

WELCOME
TO
JAVAPOLIS



JAVAPOLIS  **07**
10 - 14 DECEMBER • ANTWERP • BELGIUM

Solving planning problems with Drools Solver

Geoffrey De Smet
Drools Solver lead





Overall Presentation Goal

Solve a simple planning problem
with Drools Solver



Speaker's Qualifications

- Geoffrey De Smet is a member of the drools team (in his spare time).
- Geoffrey De Smet's main job is writing government applications at Schaubroeck N.V.

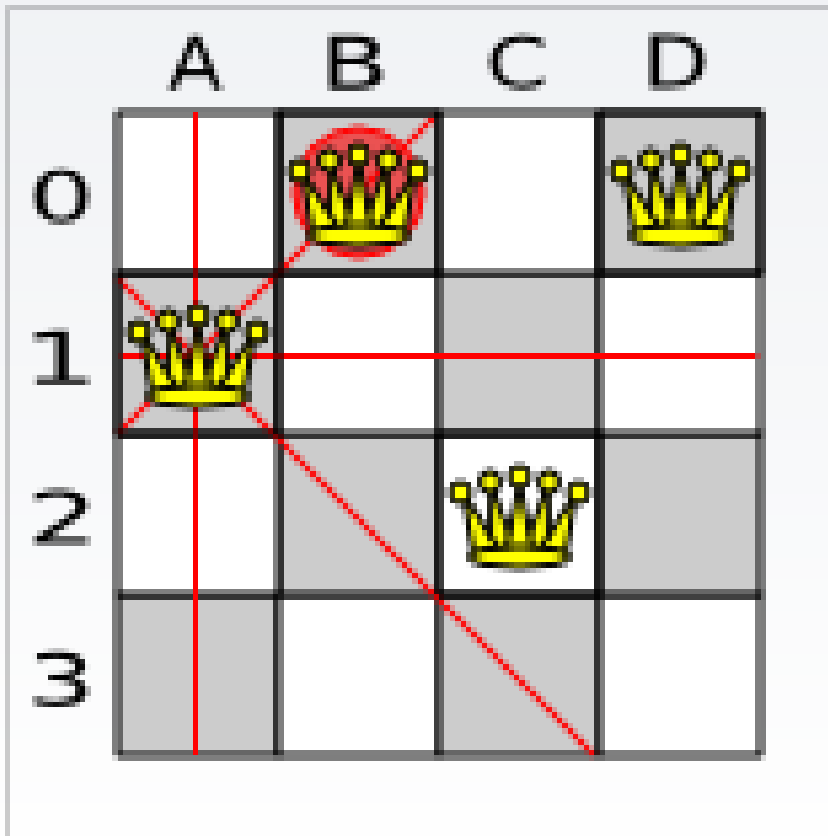


- Solve planning problems:
 - Employee shift rostering
 - Freight routing
 - Supply sorting
 - Lesson scheduling
 - Example (2 simple constraints)
 - Exam scheduling
 - Example (many complex constraints)
 - The traveling salesman problem (TSP)
 - The traveling tournament problem (= TSP++)
 - Example



Why do I need Drools Solver?
I can solve any planning problem
by brute force.

The n queens example: problem statement



- Shown with $n = 4$
- (Hard) constraints:
 - Chessboard ($n * n$)
 - Place n queens
 - No queens can attack each other







- **Hard constraint**
 - Cannot be broken
 - Any broken \Rightarrow *unfeasible solution*
 - None broken \Rightarrow *feasible solution*
- **Soft constraint**
 - Can be broken
 - Broken as little as possible
 - Least # broken (and no hard constraints broken)
 \Rightarrow *optimal (feasible) solution*



The n queens example: problem size (1/2)

- Limit # possible solutions
 - Always $n * n$ chessboard
 - Always n queens
 - Exactly 1 queen per column

	A	B	C	D
0				
1				
2				
3				



- Still n^n possible solutions

# queens	# possible	# feasible	# p / # f
4	256	2	128
8	16777216	64	262144
16	1e19	1e7	1e12
32	1e48	?	?
64	3e115	?	?
n	n^n	?	?



- Schedule exams into:
 - Periods (= timeslots)
 - Rooms
- A student has multiple exams
 - An exam has multiple students
- International timetabling competition 2007
 - Real world data (universities)
 - Real world constraints
 - Example implementation unfinished
 - Competition deadline: 25 January 2008



- Hard constraints (7):
 - Exam conflicts:
 - A student can't have 2 exams in the same period
 - Period duration limit
 - Room seating limit
 - ...
- Soft constraints (7):
 - 2 exams in a day
 - Period spread
 - Mixed durations
 - ...



- 4 datasets available at this time
- How many possible solutions?

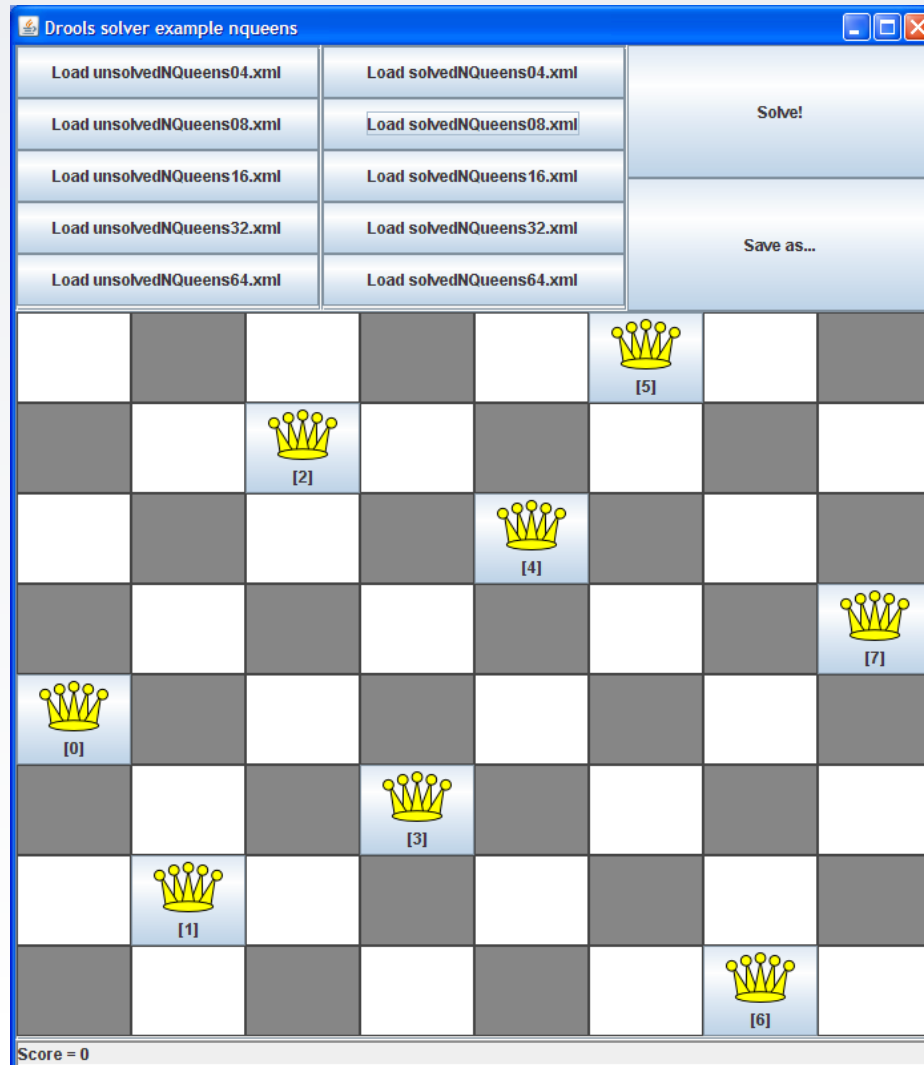
# exams	# periods	# rooms	# possible
607 (7883)	54	7	1e1052
870 (12484)	40	49	2e5761
934 (16365)	36	48	5e5132
273 (4421)	21	1	1e51
e	p	r	$(e \wedge p) \wedge r$

- Probably only a few optimal solutions each



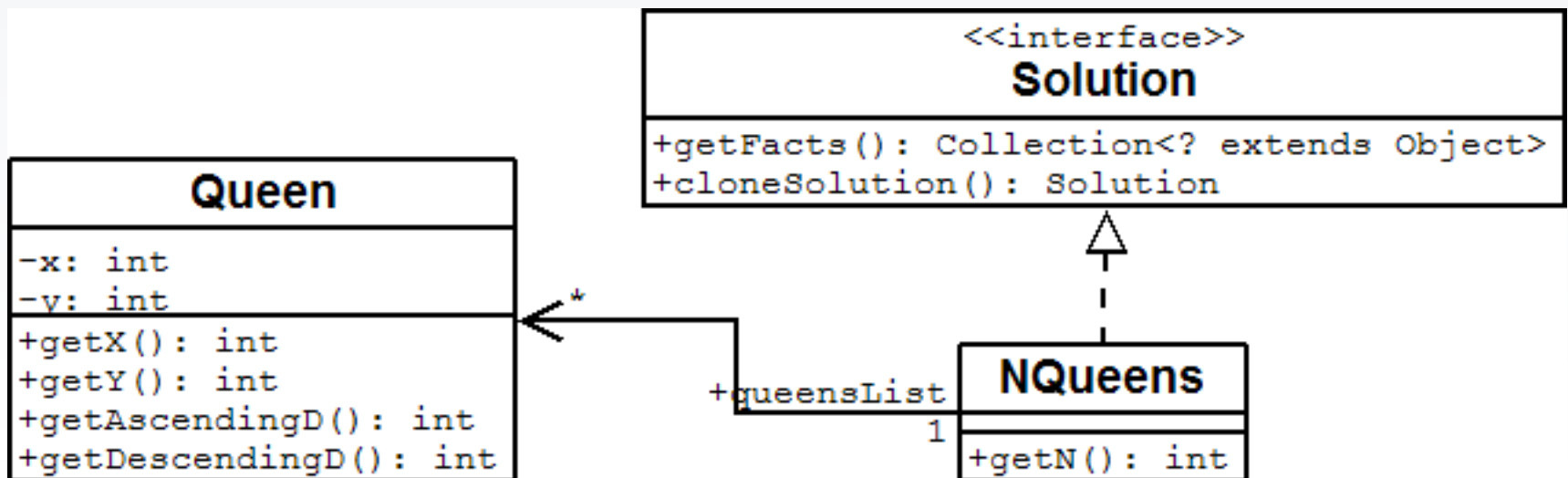
- Brute force
- Branch and bound
- Simplex
- Local search
 - Simple local search
 - Tabu search
 - Simulated annealing
- ...

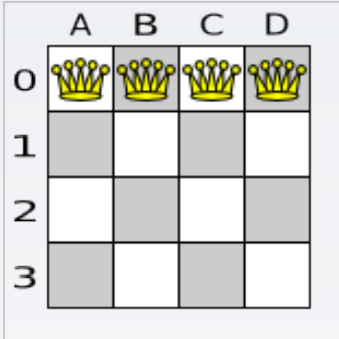
The N queens example screenshot



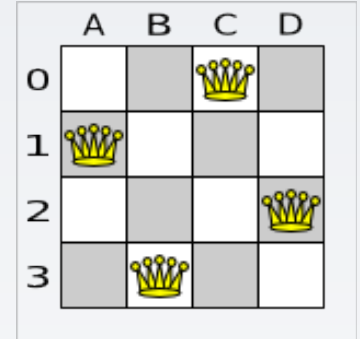


- class Queen
 - x (column), y (row)
 - Diagonal lines: ascendingD, descendingD
- class Nqueens
 - Implements Solution interface





- Build a Solver
- Set the starting solution
- Solve it
- Get the best solution found



```
XmlSolverConfigurer conf = new XmlSolverConfigurer(  
    ".../nqueensSolverConfig.xml");  
Solver solver = conf.buildSolver();
```

```
NQueens startingNQueens = ...  
solver.setStartingSolution(startingNQueens);
```

```
solver.solve();
```

```
NQueens best = (NQueens) solver.getBestSolution();
```



Solver configuration

```
<localSearchSolver>
  <scoreDrl>/.../nQueensScoreRules.drl</scoreDrl>
  <scoreCalculator>
    <scoreCalculatorType>SIMPLE</scoreCalculatorType>
  </scoreCalculator>
  <finish>
    <feasableScore>0.0</feasableScore>
  </finish>
  <selector>
    <moveFactoryClass>...NQueensMoveFactory</m...>
  </selector>
  <accepter>
    <completeSolutionTabuSize>1000</c...>
  </accepter>
  <forager>
    <foragerType>MAX_SCORE_OF_ALL</foragerType>
  </forager>
</localSearchSolver>
```



Score rules

```
global SimpleScoreCalculator scoreCalculator;

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

// multipleQueensAscendingDiagonal/DescendingDiagonal ...

rule "constraintsBroken"
    when
        $count : Number() from accumulate(
            $occurrence : ConstraintOccurrence(),
            count($occurrence)
        );
    then
        scoreCalculator.setScore(- $count.intValue());
    end
```

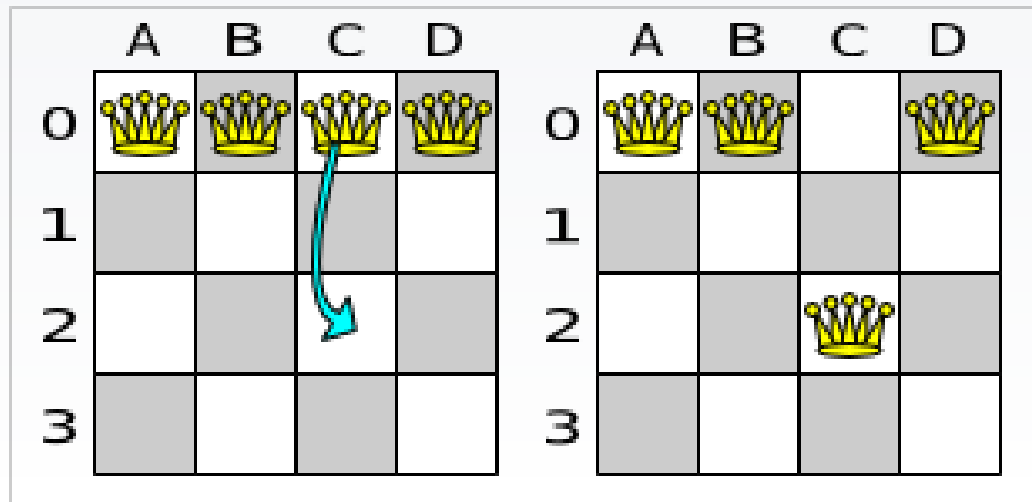


Why work with score rules?

- **Ease of development and maintenance**
 - Focused on the constraint logic
 - => DRL instead of Java
 - Separation of constraints
 - Adding/removing a constraint does not affect other constraints
- **Performance and scalability**
 - Drools rule engine pattern matching
 - Faster than for/while loops and hashmaps
 - Delta based score calculation
 - Moving queen B won't re-evaluate if queen A and queen C are on the same row



- Changes one solution into another solution
 - YChangeMove: move a queen to a different row
 - Other move types possible
- Has an undo move
 - Does the exact opposite





Move implementation

```
public class YChangeMove implements Move {

    private Queen queen;
    private int toY;

    public void doMove(WorkingMemory wm) {
        FactHandle queenHandle = wm.getFactHandle(queen);
        queen.setY(toY);
        wm.update(queenHandle, queen);
    }

    public boolean isMoveDoable(WorkingMemory wm) {
        return queen.getY() != toY;
    }

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

    // constructor, equals, hashCode
}
```

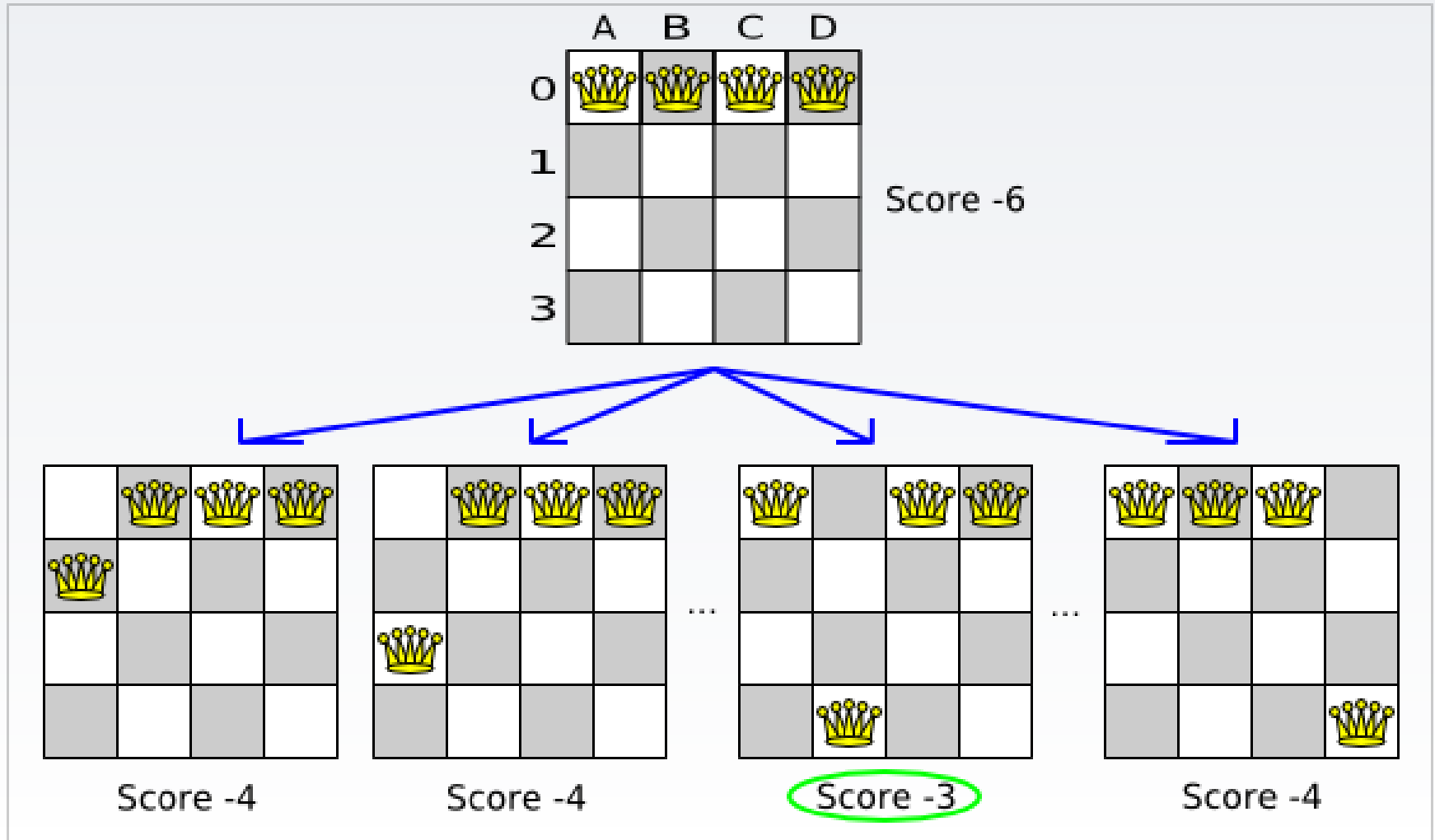


■ MoveFactory

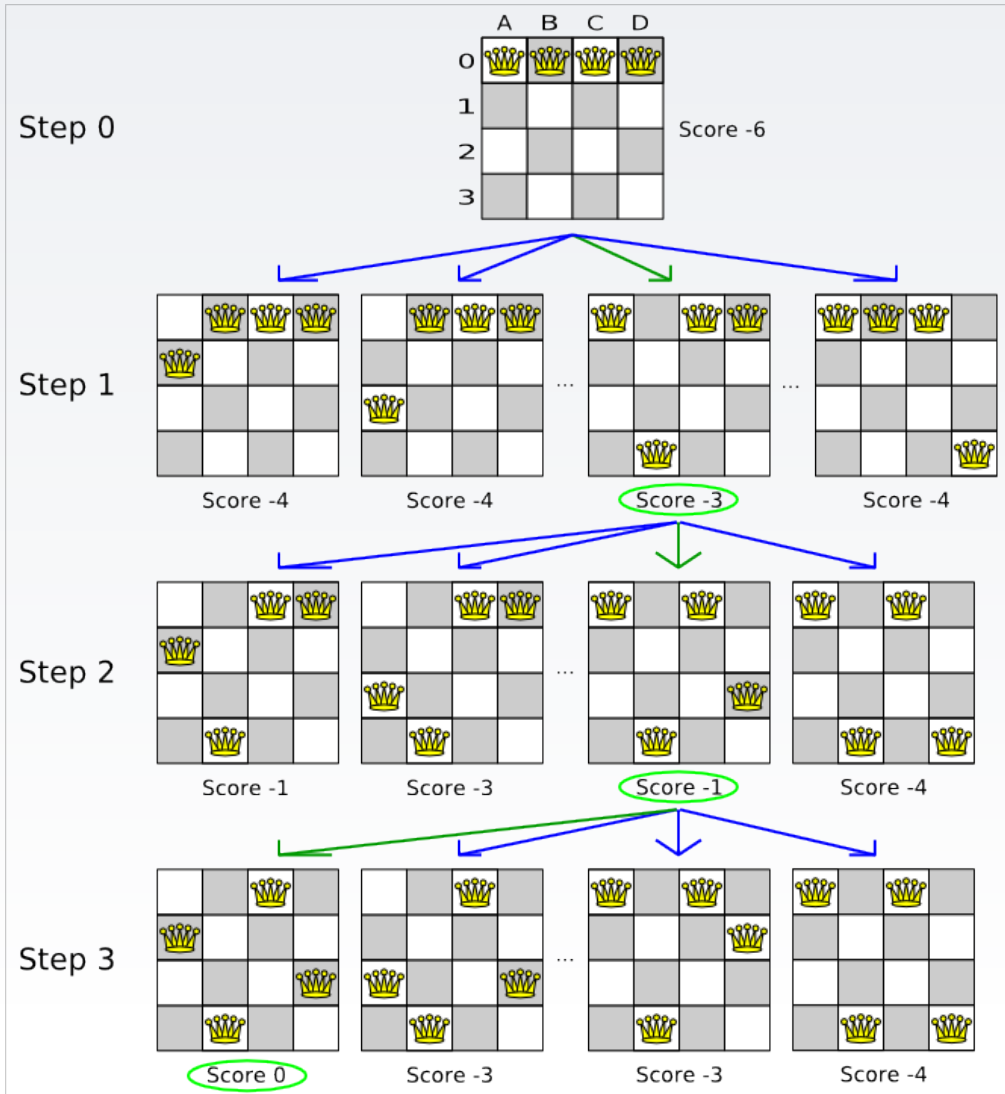
- Configured as selector in the configuration file
- CachedMoveListMoveFactory
 - createMoveList() method

```
for (Queen queen
    : nQueens.getQueenList()) {
    for (int n : nQueens.createNList()) {
        moveList.add(
            new YChangeMove(queen, n));
    }
}
```

Local search: deciding the next step



Local search: taking steps



■ Search path

- No search tree

■ 4 queens

- Solved in 3 steps
- Evaluated 37 out of 256 solutions

■ 16 queens

- 31 steps
- 7441 out of 1e19



- Simple local search = hill climbing
 - Gets stuck in local optima
 - Fixed by using an acceptor
- Tabu search acceptor
 - Solution tabu
 - Don't visit the same solution twice
 - Move tabu
 - Property tabu
- Simulated annealing acceptor
 - Randomly move around
 - Higher acceptance chance for improving moves



- **Finish**
 - Time spend finish
 - Step count finish
 - Score reached finish
 - Unimproved step count finish
 - Composition finish
 - Score reached and unimproved step count finish
- **Best solution**
 - Best encountered solution
 - Can be the optimal solution



Summary

- No official releases yet
- Download drools solver
 - Part of drools trunk in subversion
 - <http://labs.jboss.com/drools/subversion.html>
 - Build and run with maven 2
- Read the solver manual
 - Linked from blog
 - http://users.telenet.be/geoffrey/tmp/solver/manual/html_single/
- Read the drools team blog
 - <http://blog.athico.com> (label solver)

Solve your planning problems with
Drools Solver.

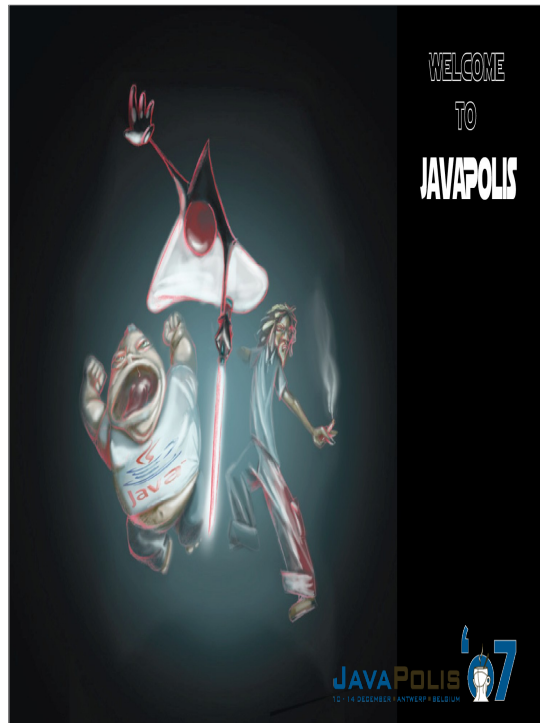
Q&A

View JavaPolis talks @ www.parleys.com



Thank you for your
attention





Welcome everybody.

Don't you just hate it?

You come to JavaPolis, you have like 5 talks you really want to attend, and what happens?

2 talks are scheduled at the same time.

That just sucks. You have to miss out on one.

Wouldn't it be great if we could all send in our favorite talks and they could schedule them, and try to avoid that people miss out on any?

Wouldn't even be better, if they could also try to avoid that people have to switch rooms between talks? I just hate having to get out my seat. Those seats are so comfortable.

Oh yea, and immediatly after lunch: no difficult talks; just standard edition stuff, no JEE talks. After lunch, I want relax a bit.

Well,

If I had to schedule the talks with those constraints, I wouldn't want to do that manually, because I am lazy. I 'd use some software to do that for me...



So, my name is Geoffrey De Smet and today I 'll talk a bit about drools-solver, which can solve planning problems.


Overall Presentation Goal

Solve a simple planning problem
with Drools Solver

www.javapolis.com


For starters, I 'll show some planning problem examples, to give you some insight on the sheer size of these problems.

After that introduction, I 'll take you through an implementation of a simple planning problem, namely the n queens problem. By the end of this presentation you should be able to start solving planning problems with Drools Solver yourself.

Speaker's Qualifications

- Geoffrey De Smet is a member of the drools team (in his spare time).
- Geoffrey De Smet's main job is writing government applications at Schaubroeck N.V.


www.javapolis.com

JAVAPOLIS


Ok, so this the mandatory slide where I have speak about myself in the 3th person, which I won't.

Anyway, I wrote 99% of the drools-solver code at the moment.
In the rest of the drools code, I help a little bit with the build process.
Drools solver uses the drools rule engine for most of the hard work.
[We actually have the drools rule engine project lead in the room here: Mark Proctor from JBoss. If you have any difficult, mind-breaking, rule-engine related questions, ask him instead of me. :)]

In my main job at Schaubroeck I am developing business applications for regional government institutions.

 Purpose of Drools Solver

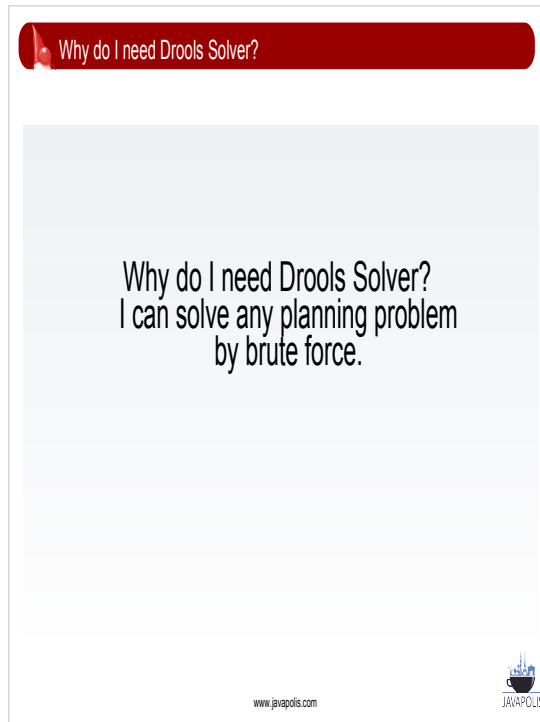
- Solve planning problems:
 - Employee shift rostering
 - Freight routing
 - Supply sorting
 - Lesson scheduling
 - Example (2 simple constraints)
 - Exam scheduling
 - Example (many complex constraints)
 - The traveling salesman problem (TSP)
 - The traveling tournament problem (= TSP++)
 - Example

www.javapolis.com

So what can Drools Solver do for you?

It can help you solve planning problems, such as:

- Employee shifts rostering
- Freight routing
- Supply sorting
- Lesson scheduling – a simple version of lesson scheduling is actually included in the examples. It has 2 basic constraints:
 - no teacher has to teach 2 lessons at the same time.
 - And no student group has to attend 2 lessons at the same time.
- Another included example is Exam scheduling – which we'll take a look at later
- Not yet included is The traveling salesman problem
- but it's advanced form, The traveling tournament problem is included. The traveling tournament problem is about scheduling football matches and minimizing the distance traveled by the football teams.
 - It's very interesting but I won't show it today.



You might be thinking:

“Why do I need Drools Solver?

I can solve any planning problem by brute force.”

I 'll just throw together some program loops and voila: that should solve that planning problem.

Let's take a look at some of the examples implemented in drools-solver, and see how far this attitude will get you.

Let's start with a simple one: the N queens problem.

The n queens example: problem statement

- Shown with $n = 4$
- (Hard) constraints:
 - Chessboard ($n * n$)
 - Place n queens
 - No queens can attack each other

www.javapoints.com

In this toy example the idea is to place a number of queens on a chessboard. That number of queens is n and it's shown here with n being 4.

Of course, there are a number of constraints, which limit you in how you can place those queens:

You must use a chessboard with n rows and n columns.

And you must place n queens on that chessboard.

Also, any 2 queens must not be able to attack each other. That's of course the difficult part.

In a solution shown on the left, you can see that the first 2 constraints are met: there are 4 rows, 4 columns and 4 queens; but the 3th constraint is broken because queen A1 and queen B0 can attack each other.

N queens only has hard constraints: it has no soft constraints.

Types of constraints

- Hard constraint
 - Cannot be broken
 - Any broken => *unfeasible solution*
 - None broken => *feasible solution*
- Soft constraint
 - Can be broken
 - Broken as little as possible
 - Least # broken (and no hard constraints broken)
=> *optimal (feasible) solution*

www.javapolis.com

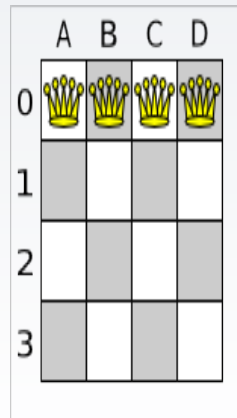
So what is a hard constraint?

A hard constraint cannot be broken. For example, a teacher can only teach 1 lesson at a time.

A soft constraint, on the the other hand, can be broken, but we should try to avoid breaking it as much as possible. For example, swimming lessons are best not thaught after lunch.

We are going to look for a solution without a single hard constraint broken. Such a solution is called a feasible solution.

And, on top of that, we are going to look for the optimal solution, which is the feasible solution with the least number of soft constraints broken.



■ Limit # possible solutions

- Always $n \times n$ chessboard
- Always n queens
- Exactly 1 queen per column

www.javapolis.com




If we do tackle the n queens problem with brute force, how many solutions can we expect to evaluate before finding a feasible solution?

To limit the number of possible solutions to evaluate, we'll always use the correct chessboard size and the correct number of queens.

Also, we'll assign each queen it's own column.

So one possible solution is shown on the left.


Another would be with all queens on the diagonal line going from the top left to the bottom right. Both solutions are unfeasible because several queens can attack each other.


The n queens example: problem size (2/2)

■ Still n^n possible solutions

# queens	# possible	# feasible	# p / # f
4	256	2	128
8	16777216	64	262144
16	1e19	1e7	1e12
32	1e48	?	?
64	3e115	?	?
n	n^n	?	?

www.javapolis.com



Even with these limitations, there are N to the power N possible combinations.

For 4 queens, that makes 256 possible solutions, of which 2 are feasible.
On average, a brute force algorithm would need to evaluate 128 solutions to find a feasible solution.

For 8 queens that makes 16 million possible solutions.
For 16 queens it's 10 to the power 19. That's a really big number.
And for 64 queens, it's more than a 3 with 115 zero's behind it.

Notice how the chance to encounter a feasible solution with brute force turns bad quickly.
And don't forget: every single one of those solutions needs to be evaluated, usually by nested program loops.
How long do you think brute force will take to solve 64 queens?

Drools Solver finds a feasible solution for the 64 queens problem in less than a minute, on my 2 year old desktop pc. Of course, it doesn't use brute force.

Now, let's take a look at a real example, which will make these numbers look like a joke.

The ITC2007 examination example: problem statement (1/2)

- Schedule exams into:
 - Periods (= timeslots)
 - Rooms
- A student has multiple exams
 - An exam has multiple students
- International timetabling competition 2007
 - Real world data (universities)
 - Real world constraints
 - Example implementation unfinished
 - Competition deadline: 25 January 2008

www.javapolis.com

The examination example schedules exams.
Each exam needs to be scheduled
into a period
and into a room.
Each exam has a duration and multiple students.
Each student has multiple exams.


I am actually competing in the international timetabling competition 2007 with this example.
They defined the problem statement.
They use data and constraints from real universities.
For more information, there's a link in the solver manual.

I have just started working on it, but expect some results soon, because the competition deadline is in a month.
By the way, there are 2 other competition tracks, which I won't compete in due to lack of time, which also fit Drools Solver like a glove and also have a main prize of 500 pounds each. Anyone interested?

The ITC2007 examination example: problem statement (2/2)

- Hard constraints (7):
 - Exam conflicts:
 - A student can't have 2 exams in the same period
 - Period duration limit
 - Room seating limit
 - ...
- Soft constraints (7):
 - 2 exams in a day
 - Period spread
 - Mixed durations
 - ...

www.javapolis.com



So what are some of the constraints?

Exam conflicts:

Student John Smith can't have 2 exams at the same time.

Period duration limit:

The Math exam takes 2 hours, so it can't fit in a period of only 1 hour long.

Room seating limit:

55 students taking 3 different exams at the same time can't fit into a single room with only 40 seats.

2 exams in a row:


John Smith doesn't like to have 2 exams on the same day.

Period spread:

John Smith prefers 4 free days between 2 consecutive exams.

In total there are about 7 hard constraints and 7 soft constraints.


The constraints are weighted, for example the “period spread” constraint has a lower weight than the “2 exams in a day” constraint.


The ITC2007 examination example: problem size

- 4 datasets available at this time
- How many possible solutions?

# exams	# periods	# rooms	# possible
607 (7883)	54	7	1e1052
870 (12484)	40	49	2e5761
934 (16365)	36	48	5e5132
273 (4421)	21	1	1e51
e	p	r	$(e^p)^r$

- Probably only a few optimal solutions each


JAVAPOLIS

www.javapolis.com

There are 4 datasets available at this time. Let's take a look at one.

The first dataset has about 600 exams, 7000 students, 50 periods and 7 rooms. It's number of possible solutions is about a 1 with a 1052 zero's behind it. N queens, eat your heart out!

Let's say we use a smarter form of brute force, and we're able to ignore 99% of those possible solutions, that would still leave a 1 with a 1050 zero's behind it. Let's say we can evaluate a billion solutions every millisecond, which we can't. It would still take over 10 to the power a 1030 years to evaluate all possible solutions. The earth isn't even 10 to the power 10 years old.

And if you look at the second dataset, with 800 exams and 12'000 students, you 'll see that it can get a lot worse easily.

Unlike n queens, there's usually only 1 or a few optimal solutions. Most universities however, would be happy if we'd find a relatively good feasible solution, better and faster than their current manual planning.

So, how do we solve these kind of problems efficiently?

Types of solvers

- Brute force
- Branch and bound
- Simplex
- Local search
 - Simple local search
 - Tabu search
 - Simulated annealing
- ...

www.javapolis.com

JAVAPOLIS

Well, each type of solver has its own algorithm to find the best solution out of all possible solutions.

Brute force evaluates every possible solution, usually by creating a search tree. It's useless for real-world use cases, as I hope I have demonstrated, but it's useful to verify the output of other implementations.

It's not implemented in drools-solver yet, but it will be at some point.

Branch and bound is an improvement over brute force, as it prunes away part of the search tree. However, it still doesn't scale enough to be useful

It's also not implemented in drools-solver yet.

Simplex turns all constraints and data into one big equation. It then uses a search path to find the optimum of that equation.

As far as I know, it's very limiting on how you implement your constraints.

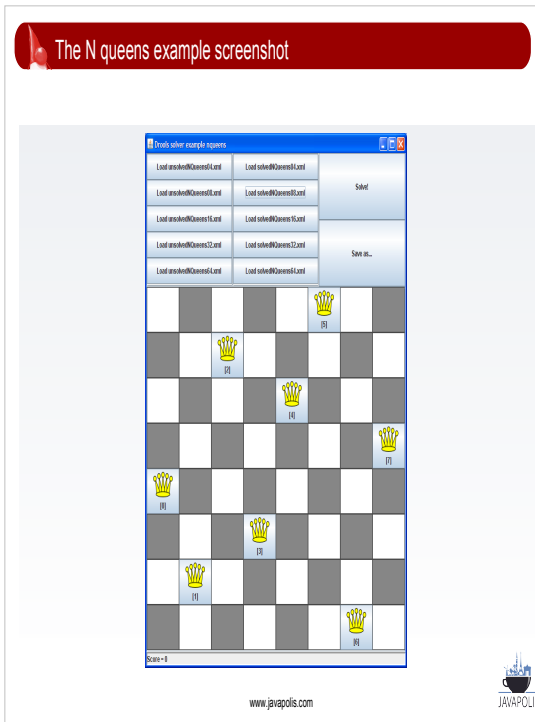
It's also not implemented in drools-solver yet.

Local search however, is implemented in drools-solver.

Local search works pretty much like most humans: it starts from an initial solution and it evolves that single solution into a better and better solution. It uses a single search path and it's very scalable.

There are different forms of local search, such as tabu search and simulated annealing. These are implemented in drools-solver too.

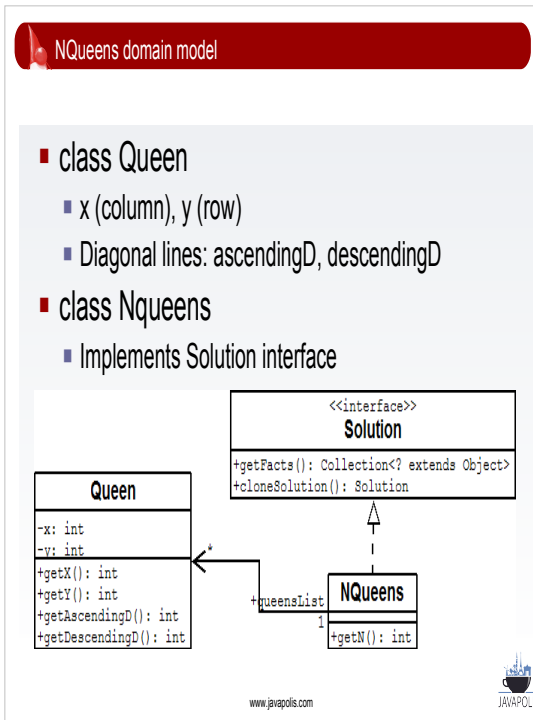
Let's take a look at the n queens example implementation, configured with tabu search.



Each of the included examples in drools-solver has a raw GUI.
This is a screenshot of the n queens GUI.

With the top left buttons you can load a problem instance.
In this case 8 queens is loaded.

With the top right button you can solve an instance.
As you can see, in this case it's already solved.
Notice that the score on the bottom left is 0, because no constraints are broken.
We'll use the drools rule engine to calculate that score.



The Queen class (on the left) represents a single Queen.

A queen instance has an x and a y. X is the column, y the row.

Because we'll appoint each queen its own column, the x property of a queen will actually never change.

Each queen can calculate its horizontal line (y), its vertical line (x), its ascending diagonal line ($x + y$) and its descending diagonal line ($x - y$).

So if 2 queens share any of those lines, they can attack each other.

The Nqueens class (on the bottom right) represents a single solution instance.

It has a list of n - queen instances and implements the Solution interface.


A NQueens instance can represent the problem in a state that is solved or unsolved or anything in between.




We will start with an unsolved NQueens instance, which has all queens on the first row in a different column.

When we solve that NQueens instance, the y of its queen instances will change.





Notice that a Solution needs to implement the getFacts method, so the solver can supply facts to the drools rule engine for score calculation. NQueens implements that method by simply returning the queensList.

Also, a Solution needs to be cloneable, because a solver needs to be able to remember the best encountered solution. This cloneSolution method only needs to deep clone the changing parts. For NQueens however, this means cloning all Queen instances because their y's change during solving.



	A	B	C	D
0				
1				
2				
3				

- Build a Solver
- Set the starting solution
- Solve it
- Get the best solution found

	A	B	C	D
0				
1				
2				
3				


```
XmlSolverConfigurer conf = new XmlSolverConfigurer(
    ".../nqueensSolverConfig.xml");
Solver solver = conf.buildSolver();

NQueens startingNQueens = ...
solver.setStartingSolution(startingNQueens);

solver.solve();

NQueens best = (NQueens) solver.getBestSolution();
```

www.javapolis.com



So how do we solve a problem?

First we build a solver, usually based on an XML configuration file.
In that configuration file we configure the solver type, time limitations, etc.

Then we load or generate the starting solution, which you can see on the top left.
For 4 queens, we use an NQueens instance with 4 Queen instances, of which each y is 0 and the x is from 0 to 3.

Next, we solve it. For non trivial problems, it could take some time before this method returns.

And then we get the best solution, which you can see on the top right.
As you can see, no 2 queens can attach each other, so it's completely solved.

Let's take a look at that configuration file.

 Solver configuration

```
<localSearchSolver>
  <scoreDrl>.../nQueensScoreRules.drl</scoreDrl>
  <scoreCalculator>
    <scoreCalculatorType>SIMPLE</scoreCalculatorType>
  </scoreCalculator>
  <finish>
    <feasibleScore>0.0</feasibleScore>
  </finish>
  <selector>
    <moveFactoryClass>...NQueensMoveFactory</m...>
  </selector>
  <accepter>
    <completeSolutionTabuSize>1000</c...>
  </accepter>
  <forager>
    <foragerType>MAX_SCORE_OF_ALL</foragerType>
  </forager>
</localSearchSolver>
```


www.javapolis.com

Here we configure a local search solver. The red text is domain specific.

Any solver that we configure, will need a way to calculate the score of a solution. If it can't do that, it has no way of comparing solutions and recognizing the best one.

Drools solver uses the drools rules engine to calculate the score, based on a DRL file with score rules. We 'll take a look at those N Queens score rules in the next slide.

Usually we use a `hard_and_soft_constraints` score calculator, but because N queens only has hard constraints, a simple score calculator is enough.

Local search doesn't recognize the optimal solution when it finds it, it just continues to try to find a better, non existing solution. This might seem like a big disadvantage of local search. But for real use cases, you won't have time to wait a billion years to find that optimal solution anyway.

So you need to tell the solver to stop at some point. Drools solver supports several ways to finish solving, such as: time spend, score reached, etc.

In this case we know the perfect score is zero, so we 'll finish solving once we reach a score of 0.

To evolve the starting solution into the best solution, local search uses a move selector, a move accepter and a move forager.

Here, we're using a tabu accepter and a max score forager, which makes it tabu search. By simply changing the accepter and forager, we can make it simulated annealing, etc without changing anything to the code.

Score rules


```
global SimpleScoreCalculator scoreCalculator;

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

// multipleQueensAscendingDiagonal/DescendingDiagonal ...

rule "constraintsBroken"
when
    $count : Number() from accumulate(
        $occurrence : ConstraintOccurrence(),
        count($occurrence)
    );
then
    scoreCalculator.setScore(- $count.intValue());
end
```

www.javapolis.com

JAVAPOLIS

This the DRL file which defines the score rules.

If you haven't worked with the drools rules engine before, this might seem very strange and complicated. But in fact, for non trivial use cases, this makes score calculation easier and a lot more scalable.

A single global scoreCalculator is available for use in the rules.

The score rules need to keep the score of the scoreCalculator up to date.

All the Queen instances are asserted into the drools working memory, which means they can be used for declarative pattern matching.

Every time the solver moves a queen to a different row, it fires the rules and expects to find the correct score in the scoreCalculator afterwards.

The multipleQueensHorizontal rule fires when 2 different queens are positioned on the same y, so on the same horizontal line. It inserts a logical ConstraintOccurrence. So if and only if 2 queens are on the same row, a ConstraintOccurrence exists between them.

The 2 score rules to detect ascending and descending diagonal line violations are implemented in the same way.


The constraintsBroken rule counts all ConstraintOccurrence's and updates the scoreCalculator. Notice that actually the negative count is set as the score, because the solver tries to find the solution with the highest score.

Most planning problems tend to use negative constraints and negative scores.

Note that the ConstraintOccurrence's can be weighted so certain score rules can have a bigger impact on the score than others. Of course in that case, we would need to use a sum accumulate instead of a count accumulate.

Why work with score rules?

- Ease of development and maintenance
 - Focused on the constraint logic
 - => DRL instead of Java
 - Separation of constraints
 - Adding/removing a constraint does not affect other constraints
- Performance and scalability
 - Drools rule engine pattern matching
 - Faster than for/while loops and hashmaps
 - Delta based score calculation
 - Moving queen B won't re-evaluate if queen A and queen C are on the same row

www.javapolis.com

What are the advantages of using a rule engine to do score calculation?

If you need to find a String pattern in a String, you don't use basic java: you use a regular expression (in java). Why? Because it's faster, easier to define and you don't have to mess around with program loops.

If you need to find a constraint pattern in a Solution, don't use basic java, use a DRL (in java).

It's easier to define a constraint in DRL, because it focuses on the constraint logic and you don't have to worry about program loops.

Of course, there's a catch: you have to learn the DRL language, just like you had to learn regular expressions.

Each constraint has it's own score rule, which makes maintenance easier.

The drools rule engine is build for high performing and scalable pattern matching. Chances are slim to none that you will do better with custom program loops and hash maps.

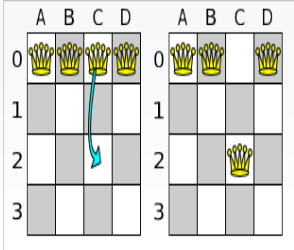
Besides fast pattern matching, it's also very important to have delta based score calculation. This means that when a part of the solution changes, it won't re-calculate the score of the unchanged parts. So if we move queen B, it won't need to check if queen A and C can still attack each other.

The drools rule engine defaults to forward chaining, so you get delta based score calculation and its huge performance boost for free. Implementing delta's manually is very difficult and very time-consuming.

In short, adding extra constraints is easy and scalable in drools-solver. And it's definitely worth learning DRL for this.

A move

- Changes one solution into another solution
 - YChangeMove: move a queen to a different row
 - Other move types possible
- Has an undo move
 - Does the exact opposite



www.javapolis.com

But how do we get to that solution with the highest score?

Well, we make a move on the current solution.

Shown is a move that moves queen C to row 2.

That's a domain-specific YchangeMove. Other moves types, for example switching the rows of 2 queens, could also be used.

Each move also has an undo move, which changes the new solution back to the original solution. In the shown case, that's the moving queen C back to row 0.

Notice that it's not possible to get to every possible solution with a single YChangeMove. We can't get to the solution with all queens on the last row in a single YChangeMove.

However if we combine multiple YChangeMoves, we can get to any possible solution.

When defining your move types, you should verify that they allow you to reach any possible solution.

Let's take a look at the YChangeMove implementation.

Move implementation

```
public class YChangeMove implements Move {


    private Queen queen;
    private int toY;

    public void doMove(WorkingMemory wm) {
        FactHandle queenHandle = wm.getFactHandle(queen);
        queen.setY(toY);
        wm.update(queenHandle, queen);
    }

    public boolean isMoveDoable(WorkingMemory wm) {
        return queen.getY() != toY;
    }

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

    // constructor, equals, hashCode
}
```

www.javapolis.com

A YChangeMove has a queen and a row to move the queen to.

The Move interface requires you to implement 3 methods: doMove, isMoveDoable and createUndoMove.

The doMove method basically just sets the y of the queen to it's new row. However, it also needs to tell the workingMemory – which is a drools rule engine thing - that it made a change to that queen, so the drools rule engine knows what rules might be dirty the next time it is asked to calculate the score.

The isMoveDoable method allows you to get a move ignored by the solver, because it would create a corrupted solution or it wouldn't change the solution. In this case, if the queen is moved to the row it's already located on, which means the solution won't change, the move is not doable.

The createUndoMove method creates a move which can change the new solution back to the original solution. In this case, it simply moves the queen back to its original row.

Note that a createUndoMove method is called before the move is done, so before the solution has been changed.

So where are these moves created?

Move generation

■ Doable move
■ Not doable move (no change)

■ MoveFactory

- Configured as selector in the configuration file
- CachedMoveListMoveFactory
 - createMoveList() method

```

for (Queen queen
    : nQueens.getQueenList()) {
    for (int n : nQueens.createNList()) {
        moveList.add(
            new YChangeMove(queen, n));
    }
}
            
```

www.javapolis.com

A MoveFactory creates the move instances.

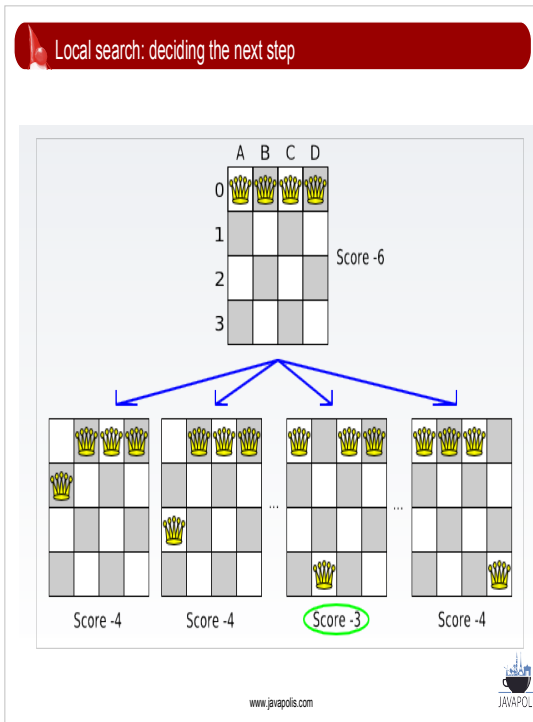
It is configured as part of the selector in the configuration file.

The selector selects moves from the move factory, but the current implementation is pretty basic, it just selects all moves. Expect some improvements on the selector in the near future, to make it more scalable.

The MoveFactory needs a domain-specific implementation, but if you extend the CachedMoveListMoveFactory, you only need to implement the createMoveList method.

Here it just creates a move for every queen for every row.

That will make 12 doable moves and 4 not doable moves. The not doable moves are automatically ignored by the solver.



Which move is picked to change the solution?

Well, the local search solver tries every move on a solution. It then picks the move which leads to the solution with the highest score. That move is called a step.

Here we start from the starting solution with all queens on row 0. It has a score of -6, because

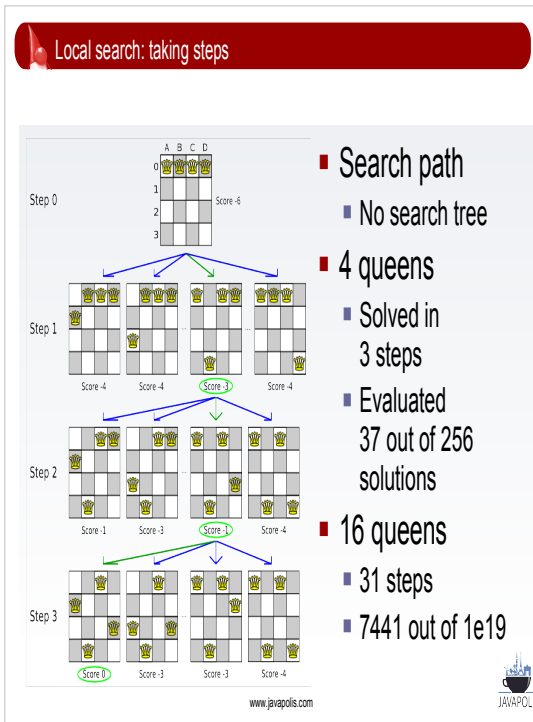
A can attack B, C and D,
B can attack C and D
and C can attack D.

If we move queen A to row 1 (as shown on the left), it can no longer attack C or D, so its score is only -4.

However, if we look at all moves, it turns out that moving queen B to row 3 has a highest score of -3, so the local search solver picks that move as the step.

This new solution only violates the constraints 3 times instead of 6 times, but how do we get to 0 violations?

Well, we just do it again, starting from that new solution.



If we do that a couple of times, we get this.
The green lines indicate the steps.

Notice it's a search path, not a search tree.

I've just explained the first step.

The second step moves queen D to row 2, which gives us a score of -1.

The final step moves queen A to row 1 and we get a solution with no constraints broken and a score of 0.


The 4 queens problem is solved after 3 steps.

In those 3 steps, we've evaluated 36 solutions, only a fraction of all 256 possible solutions.


For 16 queens this ratio is even better: only 7000 out of 10 to the power 19 possible solutions needed to be evaluated. That's pretty good, isn't it?

But what would happen if all the evaluated moves deteriorate the score and we haven't reached the optimal solution yet?

Would the local search solver get stuck? I am afraid so.

Local optima

- Simple local search = hill climbing
 - Gets stuck in local optima
 - Fixed by using an acceptor
- Tabu search acceptor
 - Solution tabu
 - Don't visit the same solution twice
 - Move tabu
 - Property tabu
- Simulated annealing acceptor
 - Randomly move around
 - Higher acceptance chance for improving moves

www.javapolis.com

Simple local search works a lot like if you were hill climbing to the top of the world: you always try to go up.

At some point you can end up on the top of the Mont Blanc. If you're not willing to go down at that point, you'll never climb the Mount Everest and you will never reach the highest point on earth.

So simple local search can get stuck on a hill top. Such a hill top is called a local optima.

Tabu search resolves that issue. Solution tabu remembers recently encountered solutions and doesn't accept a move if it leads to an already visited solution.

As you've seen earlier, the n queens configuration file uses a tabu acceptor. Notice that you don't need to implement anything yourself to use tabu search, just switch it on.

Drools solver also support move tabu and property tabu.

Simulated annealing has a very interesting way to avoid getting stuck. It doesn't try all moves at every step, but it just randomly picks the first move it likes as the next step. Improving moves have higher chance to be picked as the next step, especially if many steps have already been taken. Because simulated annealing doesn't check all moves at every step, it steps a lot faster.

This seemingly weird algorithm does perform very well and it is worth to test it on your planning problem.

Luckily, drools-solver has a benchmarking utility, which allows you to play out different algorithm configurations against each other, so you can use the best one in production.

 Finish solving

- Finish
 - Time spend finish
 - Step count finish
 - Score reached finish
 - Unimproved step count finish
 - Composition finish
 - Score reached and unimproved step count finish
- Best solution
 - Best encountered solution
 - Can be the optimal solution

www.javapolis.com

Local search doesn't know when it finds the optimal solution. For real world use cases, you won't have time to wait years to find that optimal solution anyway. So at some point the solver will need to stop solving, by using a finish.

There are a couple of build-in finishes, which allow you to finish for example after 5 minutes,
when a 1000 steps have been taken
when a score of 0 is reached,
or when the score hasn't improved in the last 10 steps.


You can even combine finishes, for example
when a certain score is reached and the score hasn't improved recently.
This avoids outputting a solution, on which a user can make an obvious improvement.

Once the solve method returns, you can get the best solution from the solver. It's the solution with the highest score encountered. This isn't necessary the optimal solution, because it might not have been encountered yet. However, the best solution should be a lot better than what human planners can do in the same amount of time. Although a human planner is still very usefull to help you define and tweak score rules and move granularity.

Summary

- No official releases yet
- Download drools solver
 - Part of drools trunk in subversion
 - <http://labs.jboss.com/drools/subversion.html>
 - Build and run with maven 2
- Read the solver manual
 - Linked from blog
 - http://users.telenet.be/geoffrey/tmp/solver/manual/html_single/
- Read the drools team blog
 - <http://blog.athico.com> (label solver)

www.javapolis.com

JAVAPOLIS

Do you want to get started with drools solver?

Well, there are no releases yet, but you can easily check it out and build it yourself.

The source code is part of the drools trunk in subversion.

It builds with maven 2.

And even the GUI examples can be run directly from the command line with maven 2.

So that's a total of 3 command line instructions to get started, which are documented in the solver manual.

The solver manual isn't included in the main drools manual yet, but it already covers all the basics and some advanced stuff. It is published on my website and on the blog.

If you're interested in this technology, I can recommend you to read our drools team blog. I 'll post a link to this presentation on the blog too by the way.

Feed-back is welcome on the blog or on the drools mailing lists or even directly after this presentation.

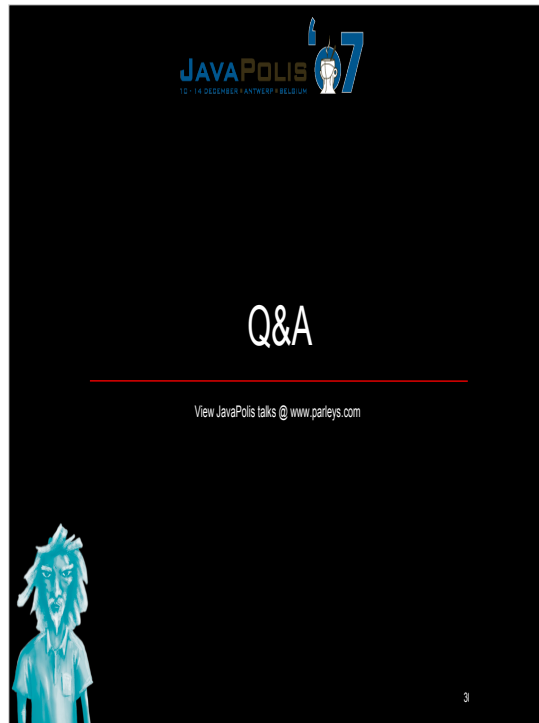
Concluding statement

Solve your planning problems with
Drools Solver.

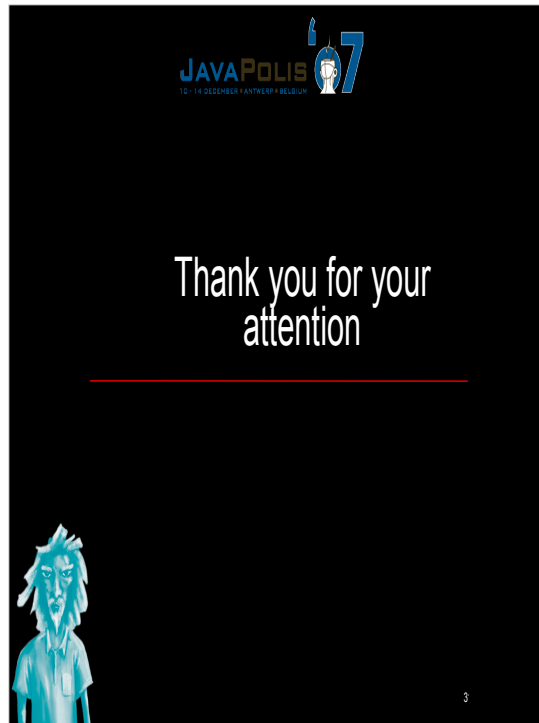
www.javapols.com

So, as a conclusion:

Solve your planning problems with Drools Solver.



Any questions?



Thank for very much for your attention. Have a nice evening.