SOLVING PLANNING PROBLEMS WITH DROOLS PLANNER A TUTORIAL

D.V.I. WEPPENAAR AND H.J. VERMAAK

Abstract

Planning problems are frequently encountered in everyday situations. The brute force approach of evaluating every possible solution for any medium size planning problem is just not feasible. Drools Planner is an open source Java library developed to help solve planning problems by using meta-heuristic algorithms. Drools Planner uses the Drools Expert (rule engine) for score calculation to greatly reduce the complexity and effort required to write scalable constraints in a declarative manner. This paper presents an introduction to Drools Planner, how it can be used to solve problems and concludes with an example scenario.

Keywords: automated problem solving, meta-heuristic algorithms, drools planner

1. INTRODUCTION

Solving planning problems can be a difficult and time-consuming process. Algorithms designed for this very purpose have been a topic of interest in recent years. This is confirmed by organised events, such as the 2007 International Timetabling Competition [1]. One participant in this competition, de Smet [2], employed an early version of Drools Planner [3] (known as Solver then). Although the solving of planning problems is greatly simplified by automated software, such as Drools Planner, there are still some matters to consider.

In this work we present a tutorial that discusses some of the main concepts of Drools Planner in the interest of solving planning problems frequently encountered in research and other fields. The features of Drools Planner and the main configuration settings are discussed. An example scenario (first presented in [4]) aims to demonstrate the capabilities and usage of Drools Planner.

The rest of the paper is organised as follows: Section 2 discusses a simple planning problem and how drools can be applied to it. Section 3 provides an overview of Drools Expert; the rule engine behind Drools Planner. In section 4 we discuss Drools Planner and related concepts in more detail. Section 5 presents the example problem from the work conducted in [4] and how the strengths of Drools Planner were leveraged to solve this particular planning problem. Section 6 presents some concluding remarks.

2. PLANNING PROBLEMS

Drools Planner [3] (formerly known as Drools Solver) optimizes automated planning. This optimisation is made possible by supporting several search algorithms (such as taboo search or simulated annealing) and the ability to easily switch between these algorithms. Figure (adapted from [5]) represents a simple, two-dimensional bin packaging example where five items of varying dimensions are placed into a four-by-eight container.

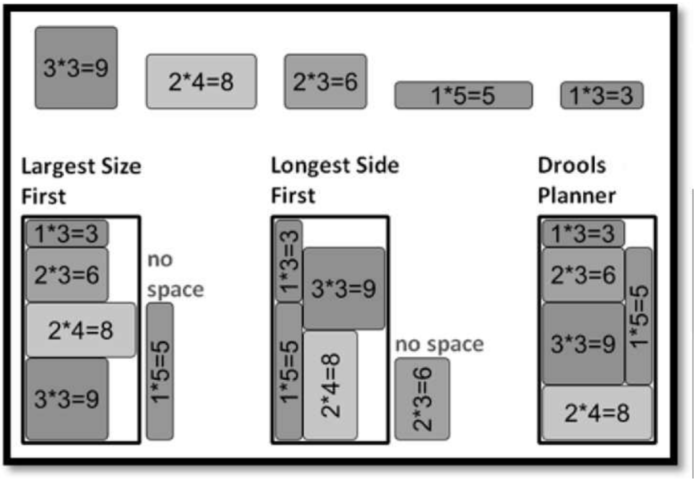

Figure 1: Two-dimensional bin packaging example

Three solutions are presented, of which only the last solution is feasible:

- The first algorithm sorts the items according to their respective areas and then adds them to the container in descending (largest to smallest) order. It starts with the item with an area of nine. Unfortunately the one-by-five (area of five) item does not fit into the container.
- The second algorithm sorts the items according to their longest side length. The items are then loaded into the container in order of longest-to-shortest side length. The item with a side length of five is added first, but as the container fills up with items it is soon evident that there is not sufficient space for the two-by-three (area of six) item.
- The last algorithm, which is implemented by Drools Planner, manages to fit the items into the designated container and subsequently generating a feasible solution.

The question arises: Where is the best position to place a specific item in the container if it does not influence the feasibility of the solution? Figure 2 (adapted from [5]) shows three more problem instances as well as a feasible solution for each of them.

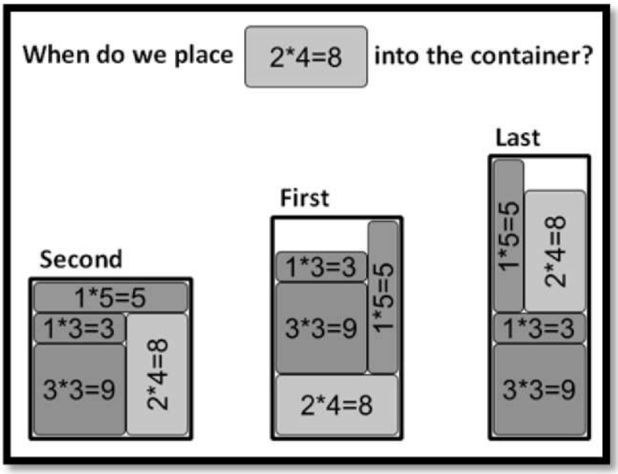

Figure 2: The bin packaging example is NP complete

The two-by-four item is located in a different location in each of the instances. The location depends upon the size of the container, the size of the other items involved and a set of constraints. This phenomenon is commonly known as NP complete:

- Where it is easy to verify a feasible solution,
- but hard to find a feasible solution.

Only by evaluating many (or all) of the solutions, a feasible solution can be identified. In fact, it is difficult to prove that a feasible solution even exists. This difficulty is illustrated by Figure 3 (adapted from [5]).

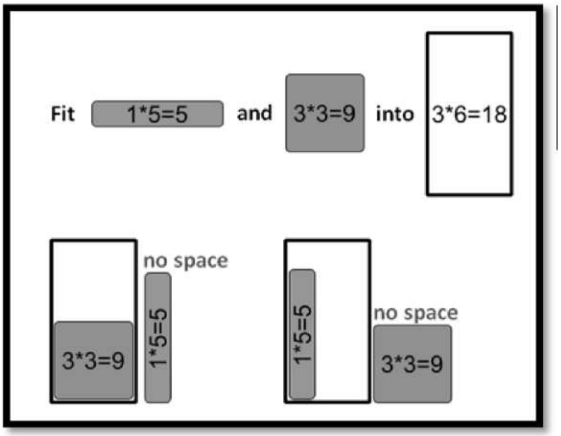

Figure 3: Difficulty with NP complete problems

Considering that the total size of the items is equal to 14 and the container has a size of 18, it is simple for human beings to determine that no feasible solution exists for this trivial planning problem. Real world bin packaging problems, however, can have thousands of items, multiple containers, varying container dimensions and several more constraints.

The brute force approach of trying every possible solution is just not feasible. Instead, Drools Planner optimizes the search function with meta-heuristic algorithms, such as taboo search and simulated annealing.

3. DROOLS EXPERT

A Production Rule System is Turing complete (has a computational power equivalent to a universal Turing machine) with a focus on knowledge representation to express propositional and first order logic in a concise, non ambiguous and declarative manner. The brain of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules, also called Productions or just Rules, to infer conclusions which result in actions. A Production Rule is a two-part structure using First-Order Logic for knowledge representation [6].

```
when
        <conditions>
then
        <actions>
```

The process of matching the new or existing facts against Production Rules is called Pattern Matching, which is performed by the Inference Engine. Drools implements and extends the Rete algorithm. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimised implementation of the Rete algorithm for Object Oriented systems.

A Production Rule System's Inference Engine is state-full and able to enforce truthfulness called Truth Maintenance [6]. A logical relationship can be declared by actions which mean the action's state depends on the inference remaining true; when it is no longer true the logical dependent action is undone. The "Honest Politician" is an example of Truth Maintenance, which always ensures that hope can only exist for a democracy while there are honest politicians.

```
when
        an honest Politician exists
then
        logically assert Hope

when
        Hope exists
then
        print "Hurrah!!! Democracy Lives"

when
        Hope does not exist
then
        print "Democracy is Doomed"
```

The Truth Maintenance employed by Drools Expert is an important aspect as it ensures that assertions made based on facts that are no longer true are automatically removed. This saves execution time by not evaluating cases that have become untrue as well as the associated memory of the deleted assertions.

4. DROOLS PLANNER

Drools Planner does automated planning by combining a search algorithm with the power of the Drools rule engine (Expert). Good examples of such planning problems include calculating employee shift rosters, routing (vehicle, freight, people, etc.), educational or organisational scheduling, bin packaging (stock sorting and storage organising), machine queue planning and even Miss Manners.

A planning problem consists out of a number of constraints. Generally, there are three types of constraints [7]:

- A negative hard constraint must not be broken.
- A negative soft constraint should not be broken if it can be avoided.
- A positive constraint (or reward) should be fulfilled if possible.

These constraints define the score function of a planning problem. This is where the benefits of applying the Drools rule engine becomes apparent; adding constraints with score rules is easy and scalable.

A planning problem usually has a number of solutions, each with a score that can be calculated. There are three categories of solutions [7]:

- A possible solution is a solution that does or does not break any number of constraints. Planning problems tend to have an incredibly large number of possible solutions. Most of these solutions can be discarded as 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, but every feasible solution is a possible solution.
- An optimal solution is a solution with the highest score. Planning problems tend to have one or a few optimal solutions. There is always at least one optimal solution, even in the remote case that the solution is not a feasible solution because there are no feasible solutions.

"Drools Planner supports several search algorithms to efficiently wade through the incredibly large number of possible solutions [7]." It makes it easy to switch the search algorithm, by simply changing the solver configuration.

4.1 Solving a Problem

Every built-in solver implements the Solver interface. The built-in solvers should suffice for most situations and it is thus not usually necessary to implement the Solver interface. Solving a planning problem with Drools Planner generally consists of four steps [7]:

- Build a solver, specifying the particulars of the solver in the configuration.
- Set a starting solution on the solver, with the help of an implemented Starting Solution Initialiser.
- Issue the solve command and wait for the solver to terminate.
- Get the best solution found by the solver.

A Solver should only be accessed from a single thread, except for the methods that are specifically thread-safe. A Solver instance can be built with the XML solver configurer and then configured with a solver configuration XML (extensible markup language) file. Drools Planner makes it relatively easy to switch a solver type by making alterations to the configuration. The benchmark utility allows one to play out different configurations against each other and report the most appropriate configuration for the problem at hand.

4.2 Constraints

In Drools the rules or constraints are specified in a special file or set of files. These rule files have the DRL extension. The rules determine how a solutions score is calculated. This is how the rule engine determines whether a new solution is an improvement on the previous one or not. Rules come in two varieties, hard- and soft constraints.

Hard constraints cannot be broken and if they are, the solution is not considered valid or feasible. Soft constraints can be broken and a solution that has a soft constraint or several soft constraints broken is still considered feasible. It is most likely not the best possible solution, but a solution nonetheless. It is possible to have score calculation based on soft or hard constraints separately, or as a combined score. This configuration is determined by the XML configuration file.

4.3 Score Calculation

It is recommended that Drools be used in the forward-chaining mode (which is the default behaviour), and for score implementations this will enable delta based score calculation instead of a full score calculation on each evaluation of a solution. This means that only constraints that have changes in one or more of their conditions since the previous evaluation will be recalculated. The performance gain is considerable and increases exponentially relative to the size of the planning problem. This performance gain is automatic and simplifies implementations considerably without the need to write a complex delta-based score calculation algorithm. The rule engine performs most of the heavy lifting.

4.4 Solution Initialiser

The idea behind the solution initialiser is to create a simple algorithm to come up with an acceptable (not perfect) starting solution for the solver to work with. For large problems, a simple filler algorithm that merely places objects to be scheduled in arbitrary positions does not suffice. A (local search) solver starting from a bad starting solution wastes a great deal of time to reach a solution which an initialiser algorithm can generate in a fraction of that time. An initialiser algorithm usually works as follows [7]:

- It sorts the unplanned elements in a queue according to some general rules, for example by examination student size.
- Next, it plans them in the order they come from the queue. Each element is placed into the best, still available spot.
- It does not change an already planned element. It exits when the queue is empty and all elements are planned.

Such an algorithm is very deterministic: it is really fast, but it cannot be given more time to generate a better solution. In some cases, the solution it generates will be feasible, but in most cases it will not. The algorithm only lays the foundation for a real solver to get to a feasible or more optimal solution. Nevertheless, such an initialiser gives the real solver a serious head start.

This is done by implementing the starting solution initialiser interface. A (uninitialised) solution is set on the solver. Once the solver starts, it will first call the starting solution initialiser to initialise the solution.

If the starting solution initialiser adds, edits or removes facts, it needs to notify the working memory about this. It can use score calculation during its initialisation process.

4.5 Selecting a Next Step

A move is the change from a solution A to a solution B. A move can have a small or large impact. Each of the move types will be an implementation of the Move interface. Note that one can alter multiple facts in a single move and effectively create a big move (also known as a coarse-grained move). Drools Planner automatically filters out moves that cannot be done. A move that cannot be done is a move that changes nothing on the current solution or a move that is impossible to do on the current solution. A move that can currently not be done can become possible in a later solution.

Each move has an undo move: a move (usually of the same type) which does the exact opposite. An undo move can be created from a move, but only before the move has been done on the current solution. The local search solver can do and undo a move more than once, even on different (successive) solutions. Two moves which make the same change on a solution must be equal.

At each solution, local search will try all possible moves and pick the best move to change to the next solution. It is up to the programmer to generate those moves. Not all moves can be done. The number of possible solutions is much more than the number of moves that can be done. It is important not to create moves to every possible solution. Instead use moves that can be sequentially combined to reach every possible solution.

It is recommended to verify that all solutions are connected, in some manner, by your move-set. This means that by combining a finite number of moves one can reach any solution, from any solution. If this is not the case, possible solutions are already being excluded from the start. This is usually caused by limiting the generation to only big moves. Big moves may outperform small moves in a short test run, but it may be an entirely different scenario as the test run size is increased. It is recommended to mix different move types and it is generally a good idea to prefer small (fine-grained) moves to big (course-grained) moves, because the score delta calculation performance benefit will be more prominent. Sometimes it is possible to remove a hard constraint by using a certain set of big moves, thus gaining performance and scalability.

The winning move is a step. The local search solver tries every move on the current solution and picks the best accepted move as the step. The move with the highest score is picked as the next step. If multiple moves have the same highest score, one is picked randomly. The step is made and from that new solution, the local search solver tries all the possible moves again, to decide the next step. This occurs continually in a loop.

A simple local search always takes improving moves. This may seem adequate, but this is not the case. There are 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. Consequently, however, it is very real that it will pick the step back to solution X with score -1. It will then start iterating indefinitely between solution X and Y.
- It can start walking in its own footsteps, picking the same next step at every step.

Drools Planner implements superior local searches, such as taboo search and simulated annealing which can avoid these problems. It is recommended to never use a simple local search, unless the possibility of local optima is entirely eliminated in the specific planning problem. The local search solver decides the next step with the aid of three configurable components:

- A selector selects (or generates) the possible moves of the current solution.
- An acceptor filters out unacceptable moves. It can also weigh a move it accepts.
- A forager gathers all accepted moves and picks the next step from them.

4.6 Benchmarking

Drools Planner supports several solver types and it can be troublesome to determine the best solver for a particular solution. Although some solver types generally perform better than others, it really depends on the domain of the problem. Several solver types also have settings that can be configured. The settings can influence the results of a solver significantly, although most settings should perform surprisingly well out-of-the-box. Drools Planner includes a benchmarker to alleviate the difficulty of choosing a solver and settings. This allows one to play out different scenarios against each other and pick the best configuration for the problem domain at hand.

Every solver benchmark entity contains a solver configuration (for example, a local search solver) and one or more unsolved solution file entities. It will run the solver configuration on each of those unsolved solution files. A name is optional and generated automatically if omitted. The common sections of multiple solver benchmark entities can be extracted via inheritance and then be overridden for each solver benchmark entity. The benchmarker supports outputting the best score over time statistic as a graph or CSV (comma separated values) file. It will output an overview of all graphs and CSV files as an index.html file in the specified statistic directory.

5. EXAMPLE PROBLEM

This section will briefly discuss the example problem. Usually a device will have a maintenance guide in the form of hours worked or operations performed. This is not accurate in all situations as the same device, be it a robotic arm for example, can be used in different configurations. The work this device will perform will vary from configuration to configuration.

The idea is to determine how strenuously a device has been operated for a set period of time. This information is then used to augment the maintenance interval provided by the manufacturer as a guide. The devices are prioritised based on certain criteria as well as a maintenance threshold that is calculated based at what levels the device was operated, as well as the period at each load level. These thresholds and priorities are then used to schedule the devices for maintenance.

5.1 The Process

Data indicating the prognostic state was collected for a device, in this case a DC (Direct Current) induction motor. This data was processed and formatted as required, after which it was stored in the database. The scheduling component then retrieved the device information from a database, along with the maintenance schedule template. This template specified which days and time slots were available for maintenance to be scheduled.

The scheduler component then proceeded to iteratively generate the schedule using the schedule template. This process continued until the set criteria were reached. This criterion can be a time limit or some measurement (possibly constraints) indicating that an acceptable schedule has been reached, or a combination of both. After the scheduler concluded, the generated schedules could then once again be stored in the database to be used as required.

5.2 Scheduling Criterion

5.2.1 Motor Priority

The data collected from a DC induction motor was used to simulate 50 motors [4]. These motors were simulated at different load levels and different priorities. Priority in this scenario is defined as the product of three parameters: Need Urgency, Customer Rank and Equipment Criticality (NUCREC) [8]. This is an improved method over the standard First-In-First-Out (FIFO) or first-come-first-served approaches more commonly used. NUCREC improves on the Ranking Index for Maintenance Expenditures (RIME) in several ways. A rating system of numbers 1 to 4 is recommended. Since most human beings think of number 1 as the first priority to get done, the NUCREC system does number 1 first.

The product of the ratings gives the total priority. That number will range from 1 (which is 1 x 1 x 1) to 64 (4 x 4 x 4). The lowest number signifies the first priority. A '1' priority is a first-class emergency. When several work requests have the same priority, labour and materials availability, locations, and scheduling, fit may guide which is to be done first. “With these predetermined evaluations, it is easy to establish the priority for a work order either manually by taking the numbers from the equipment card and the customer list and multiplying them by the urgency or by having the computer do so automatically [8].” Naturally, there may be a few situations in which the planner's judgment should override and establish a different number, usually a lower number so that the work gets done faster.

5.2.2 Maintenance Threshold

In addition to priority, another metric is also used to determine the scheduling order and frequency of motors. This metric is referred to as a maintenance threshold. The maintenance threshold indicates how far the device has progressed in nearing its specific maintenance 'due date'. It is important to note at this juncture that the determination of this threshold has been simplified for the implementation of the example. It is advised for real-world scenarios and implementations that the designer employs a method (or combination of methods) such as those discussed in [4].

For this simulation the maintenance threshold was determined by taking the time units a motor was operational and multiplying these by the normal operational value, which in this case was load level three. This gives a base value to be used in a calculation to determine the percentage of time that has elapsed towards the full maintenance term. For a more detailed description please consult [4].

5.2.3 Maintenance Personnel

Devices that require maintenance constitutes only one part of the maintenance process. Human technicians are required to perform the physical tasks. The human element requires that certain ethical requirements be taken into consideration. One such requirement is the legal regulations associated with work hours. The department of labour usually determines what is fair when it comes to working hours and overtime. This is to protect the employees and companies involved by clearly stating the legal requirements. For this simulation the regulations as specified by the South African Department of Labour (DoL) [9] were used as a guideline.

Maintenance personnel are required to perform maintenance for a set number of hours per day and allowed a certain number of overtime hours per day and per week according to the regulations of the DoL. Four maintenance technicians were available to be scheduled for maintenance tasks in this example.

5.2.4 Schedule

A maintenance schedule depends on various elements such as the production schedule and maintenance policy of the company, available maintenance personnel, and the physical state of the devices, to name a few. This section introduces the schedule used for this simulation [4]. It is first assumed that no maintenance or production takes place over weekends (Saturday and Sunday). It is likely that most real assembly systems will operate everyday for the maximum number of hours per day.

A simplified schedule is used in this simulation as it serves only to test some concepts. The schedule is the same for all working days (Monday to Friday) of the week. Production occurs for a period of 14 hours a day, from 6am until 8pm. During this time no maintenance is scheduled which will disrupt production. Unavoidable, emergency maintenance, such as a catastrophic failure, is not considered in this simulation. Normal maintenance can be scheduled from 2am until 6am and from 8pm until 10pm, respectively. It is assumed that the maintenance technicians work in shifts around the production. Any maintenance scheduled between 10pm and 2am of the following day is considered to be overtime.

5.3 Constraints and Initialisers

5.3.1 Hard Constraints

Constraints in this section were not allowed to be broken and any solution that contains broken hard constraints is not considered a valid solution. The hard constraints of the simulation are as follows [4]:

- Maintenance Tasks In Same Timeslot - This rule ensured that no maintenance task is scheduled in the same maintenance slot (day and time slot). For each maintenance task scheduled on the same slot as another, a hard score reduction of -1 is added.
- Schedule During Production - This rule was entrusted to ensure that no maintenance is scheduled during periods of production. For each maintenance task scheduled in a slot marked as production a hard score reduction of -1 is added.
- Overtime Per Day - No employee may be scheduled for more than three (3) hours of overtime per day (24 hour period), according to the South African labour guidelines for general employment [9]. The number subtracted from the total hard score is equal to the number of overtime tasks exceeding the 3/day limit.
- Overtime Per Week No employee may be scheduled for more than 12 hours of overtime per week (7-day period) according to the South African labour guidelines for general employment [9]. The number subtracted from the total hard score is equal to the number of overtime tasks exceeding the 12/week limit.

5.3.2 Soft Constraints

The following constraints do not invalidate a solution, they aid in ranking feasible solutions in an effort to find the 'best' possible solution. This is important since finding the perfect solution is in all probability not possible. These are the soft constraints used in the simulation [4]:

- Motor Priority Each motor has a priority that is calculated based on what was discussed in 5.2.1. Values closer to one (1) signify highest motor priority. Motors with high priorities should be scheduled first.
- Maintenance Threshold Each motor has a maintenance threshold that depends on the load level at which the motor was operated, as discussed in

5.2.2. Motors that are closer to qualifying for maintenance should be scheduled before motors that are not. The value of the soft score is determined by the difference between the threshold of the current motor and the motor with a more advanced threshold, scheduled after the former.

It is important to note that these two constraints are not necessarily aligned and often in competition with one another. This is the reason why this solution will never have a 0 soft score, which would indicate a perfect solution. It is rather a question of finding the best possible solution for the supplied parameters.

5.3.3 Solution Initialisers

It is the solution initialiser's responsibility to create starting solutions from which the solver can commence solving. Without an acceptable starting solution the solver would waste considerable amounts of time just to arrive at the said starting solution. In this simulation two starting solution initialisers were evaluated [4]:

- Default Initialiser - Simulates a poor starting solution by generating a starting solution that does not take into account any of the rules. The motors are assigned to the maintenance slots in the order in which they are retrieved from the database (motor 1 to motor 50).
- Structured Initialiser - The list of motors retrieved from the database is arranged according to their individual priority. If the priorities of two (or more) motors are the same, the maintenance threshold becomes the deciding factor in the scheduling order.

For this simulation only the motor scheduling differs between the two initialiser implementations. Once the motor list has been retrieved (either sorted or unsorted), the motors are assigned to the maintenance slots from the highest priority to the lowest. The rest of the parameters are populated in the same manner. The four maintenance technicians are assigned to maintenance slots in a round robin fashion until all maintenance slots have a technician assigned to them.

5.4 Solver Configuration

5.4.1 Selector

In the current version of Drools Planner (5.0) the selector generates a list of moves by making use of Move Factories. It was decided to use relative selection with a taboo search accepter [7]. The relative selection takes a subset of all the possible moves which in our case was 2% (0.02). Different relative selection values were experimented with such that the number of moves that were returned allowed for two moves to be performed per second. The number of possible moves determines the duration it takes the planner to select a next move. There exists a trade-off relationship between the two values and some time should be taken to reach an acceptable balance.

5.4.2 Acceptor

An accepter filters out unacceptable moves. It can also weigh a move it accepts. An accepter is used, in conjunction with a forager, in order to facilitate active taboo search, simulated annealing, great deluge, etc. For each move it generates an accept chance. A move can be rejected based on a score that determines acceptability. One can implement one's own accepter, although the built-in accepters should suffice for most needs. One can also combine multiple accepters. The current version of Drools Planner provides two built-in accepter types [7]:

- Taboo search accepter - When the taboo acceptor takes a step, this step is declared to be taboo. By declaring recently visited steps to be taboo (not allowed), the solver can avoid getting stuck in local-optima. Drools-solver implements various taboo types. Solution taboo makes recently visited solutions taboo. Move taboo makes recent steps taboo. Undo move taboo makes the undo move of recent steps taboo. Property taboo makes a property of recent steps taboo. One can combine taboo types, but take care not to specify the taboo size is too small, this could cause the solver to still get stuck in a local optimum. With the exception of solution taboo, if one picks too large a taboo size, your solver can get stuck by bouncing off the walls. Use the benchmarker to fine-tweak your configuration.
- Simulated annealing accepter - Initially, simulated annealing (unlike taboo search) does not pick the move with the highest score, neither does it evaluate all moves. It gives un-improving moves a chance, depending on the score and 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.

For this simulation a complete property taboo size of five (5) and a complete solution taboo size of 1500 were selected after trying a few different configurations.

5.4.3 Forager

The forager gathers all accepted moves and picks the move which is taken as the next step. A forager can reduce the subset of all selected moves to be evaluated, by halting early if a suitable move has been accepted. One can implement one's own custom Forager, but for most situations the built-in foragers should suffice. Drools Planner provides several predefined forages [7]:

- 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 the accept chance metric.
- 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.
- 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.
- First randomly accepted forager - Generates a random number for each accepted move and if the accept-chance of a specific move is higher, that move is selected as the next move.

A maximum score of all (MAX_SCORE_OF_ALL) forager was selected in conjunction with taboo accepter as suggested by the documentation [7].

5.5 Moves

This section examines the possible moves the solver could take in the pursuit of a better solution. Each of these moves had its own Move Factory associated with it which fulfils the function of generating all the possible Moves from the current solution. These moves are then evaluated and weighted and a “winning move” is then selected out of the subset determined by relative selection. The following moves are applicable to the simulation scenario:

- Maintenance Slot Change - This move swaps out the maintenance slot (time slot and day) of the maintenance task. This is the equivalent of swapping one task with another.
- Maintenance Technician Change - Swaps the maintenance technician associated with maintenance task with the technician of another task.
- Motor Change - Swaps the motor assigned to one task with the motor assigned to another task.

It is important to note that each move only really alters one parameter. The effect that the move in question has on the score of the solution determines whether the move is included in the set of acceptable moves. Which move the solver eventually selects is determined by several factors including the weight of the move compared to others.

5.6 Results

5.6.1 Solution Initialiser Comparison

This section examines how the score of the solution and solver performance is affected by using a different solution initialiser, as discussed in section 5.3.3. The starting scores were -0hard/-1280soft and -0hard/-921soft for the default- and structured initialisers, respectively. The benchmark results are given in Table 1.

Table 1: Solution initialiser benchmarks

Run#	Minutes	# of Steps	Default Initialiser	Structured Initialiser
1	0.5	122	-1198	-914
2	1	243	-1169	-914
3	2	486	-1165	-882
4	5	1214	-1043	-882
5	15	3641	-963	-826
6	30	7281	-916	-792
7	60	14562	-857	-792
8	120	29124	-772	-757
9	240	58248	-733	-727
10	480	116496	-733	-727

Note that the initial difference between the initialisers is quite sizable. This directly translates into less processing required by the solver and highlights the importance of a good, well-balanced starting solution initialiser. As the length of processing increases, the difference between the solution initialisers becomes less prominent. This is the reason why the solution initialiser, although important initially, is not responsible for solving the scheduling problem and thus the actual solution would benefit little from spending a great duration in processing the starting solution initialiser.

The hard score of the solutions remains unchanged between iterations of the solver for the entire process. The reason for this is that both solution initialisers create a starting solution which breaks no hard constraints. Even if the initial solution has a hard score of 0, the constraints ensure that none of the moves that are accepted diminish the quality of the solution.

No matter how great the improvement of the soft score is, the move in question will be ignored if the hard score is worsened.

Even if the score is not “perfect”, the final solution is still an improvement over that which was initially provided to the solver. In this scenario, the solver would benefit little by increasing the allowed processing duration. This was verified for both initialisers by giving them each 24 hours to process. The resulting score was exactly the same. The criterion that governs the solver is important in order not to waste unnecessary resources, and performance benchmarks give a good indication of what is sufficient.

5.6.2 Solver Solutions

Both solutions that were created are feasible in the sense that there was no hard score penalty. The resulting schedules are almost of equal quality, as these solutions represent the best possible scores, given the scenario and solver parameters. The default initialiser solution showed a significant improvement of 547 (from -1280soft to -733soft) or 42.7% and the structured initialiser solution had a final improvement of 194 (from -921soft to -727soft) or 21.1%. Do not be misled by the seemingly more prominent improvement of the initial solution of the default initialiser, as this was a solution far from ideal compared to the one initialised by the structured initialiser. In other words, there was greater room for improvement between the former and the latter.

Tweaking the solver parameters might show an improvement in the final score of the solutions, but it is far more likely that improvements of the rules will make a bigger difference. The ultimate solver is a balance between the rules that govern the score calculation and the parameters that control the inner workings of the solver. That aside, if one plainly ignores other aspects and evaluates the solutions on their final scores, the solution resulting from the structured initialiser is a better solution. However marginal the score difference, a better score is still superior.

As this scenario represents a situation where the maintenance schedule is to be generated, it is likely that the solver would be set to halt execution only if no better solution is found after a specified number of steps and/or period of time. If this is a lengthy process it could be scheduled to run overnight whenever a new schedule is required.

6. CONCLUSION

This work presents an overview of Drools Planner and how it can be used to solve planning problems. The main features and configuration parameters were discussed to provide a basis for solving problems with Drools Planner.

An example using Drools Planner was also discussed. First, the starting solutions of the different solution initialisers were presented as a reference. The benchmarks of the two solutions were examined and the benefit of a solution initialiser was discussed. It was shown that device prognosis, irrespective of where the data originated, can be used as a measure to determine how a device is scheduled for maintenance. When implementing the system as a rule engine, the maintenance scheduling can be structured to adhere to the requirements of the company/factory as long as the constraints are formulated as rules.

It was shown that both the solution resulting from the default and structured initialisers were of similar quality, given the comparable final soft score. As the complexity of the problem increases so will that of the solution initialiser. Even a relatively simple solving problem, like the one presented in the second part of this work still exhibit a noticeable performance advantage of using a well-structured solution initialiser as opposed to not initialising the solution before invoking the solver.

7. ACKNOWLEDGEMENTS

This work forms part of the research conducted by the Research Group in Evolvable Manufacturing Systems (RGEMS) [10] at the Central University of Technology (CUT), Free State [11]. This research was funded by the CUT and the Advanced Manufacturing Technology Strategy (AMTS) [12] group under project number AMTS-07-20-P. Additional funding was provided by the National Research Foundation's (NRF) [13] Masters grant program.

8. REFERENCES

International Timetabling Competition, http://www.cs.qub.ac.uk/itc2007-/index.htm, last accessed in April 2010.

International Timetabling Competition - The Finalists, http://www.cs.qub.ac.uk/-itc2007/winner/finalorder.htm, last accessed in April 2010.

Drools Planner - JBoss Community, http://labs.jboss.com/drools/drools-planner.html, last accessed in September 2010.

Weppenaar, D., “Intelligent Maintenance Management in a Reconfigurable Manufacturing Environment Using Multi-Agent Systems,” M.Tech. Thesis, Electrical and Computer Systems Engineering, Central University of Technology, Bloemfontein, Free State, South Africa, 2010.

De Smet, G. (Feb. 22, 2010). “Automated Planning Problems Are Cursed With NP Completeness,” Available: http://java.dzone.com/articles/automated-planning-problems, last accessed in September 2010.

JBoss Rules Reference Manual, Red Hat Documentation Group, Raleigh, NC, United States of America, 2008.

Drools Solver (Planner) - Community Documentation, Red Hat Documentation Group, Raleigh, North Carolina, United States of America, 2009.

Mobley, R. Keith, *Maintenance Fundamentals*, 2nd ed., Woburn: Butterworth-Heinemann, 1999.

South African Department of Labour, http://www.labour.gov.za/, last accessed in April 2010.

Research Group in Evolvable Manufacturing Systems, http://www.rgems.co.za/, last accessed in August 2009.

Central University of Technology, Free State, http://www.cut.ac.za/, last accessed in July 2009.

Advanced Manufacturing Technology Strategy, http://www.amts.co.za/, last accessed in July 2009.

National Research Foundation, http://www.nrf.ac.za/, last accessed in September 2010.