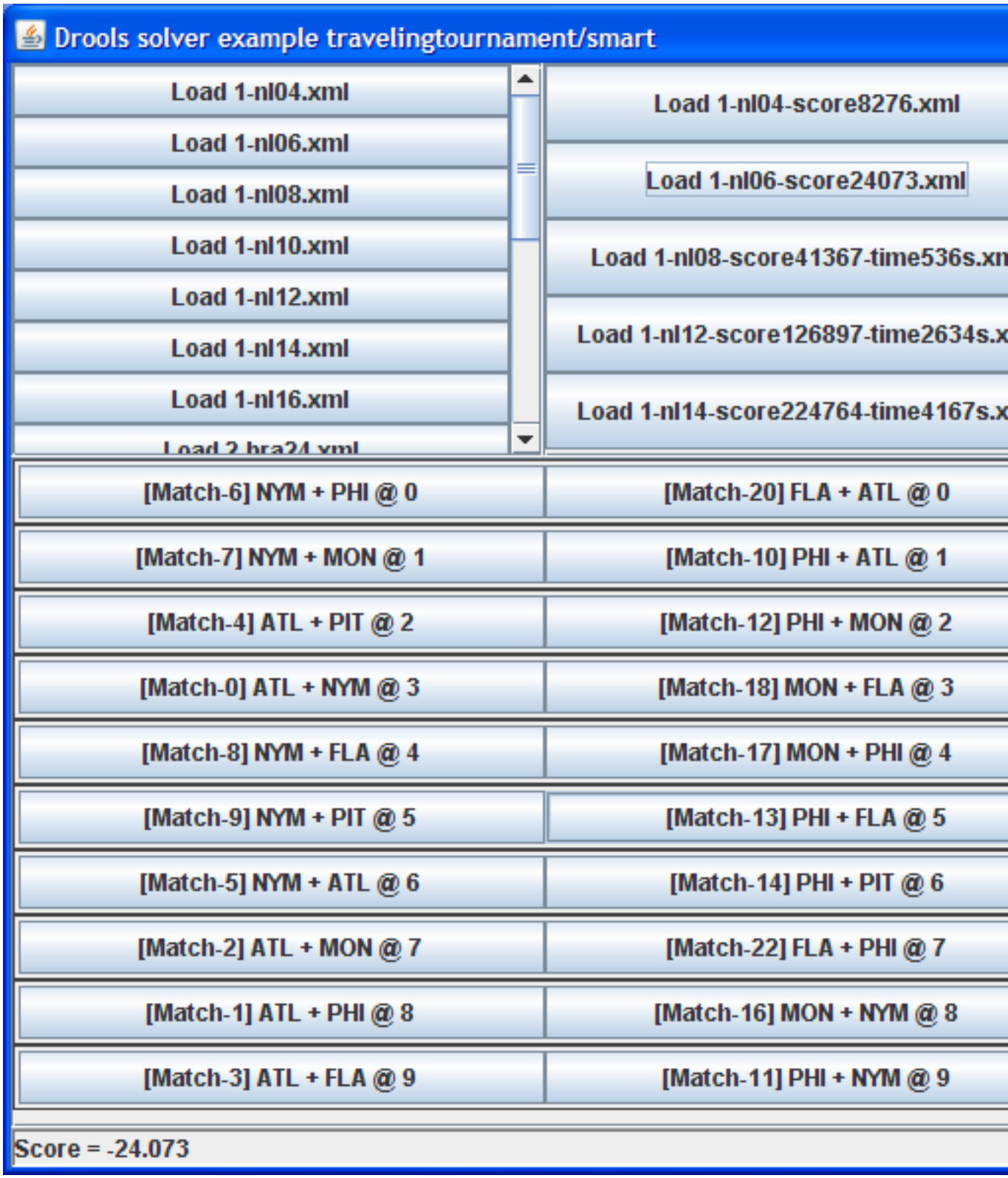


Figure 1.6. Screenshot of the traveling tournament example

1.2.4.3. Problem statement

Schedule matches between 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 playing day
- No more than 3 consecutive home or 3 consecutive away matches for any team
- No repeaters: 2 consecutive matches of the same 2 teams (so each team plays once home and once away

and the following soft constraints:

- Minimize the total distance traveled of all teams.

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

1.2.4.4. Problem size

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

Table 1.3. Traveling tournament problem size

# teams	# days	# matches	# possible solutions (simple)	# possible solutions (smart)	# feasible solutions	# optimal solutions
4	6	12	2176782336	<= 518400	?	1?
6	10	30	1000000000000	000000000000 477847258398720000	?000000000	1?
8	14	56	1.52464943788	29046560612 5.77608277425558771434498864e+43	36043e+64	1?
10	18	90	9.43029892325	5592804770 1.07573451027871200629339068e+79	52413e+112	1?
12	22	132	1.58414112478	1953204151 2.01650616733413376416949843e+126	35060e+177	1?
14	26	182	3.35080635695	1032233151 1.73513467024013808570420241e+186	89511e+257	1?
16	30	240	3.22924601799	8554007515 2.45064610271441678267620602e+259	22483e+354	1?

# teams	# days	# matches	# possible solutions (simple)	# possible solutions (smart)	# feasible solutions	# optimal solutions
n	2 * (n - 1)	n * (n - 1)	(2 * (n - 1)) ^ (n * (n - 1))	<= (((2 * (n - 1))!) ^ (n / 2))	?	1?

1.2.5. The ITC2007 examination example

1.2.5.1. Running the example

In the directory `/drools-solver/drools-solver-examples/` run the following command:

```
$ mvn exec:exec
 -
Dexec.mainClass="org.drools.solver.examples.itc2007.examination.app.ExaminationApp"
...
```

1.2.5.2. Screenshot

Here is a screenshot of the example:

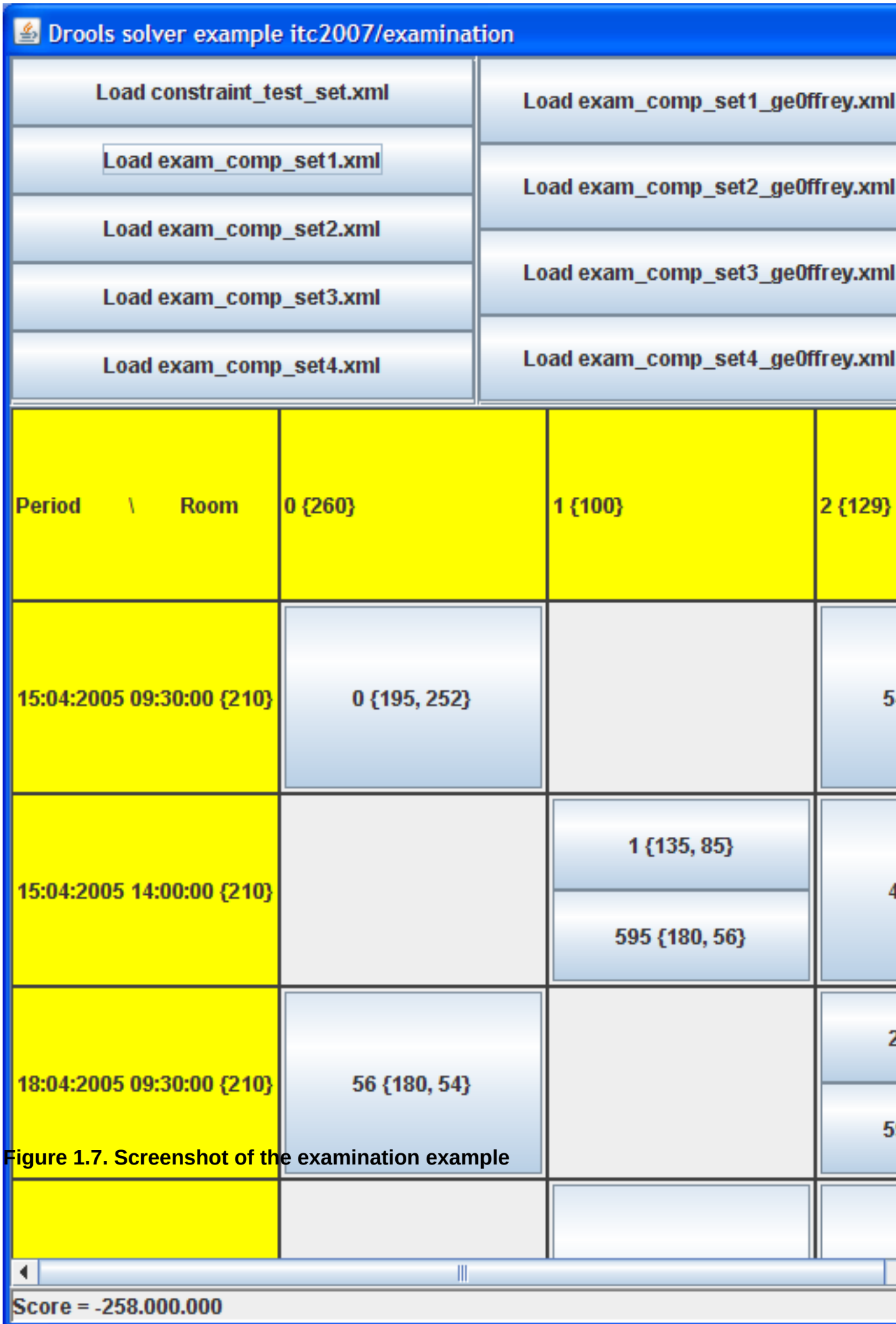


Figure 1.7. Screenshot of the examination example

1.2.5.3. 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]

1.2.5.4. Problem size

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

Table 1.4. Examination problem size

Set	# students	# exams/ topics	# periods	# rooms	# possible solutions	# feasible solutions	# optimal solutions
exam_comp_set1	7883	607	54	7	1.11000574e+1052	2.74221096e+1052	1.10367623e+1052
exam_comp_set2	12484	870	40	49	2.86903028e+5762	7.22562597e+5762	1.82749122e+5761
exam_comp_set3	16365	934	36	48	5.74648299e+5133	1.36737635e+5133	1.70728795e+5132
exam_comp_set4	4421	273	21	1	1.44349601e+52	6.26818742e+51	1.75741580e+51
exam_comp_set5	8719	1018	42	3		?	1?
exam_comp_set6	7909	242	16	8		?	1?
exam_comp_set7	13795	1096	80	28		?	1?
exam_comp_set8	7718	598	80	8		?	1?
?	s	t	p	r	(t ^ p) ^ r	?	1?

1.2.5.5. Domain class diagram

Below you can see the main examination domain classes:

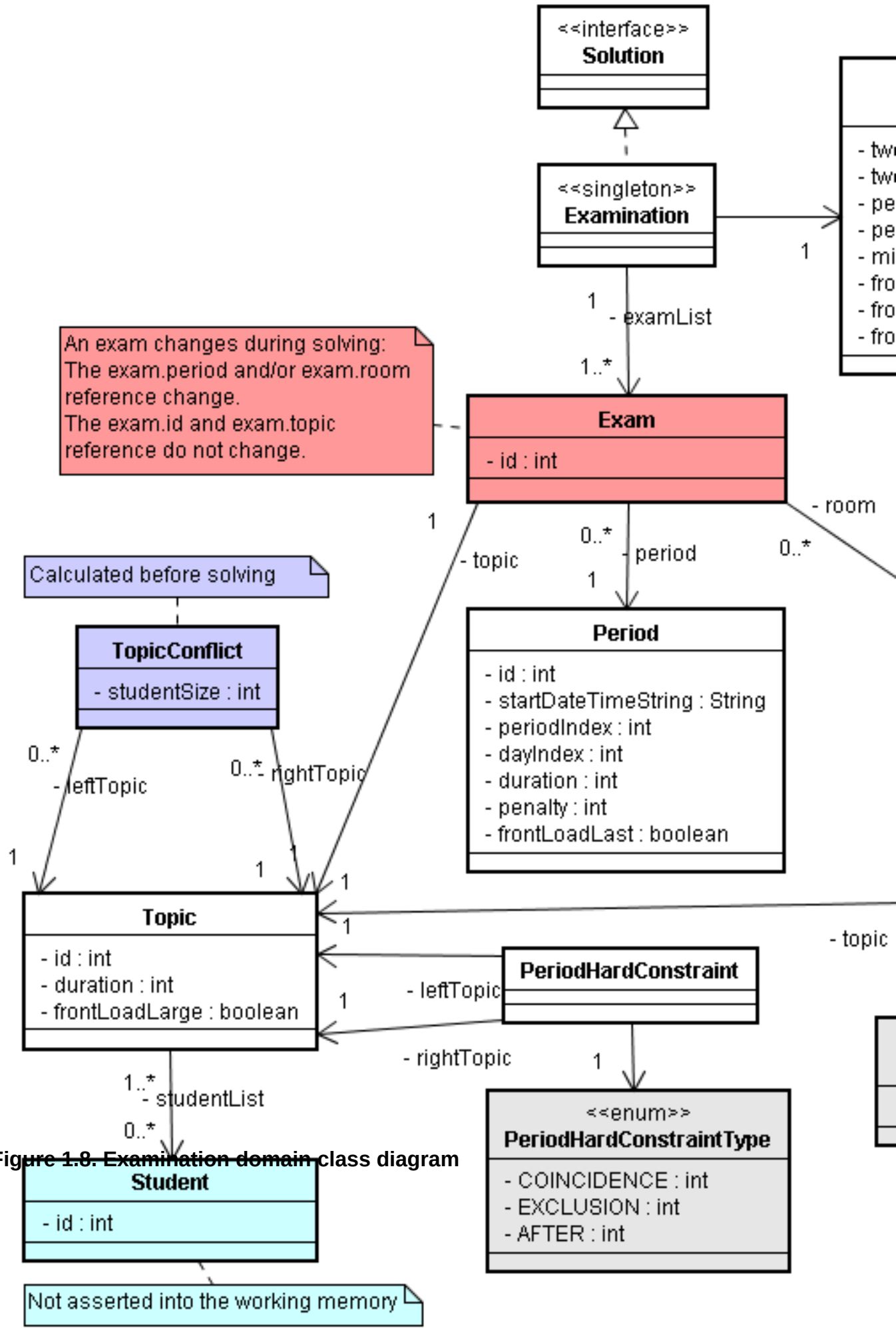


Figure 1.8. Examination domain class diagram

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

1.3. Solver configuration

1.3.1. 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.

1.3.1.1. Brute force

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

Advantages:

- It finds 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 has a horrible performance and scalability. Mostly unusable for a real-world problem due to time constraints.

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

1.3.1.2. 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 finds an optimal solution. If there is more then 1 optimal solution, it can find all optimal solutions if needed.

Disadvantages:

- It still scales very badly.

Branch and bound is currently not implemented in drools-solver.

1.3.1.3. Simplex

Simplex turns all constraints and data into a big equation, which it transmutes into a mathematical function without local optima. It then finds an optimal solution to the planning problem by finding an optima of that mathematical function.

Advantages:

- It finds an optimal solution.

Disadvantages:

- It's usually rather complex and mathematical to implement constraints.

Drools-solver does not currently implement simplex.

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

Local search starts from an initial solution and evolves that single solution into a better and better solution. It uses a single search path of solutions. At each solution in this path it evaluates a number of possible 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-solver's implementation).
- It's very scalable, even when adding extra constraints (at least in drools-solver's implementation).
- It generally needs to worry about less negative hard constraints, because the move pattern can fulfill a number of the negative hard constraints.

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-solver implements local search, including tabu search and simulated annealing.

1.3.2. The Solver interface

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

```
public interface Solver {

    void setStartingSolution(Solution solution);

    Number getBestScore();
    Solution getBestSolution();

    void solve();

    // ...

}
```

Solving a planning problem with drools-solver 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.

A `Solver` should currently directly be accessed from a single thread. Support from accessing it from a different thread, for example to finish solving early or to change the problem facts in real-time, will be added in future releases.

1.3.3. 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/solver/examples/nqueens/solver/
nqueensSolverConfig.xml");
    Solver solver = configurer.buildSolver();
```

A basic solver configuration file looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<localSearchSolver>

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

```
    <scoreCalculator>
        <scoreCalculatorType>SIMPLE</scoreCalculatorType>
    </scoreCalculator>
    <finish>
        <feasableScore>0.0</feasableScore>
    </finish>
    <selector>

 <moveFactoryClass>org.drools.solver.examples.nqueens.solver.NQueensMoveFactory</
moveFactoryClass>
    </selector>
    <accepter>
        <completeSolutionTabuSize>1000</completeSolutionTabuSize>
    </accepter>
    <forager>
        <foragerType>MAX_SCORE_OF_ALL</foragerType>
    </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-solver 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.

A solver has a single `Random` instance. Some solver configurations use that 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. In any case, during your testing it's advisable to set that `Random` instance, so your tests are reproducible.

1.3.4. The Solution interface

A Solver can only solve 1 problem 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 {

    Collection<? extends Object> getFacts();

    Solution cloneSolution();

}
```

For example, an NQueens instance just holds a list of all it's queens:

```
public class NQueens implements Solution {

    private List<Queen> queenList;

    // ...

}
```

1.3.4.1. 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;
    }
```

1.3.4.2. 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;
        return clone;
    }
```

The `cloneSolution()` method should clone no more and no less than the parts of the `Solution` that can change during solving. For example, in the lesson schedule example the lessons are cloned, but teachers, groups and timeslots are not cloned because only a lesson's appointed timeslot changes during solving:

```
    /**
     * Clone will only deep copy the lessons
     */
    public LessonSchedule cloneSolution() {
        LessonSchedule clone = new LessonSchedule();
        clone.timeslotList = timeslotList; // No Deep copy
        clone.teacherList = teacherList; // No Deep copy
        clone.groupList = groupList; // No Deep copy
```

```
        List<Lesson> clonedLessonList = new
 ArrayList<Lesson>(lessonList.size());
        for (Lesson lesson : lessonList) {
            clonedLessonList.add(lesson.clone());
        }
        clone.lessonList = clonedLessonList;
        return clone;
    }
```

1.3.5. The starting solution

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

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

1.3.6. 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).

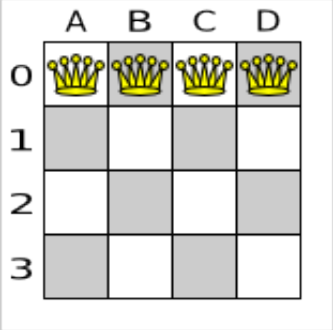


Figure 1.9. Starting solution for the 4 queens puzzle

Here's how we generate it:

```
    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.

1.3.6.1. 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 ussually 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:

```
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 intialization process.

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

```
<localSearchSolver>
    ...

 <startingSolutionInitializerClass>org.drools.solver.examples.itc2007.examination.solver.sol
startingSolutionInitializerClass>
    ...
</localSearchSolver>
```

1.3.7. 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.

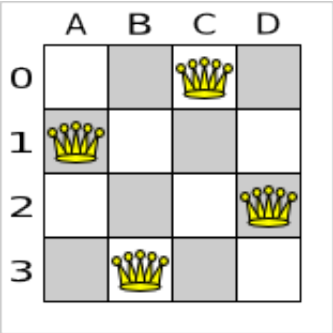


Figure 1.10. Best solution for the 4 queens puzzle (also an optimal solution)

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

1.4. Score calculation with a rule engine

1.4.1. Rule based score calculation

The score calculation of a planning problem is based on constraints (such as hard constraints, soft constraints, rewards, ...). A rules 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 drools:

```
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:

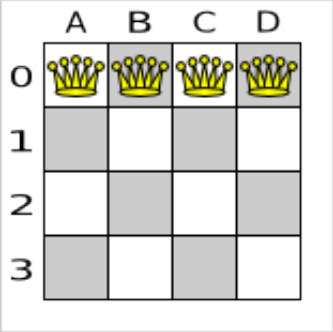


Figure 1.11. Starting solution for the 4 queens puzzle

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`.

You need to add your score rules drl files in the solver configuration, for example:

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

You can add multiple `<scoreDrl>` entries if needed.

It's recommended to use drools in forward-chaining mode (which is the default behaviour), as for most solver 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 moves from `y 0` to `3`, it won't bother to recalculate the "multiple queens on the same horizontal line" constraint for queens with `y 1` or `2`. This is a huge performance gain. **Drools-solver 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.

Adding more constraints is easy and scalable (if you understand the drools rule syntax). This allows you to add it 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 at rush hour.

1.4.2. The ScoreCalculator interface

The `ScoreCalculator` interface allows the solver to calculate the score of the currently evaluated solution. The score must a `Number` instance and the instance type (for example `Double` or `Integer`) must be stable throughout the problem.

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 that can attack each other.

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

```
    <scoreCalculator>
        <scoreCalculatorType>SIMPLE</scoreCalculatorType>
    </scoreCalculator>
```

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

- SIMPLE: A `SimpleScoreCalculator` instance which has a `setScore(Number)` method for use in the score rules.
- HARD_AND_SOFT_CONSTRAINTS: A `HardAndSoftConstraintScoreCalculator` instance, which has a `setHardConstraintsBroken(Number)` and a `setSoftConstraintsBroken(Number)` method for use in the score rules.
- DYNAMIC_HARD_AND_SOFT_CONSTRAINTS: A special `HardAndSoftConstraintScoreCalculator` instance, for more information see the javadocs.

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

The `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
```

1.4.3. 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.
- 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.
- 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.
- 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.

1.5. Local search solver

1.5.1. 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 optimal solution.

An algorithm that checks every possible solution (even with pruning) can easily run for a couple 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 making a move on the current solution to change it into a better solution. It does that number of times till 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 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-solver's local search implementation combines both. On top of that, it also offers additional support for benchmarking etc.

1.5.2. 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:

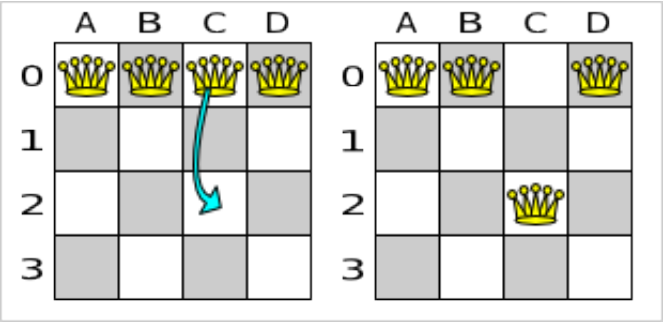


Figure 1.12. A single move (4 queens example)

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 it's current y to a different y.

Drool-solver 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);
        workingMemory.modifyRetract(queenHandle); // before changes are made
        queen.setY(toY);
        workingMemory.modifyInsert(queenHandle, queen); // after changes are
made
    }
```

Drools-solver disables shadow facts for increased performance, so you cannot use the `workingMemory.update(FactHandle, Object)` method, instead you need to call the `workingMemory.modifyRetract(FactHandle)` method before modifying the fact and the `workingMemory.modifyInsert(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-solver 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 it's 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-solver's logging more easily:

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

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

1.5.3. 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:

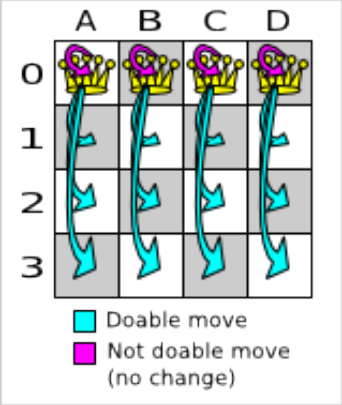


Figure 1.13. Possible moves at step 0 (4 queens example)

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`:

```
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'll be making move generation part of the drl's soon.

1.5.4. 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:

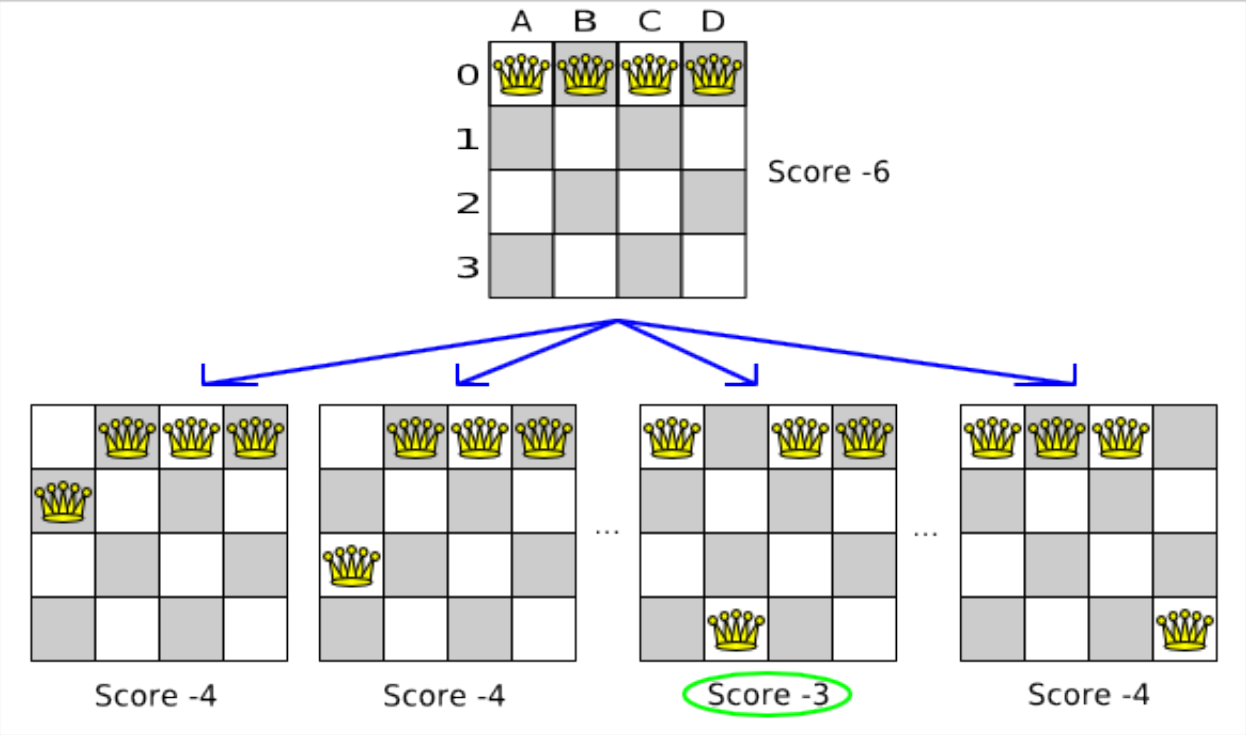


Figure 1.14. Decide the next step at step 0 (4 queens example)

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:

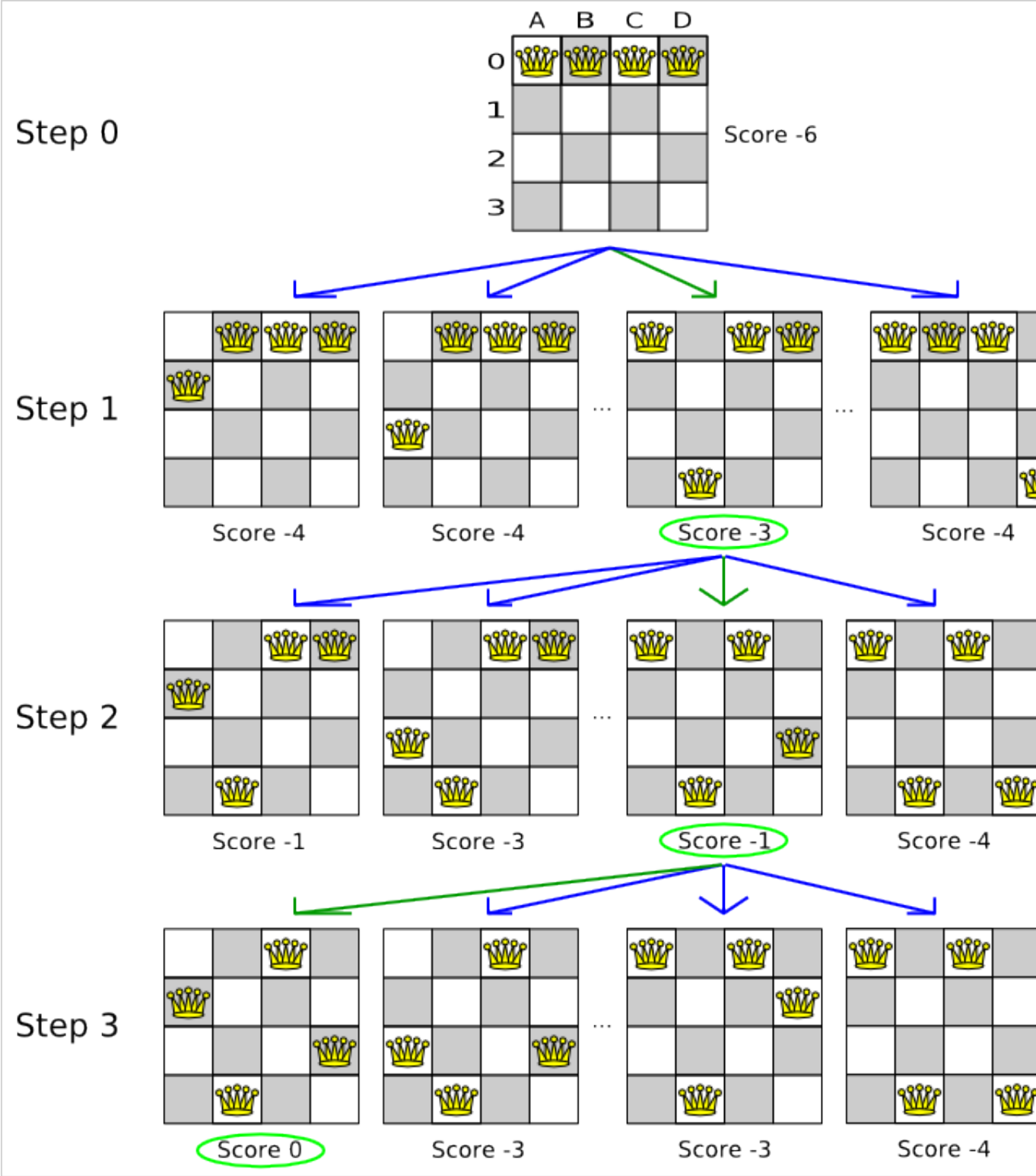


Figure 1.15. All steps (4 queens example)

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  Initial score (-6.0) is starting best score. Updating best solution
 and best score.
INFO  Step (0), time spend (0) doing next step ([Queen-1] 1 @ 0 => 3).
INFO  New score (-3.0) is better then last best score (-6.0). Updating best
 solution and best score.
INFO  Step (1), time spend (0) doing next step ([Queen-3] 3 @ 0 => 2).
INFO  New score (-1.0) is better then last best score (-3.0). Updating best
 solution and best score.
INFO  Step (2), time spend (15) doing next step ([Queen-0] 0 @ 0 => 1).
INFO  New score (0.0) is better then last best score (-1.0). Updating best
 solution and best score.
INFO  Solved in 3 steps and 15 time millis spend.
```

Notice that the logging used the `toString()` method from 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.

1.5.5. 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 it's own footsteps, picking the same next step at every step.

Of course drools-solver implements better local searches, such *tabu search* and *simulated annealing* which can avoid these problems. It's recommended to never use a simple local search, unless you're absolutely sure there are no local optima in your planning problem.

1.5.6. 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 *accepter* 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.

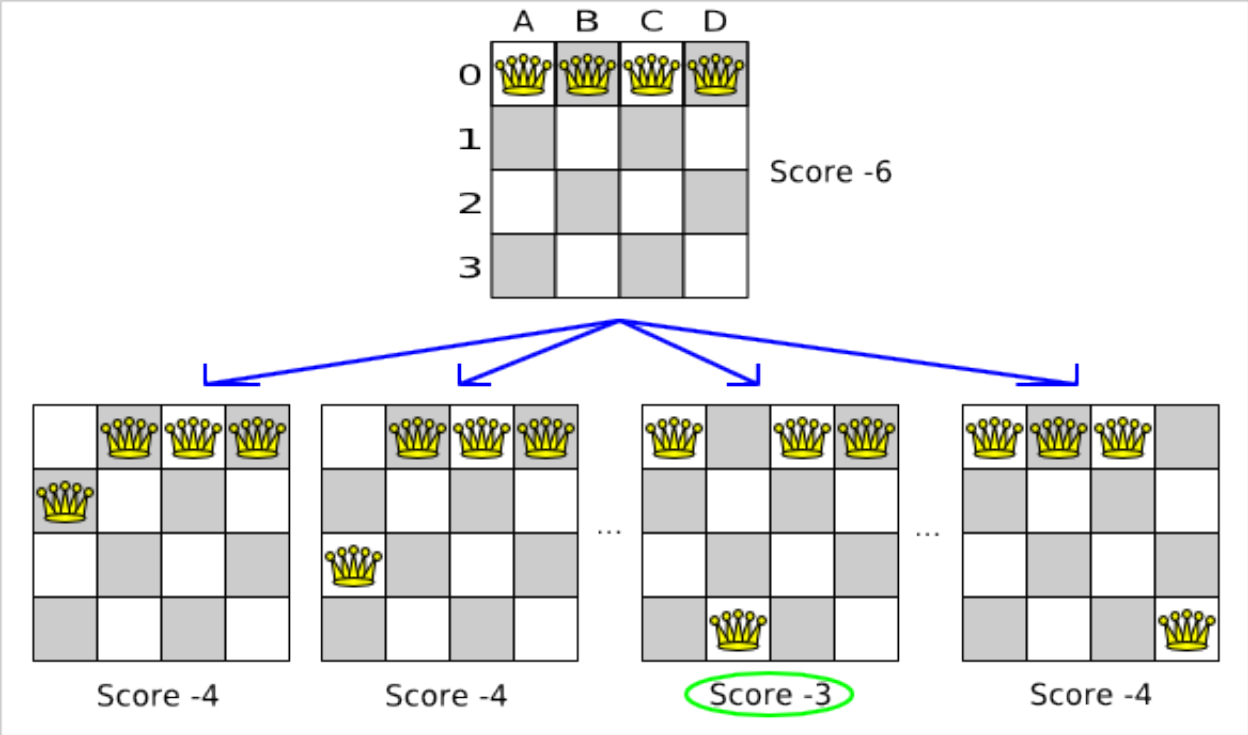


Figure 1.16. Decide the next step at step 0 (4 queens example)

In the above example the selector generated the moves shown with the blue lines, the accepter 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  Initial score (-6.0) is starting best score. Updating best solution
 and best score.
DEBUG     Move ([Queen-0] 0 @ 0 => 0) ignored because not doable.
DEBUG     Move ([Queen-0] 0 @ 1 => 1) with score (-4.0) and acceptChance
 (1.0).
```

```
DEBUG     Move ([Queen-0] 0 @ 2 => 2) with score (-4.0) and acceptChance
 (1.0).
...
DEBUG     Move ([Queen-1] 1 @ 3 => 3) with score (-3.0) and acceptChance
 (1.0).
...
DEBUG     Move ([Queen-3] 3 @ 3 => 3) with score (-4.0) and acceptChance
 (1.0).
INFO  Step (0), time spend (0) doing next step ([Queen-1] 1 @ 0 => 3).
INFO  New score (-3.0) is better then last best score (-6.0). Updating best
 solution and best score.
...
```

1.5.6.1. Selector

A selector is currently based on a `MoveFactory`. We're working on improving this.

```
    <selector>

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

You're not obligated to generate the same stable set of moves at each step. You could start with generating only big moves initially, and gradually switch to small moves. There's no build-in support for this yet though.

1.5.6.2. Accepter

An accepter 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 `Accepter`, although the build-in accepters should suffice for most needs. You can also combine multiple accepters.

1.5.6.2.1. Tabu search accepter

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-solver 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.

  ```
      <accepter>
          <completeSolutionTabuSize>1000</completeSolutionTabuSize>
  ```

```
    </accepter>
```

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

```
    <accepter>
        <completeMoveTabuSize>1000</completeMoveTabuSize>
    </accepter>
```

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

```
    <accepter>
        <completeUndoMoveTabuSize>1000</completeUndoMoveTabuSize>
    </accepter>
```

- *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.

```
    <accepter>
        <completePropertyTabuSize>1000</completePropertyTabuSize>
    </accepter>
```

  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 even combine tabu types:

```
    <accepter>
        <completeSolutionTabuSize>1000</completeSolutionTabuSize>
        <completeUndoMoveTabuSize>10</completeUndoMoveTabuSize>
    </accepter>
```

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 accepter should be combined with a *maximum score of all* or *first best score improving* forager.

1.5.6.2.2. Simulated annealing accepter

Simulated annealing does not pick the move with the highest score, neither does it evaluate all moves. At least at first.

It gives unimproving moves a chance, depending on it's score and the temperature. The *temperature* is relative to how long it has been solving. In the end, it gradually turns into a simple local search, only accepting improving moves.

A simulated annealing accepter should be combined with a *first randomly accepted* forager.

1.5.6.3. Forager

A forager gathers all accepted moves and picks the move which is the next step. A forager can choose to allow only a subset of all selected moves to be evaluated, by quitting early if a suitable move has been accepted.

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

1.5.6.3.1. Maximum score of all forager

Allows all selected moves to be evaluated and 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.

```
    <forager>
        <foragerType>MAX_SCORE_OF_ALL</foragerType>
    </forager>
```

1.5.6.3.2. First best score improving forager

Picks the first accepted move that improves the best score. If none improve the best score, it behaves exactly like the maximum score of all forager.

```
    <forager>
        <foragerType>FIRST_BEST_SCORE_IMPROVING</foragerType>
    </forager>
```

1.5.6.3.3. First last step score improving forager

Picks the first accepted move that improves the last step score. If none improve the last step score, it behaves exactly like the maximum score of all forager.

```
    <forager>
        <foragerType>FIRST_BEST_SCORE_IMPROVING</foragerType>
```

```
    </forager>
```

1.5.6.3.4. First randomly accepted forager

Generates a random number for each accepted move and if the move's accept chance is higher, it picks that move as the next move.

```
    <forager>
        <foragerType>FIRST_RANDOMLY_ACCEPTED</foragerType>
    </forager>
```

1.5.7. 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.

1.5.8. Finish

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 solver doesn't know that 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 years, so you 'll want to finish early anyway.

You can configure when a local search solver needs to stop by configuring a Finish. You can implement your own `Finish`, although the build-in finishes should suffice for most needs.

1.5.8.1. TimeMillisSpendFinish

Finishes when an amount of time has been reached:

```
    <finish>
        <maximumMinutesSpend>2</maximumMinutesSpend>
    </finish>
```

or

```
    <finish>
        <maximumHouresSpend>1</maximumHouresSpend>
    </finish>
```

Note that the time taken by a `StartingSolutionInitializer` also is taken into account by this finish. 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 finish, you will most likely sacrifice reproducability. 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.

1.5.8.2. StepCountFinish

Finishes when an amount of steps has been reached:

```
<finish>
    <maximumStepCount>100</maximumStepCount>
</finish>
```

1.5.8.3. FeasableScoreFinish

Finishes when a feasible score has been reached. You can also use this finish if you know the perfect score, for example for 4 queens:

```
<finish>
    <feasableScore>0.0</feasableScore>
</finish>
```

1.5.8.4. UnimprovedStepCountFinish

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

```
<finish>
    <maximumUnimprovedStepCount>100</maximumUnimprovedStepCount>
</finish>
```

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.

1.5.8.5. Combining finishes

Finishes can be combined, for example: finish after 100 steps or if a score of 0.0 has been reached:

```
<finish>
    <finishCompositionStyle>OR</finishCompositionStyle>
    <maximumStepCount>100</maximumStepCount>
    <feasableScore>0.0</feasableScore>
</finish>
```

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

```
    <finish>
        <finishCompositionStyle>AND</finishCompositionStyle>
        <maximumUnimprovedStepCount>5</maximumUnimprovedStepCount>
        <feasableScore>-100.0</feasableScore>
    </finish>
```

This ensures it doesn't just finish after finding a feasible solution, but also makes any obvious improvements on that solution before finishing.

Index