Decision Management Community Challenge Dec 2017- Santa's Reindeer
A solution using Drools

(Bob Moore, JETset Business Consulting, 15 Dec 2017)

1 Problem Statement (cut and pasted from the web site)

Santa always leaves plans for his elves to determine the order in which the reindeer will pull his sleigh. This year, for the European leg of his journey, his elves are working to the following schedule, which will form a single line of nine reindeer. Here are the rules:

1. Comet behind Rudolph, Prancer and Cupid
2. Blitzen behind Cupid
3. Blitzen in front of Donder, Vixen and Dancer
4. Cupid in front of Comet, Blitzen and Vixen
5. Donder behind Vixen, Dasher and Prancer
6. Rudolph behind Prancer
7. Rudolph in front of Donder, Dancer and Dasher
8. Vixen in front of Dancer and Comet
9. Dancer behind Donder, Rudolph and Blitzen
10. Prancer in front of Cupid, Donder and Blitzen
11. Dasher behind Prancer
12. Dasher in front of Vixen, Dancer and Blitzen
13. Donder behind Comet and Cupid
14. Cupid in front of Rudolph and Dancer
15. Vixen behind Rudolph, Prancer and Dasher

2 Revisiting the Problem

I already produced a solution for this in Python, but it didn't seem right to just solve it using some code in a programming language, however clever the language might be, so I wanted to take a stab at it with a decision tool of some sort.

As before I didn't want a solution to work out only the order of the reindeer for Christmas 2017. Jacob's[1] solution assures us this could be done by a seven-year-old using bits of paper with the reindeer's names on – and if I have a solution I need to rewrite every year to take account of Santa having new plans each year – the seven-year-old is not only a much cheaper option, but will probably come up with the solution faster.

So, I was looking for a solution that treats Santa's 'rules', not as part of the decision system itself, but as input data, so I can use the same system year after year. Ideally the solution should not contain information about specific reindeer either (so the names like 'Prancer' and 'Rudolph' are part of the input too), nor about how many reindeer there are, so in different years we can use different reindeer as well as different orders. In passing it should be able to detect if Santa has it wrong – and there is no order which satisfies his instructions, and ideally also if he's not been specific – where more than one order will satisfy them.

[1] See https://openrules.wordpress.com/2017/12/09/openrules-solution-for-the-christmas-challenge

As before I couldn't see how I could make use of decision tables. Three of the solutions to the challenge have used this approach, but to do so they all needed to explicitly mention the rules and the reindeer in the cells and/or the column headings. I couldn't (and still can't) see how to avoid this. If I don't have the reindeer names and the rules when I'm designing my decision table, what do I use for the rules/rows? For that matter what do I use for attributes/columns?

Damir[2] offers a solution in SQL. While this does hardcode the rules and the reindeer names in the query, one could envisage wrapping it up with some procedural logic. We could read in Santa's rules and then dynamically generate the query conditions from the rules before actually sending the query to the database. This approach would give us a solution we could reuse from year to year, but it is based on brute force and to a lesser extent the assumption there are always nine reindeer. With the increasing world population, maybe Santa will need his sleigh to fly faster to get around to all the houses, so he may need to hitch up a few more reindeer to pull it. Brute force quickly becomes impractical.

So, what to do? As I said in my first solution, I've was considering this as a constraint problem[3]. So, I decided to have a go with OptaPlanner. This did not work out. OptaPlanner proved much harder to get up and running than I expected (I'm still working on that), but also it became clear OptaPlanner is not actually aimed at solving this kind of problem. The problem is not NP hard and it turns out there are perfectly good high-performance algorithms for solving it. Then from a throwaway remark I made in a comment on one of the other solutions, I found myself thinking maybe using pattern matching inference rules might work. It probably wouldn't be as efficient as the algorithmic Python solution, but it would be using real rules.

So, I downloaded and installed Drools into Eclipse. I grabbed the Java classes I'd put together in my abortive efforts with OptaPlannner and created a starter Drools project using the wizard. Then, working from the algorithm I came up with for the Python solution, I tried typing in the following rule:

```
1   rule "Find last"
2     when
3       r : Reindeer()
4       not OrderingConstraint(inFront == r.name)
5     then
6       System.out.println("Determined that " + r.getName() + " is next Reindeer to add");
7   End
```

After I persuaded the project to run, slightly to my surprise and much to my delight the following line appeared on the console.

```
Determined that Dancer is next Reindeer to add
```

My previous solution was based on the idea that if there were no constraints saying that a given reindeer was in front of another, then that reindeer could be safely be

---

[2] See https://www.damirsystems.com/reindeer-ordering

[3] It turns out it isn't really – or at least not in the sense I was thinking – but in the manner of a far more erudite doctor 'I'll explain later' – in section 4.

considered as the next one to place (building the ordering back to front). What this rule says in effect is find me a Reindeer (line 3), such that there are no ordering constraints where this reindeer is named as being in front of another reindeer (line 4). And 'Dancer' is the only reindeer for which this is initially true.

The next step in my previous solution was having found such a reindeer, is to add the reindeer to the solution and also 'forget' about the constraints which mention it (since it's already positioned). I could in principle do the last bit by looping over the ordering constraints, but what is more fitting in the style of writing inference rules is to trigger another pattern matching rule. So, let's update the "Find Last" rule to:

```
1   rule "Find last"
2     when
3       r : Reindeer()
4       not OrderingConstraint(inFront == r.name)
5     then
6       System.out.println("Determined that " + r.getName() + " is next Reindeer to add");
7       result.add(r.getName());
8       modify(r) { setPositioned(true) }
9   end
```

Line 7 adds the reindeer to the results, and line 8 marks the reindeer as dealt with by setting the 'positioned' flag. The engine can propagate this to trigger the following rule:

```
1   rule "When Positioned"
2     when
3       r : Reindeer(positioned)
4       oc : OrderingConstraint(behind == r.name)
5     then
6       delete(oc);
7   end
```

Which simply says that for any reindeer which has been positioned (line 3), find all the ordering constraints which says it is behind another (line 4) and remove them (line 6).

Now when we run the project, the console prints out:

```
Determined that Dancer is next Reindeer to add
Determined that Donder is next Reindeer to add
Determined that Comet is next Reindeer to add
Determined that Vixen is next Reindeer to add
Determined that Blitzen is next Reindeer to add
Determined that Dasher is next Reindeer to add
Determined that Rudolph is next Reindeer to add
Determined that Cupid is next Reindeer to add
Determined that Prancer is next Reindeer to add
```

And but for a little formatting we have our solution – with only two little rules! And I'd though my Python solution was slick!

But this is a bit smoke and mirrors, like a swan serenely gliding across the water while hidden away is lots of frantic action. To get here I'd already written quite a lot of Java

code, so I could cook up this solution comprising only a couple of rules. What are the **OrderingConstraint** objects and **Reindeer** objects the rules use, and where did they come from? What's hiding underneath?

3 A (More) Complete Solution

Pragmatically generating a solution comprises the following steps:

- Read in Santa's rules
- Extract the basic ordering constraints and identify the reindeer
- Determine the order the reindeer should be hitched up
- Present the results

My initial 'smoke and mirrors' solution performs the first two steps in Java, the third in my two rules above and sort of skips over the final step leaving it to my debugging print statements in the "Find last" rule. Reading the input, and presenting the output is just code so legitimately done in Java, but I wanted to try and get the building of the constraints into the rules.

So, my (more) complete solution comprises:

- The **FindOrder** Java class which orchestrates initialisation of the Drools systems and loading the input data as well as printing out the result
- The **SantaRule** Java class which is a simple object (or in Java terms a POJO) representing one of Santa's rules as presented to the elves
- The **OrderingConstraint** class which is a POJO representing the fact that one named reindeer must be in front of another named reindeer
- The **Reindeer** Java class with is a POJO representing a reindeer
- The **OrderingRules** which is a Drools drl file containing rules

The full text of these is in the appendix. The **FindOrder** class is the only one with any real logic containing the technical details of initialising Drools, reading and parsing the input file of Santa's rules into a list of **SantaRule** objects and then inserting the objects into the Drools working memory. It's all very straightforward so I'm not going to discuss it in detail.

The seven rules in the rule file are the meat of the solution. Two rules create the **OrderingConstraint** objects we need from the **SantaRule** objects. Two rules then exact from these the **Reindeer** objects, so we know which reindeer we are working with. The next two rules work out the solution (and are similar to the ones we looked at in the previous section). Finally, the last rule catches the case where there is no solution to be found. Let's look at them in turn:

Each **SantaRule** object has three parts, the name of the reindeer the rule applies to, a 'direction' saying if the rule tells us about other reindeer in front of or behind this reindeer, and then the list of which the other reindeer are. So, the rule "Build 'behind' constraints" simply builds **OrderingConstraint** objects from **SantaRule** objects where the 'direction' is 'behind', and "Build 'in front of' constraints" rules does the same for **SantaRule** objects where the 'direction' is 'in front of'. As each **OrderingConstraint** object is created it is added into the working memory. Each rule essentially acts as a for

loop. There is then a nested loop in the then part of the rule – I think this too can be pushed into the rule conditions, but I've failed to figure out how to do it so far.

```
1   rule "Build 'behind' constraints"
2    when
3     s : SantaRule(direction == "behind")
4    then
5     for(String inFront: s.getRelations()) {
6      insert(new OrderingConstraint(inFront, s.getName()));
7      System.out.println("Added constraint - " + inFront + " must be in front of " + s.getName());
8     };
9   end
10
11  rule "Build 'in front of' constraints"
12   when
13    s : SantaRule(direction != "behind")
14   then
15    for(String behind: s.getRelations()) {
16     insert(new OrderingConstraint(s.getName(), behind));
17     System.out.println("Added constraint - " + s.getName() + " must be in front of " + behind);
18    };
19  End
```

Now we've created our **OrderingConstraint** objects, we can examine them to extract the names of the reindeer[4]. So, we have two rules to do this, one looking at the reindeer mentioned in the 'in front of' part of the **OrderingConstraint** object, one looking at the reindeer mentioned in the 'behind' part, the second condition in the rules (lines 4 and 13) just ensure we don't create the same reindeer twice:

```
1   rule "Find reindeer who are the 'in front' part of constraints"
2    when
3     oc : OrderingConstraint()
4     not Reindeer(name == oc.inFront)
5    then
6     insert(new Reindeer(oc.getInFront()));
7     System.out.println("Added reindeer " + oc.getInFront());
8   end
9
10  rule "Find reindeer who are the 'behind' part of constraints"
11   when
12    oc : OrderingConstraint()
13    not Reindeer(name == oc.behind)
14   then
15    insert(new Reindeer(oc.getBehind()));
16    System.out.println("Added reindeer " + oc.getBehind());
17  End
```

That's nice and simple. Next, we have the two rules which solve the problem:

---

[4] As in the Python solution, if Santa wanted to use a additonal reindeer, but didn't mention in a constraint, we'd have no idea where to put it in the line-up, so we have to assume all are mentioned in at least one constraint. So we don't need a separate list of reindeer names in the input, just Santa's rules

```
1   rule "Find first"
2    when
3     r : Reindeer()
4     not OrderingConstraint(behind == r.name)
5    then
6     System.out.println("Determined that " + r.getName() + " is next Reindeer to add");
7     modify(r) { setPositioned(true) }
8     result.add(r.getName());
9   end
10
11  rule "When positioned"
12   when
13    r : Reindeer(positioned)
14    oc : OrderingConstraint(inFront == r.name)
15   then
16    delete(oc);
17  end
```

These are slightly different from the rules in the previous section. Instead of looking for the reindeer who goes at the back, it looks for the reindeer which goes at the front, so the roles of the 'inFront' and 'behind' fields of the **OrderingConstraint** objects are reversed. This has the virtue of building the result in the correct order so one does not have to reverse it at the end (as happens in my Python solution).

Finally, we build a rule to check there really is a solution and Santa's rules are not inconsistent. If the rules are inconsistent, at some stage the "Find first" rule will fail to fire, because the constraints say that every reindeer must be behind some other reindeer (there is a circular dependency), so somewhere in working memory we'll have at least one reindeer which has not been positioned – we can express this as follows:

```
1   rule "Check for inconsistent constraints"
2    when
3     r : Reindeer(!positioned)
4    then
5     result.add("Inconsistent constraints - cannot find anywhere to put " + r.getName());
6   end
```

All this says is if a reindeer ends up not being positioned the constraints are inconsistent and lists out the reindeer we can't position. Typically, if it fires once, it fires many times.

So, we're done? No, we are not! There is a rather nasty problem hiding away here. And it's one of those things I have a bit of a bee in the bonnet about – the idea that 'declarative' code is a panacea, or more rather that the order of rules doesn't matter.

When I first ran my rules in Drools the right answer popped out sweetly just as I would hope. But being a suitably paranoid developer, I shuffled my rules around and ran it again and found my ordering list ended up with thirty-three entries not nine. Shuffling them around again, I now found I only got eight entries. What's going wrong?

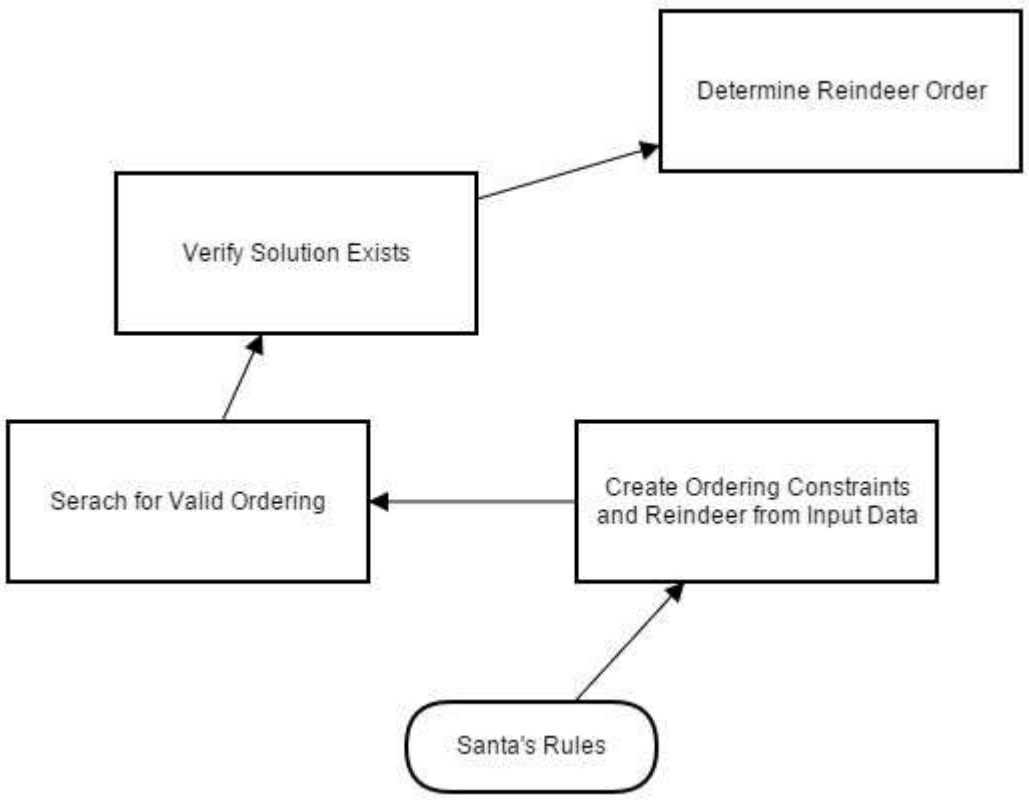
The problem is that the "Find first", "When positioned" and "Check for inconsistent constraints" rules all assume that they know about *all* the reindeer before they start. but

if we hit a constraint about ‘Cupid’ before we hit one about ‘Prancer’ in the “Find reindeer who are the '…' part of constraints” rules we add ‘Cupid’ to working memory and, the conditions of the "Find first" rule will be satisfied because ‘Prancer’ is not in working memory yet (so cannot match the first condition).

With the working memory in this state, if the "Find first" rule is selected for execution, ‘Cupid’ will get added to the results list prematurely. Likewise, the "Check for inconsistent constraints" rule can fire at once as soon as a reindeer appears in working memory, and if the engine decides to fire it before the "Find first" runs, we will be told the problem is inconsistent even when it isn’t. The order of the rules in the rule file affects the order the engine decides to execute them, with some orderings of the rules everything works fine, but in other orderings it does not.

The idea that a ‘declarative’ solution doesn’t care about order is wrong. And it is, in my experience, the exception rather than the rule (!) that we can write our rules in any old order and let the rule engine sort out which order to execute them. Yet the idea that the order of your rules doesn’t matter is a myth promulgated again and again by BRMS tools vendors. Yes, there are a lot of cases where the order doesn’t matter, but you can’t assume it because you are using a ‘declarative’ solution, you need to verify it.

There are multiple solutions to this execution ordering problem. The correct one (in my view) is to use a ‘rule flow’, to split the process into three separate steps. The first creating the **OrderingConstraint** objects and **Reindeer** objects from the first four rules described. The second would solve the problem with the next two rules (if a solution exists). The third would check a solution has been found. This would look nice as a DRD.



Another approach which is messier would be to add flags to the various objects to mark if they have been processed by the rules and extend the rule conditions to check the prerequisites have been done.

And yet another approach would be to add a priority to each rule (salience in Drools terminology) with the high priority rules being fired first. Shamefully, I have in fact adopted this latter approach in my solution (in effect associating the rules with the decisions in the DRD using priority). I've justified this to myself as it being the simplest option so the expedient one. On such a small set of rules it's ugly but acceptable, but explicit priority to dictate execution order is in general a very bad idea (as in my mind having explicit priorities in DMN decision tables is an extremely bad idea[5]).

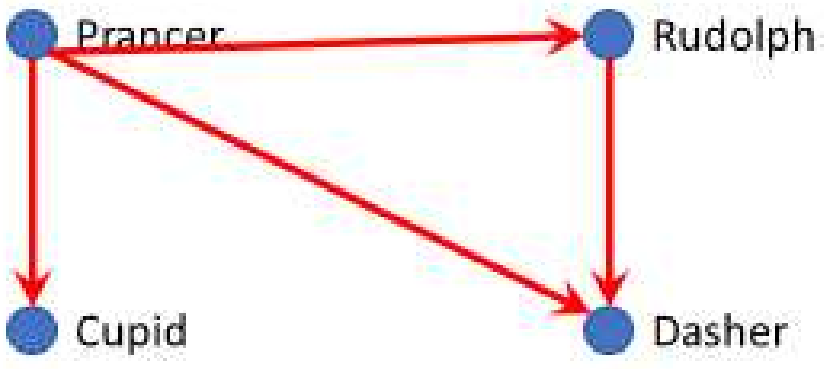
4 A Deeper Look at the Generalized Problem

From the beginning I thought the ordering problem was a particular instance some 'classical' problem in mathematics, but one of the things I scrupulously avoided when building the Python solution was to look about to see how this problem has been addressed in the past. However, between then and getting started on the Drools solution, I succumbed to temptation and tried to try and see if there are known approaches to. After a few false starts, I found that the problem of ordering Santa's reindeer is essentially equivalent to the problem of determining a topological sorting of a directed acyclic graph. This sounds rather academic and esoteric, but is actually an important practical problem in planning. Is it really a constraint problem? Well, sort of, but it is a very specialised one.

Let's consider a simplified version of the problem with only four reindeer not nine.

> Prancer in front of Cupid and Dasher
> Rudolph behind Prancer
> Rudolph in front of Dasher

This gives us four ordering constraints on our reindeer. We can draw a simple picture which captures the reindeer and the ordering constraints:



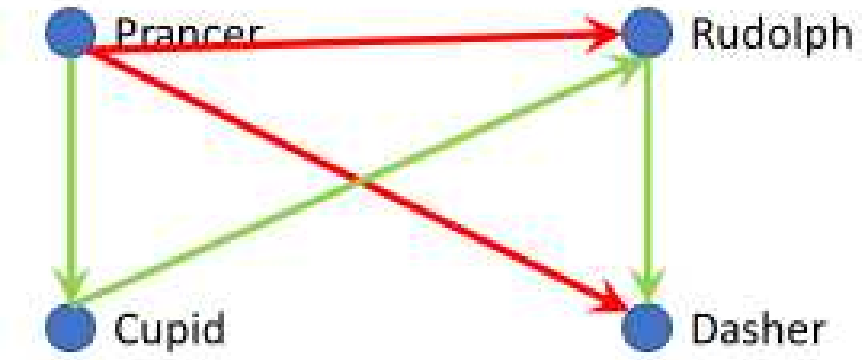
So, we can visualise our problem as a directed graph, where the vertices are the reindeer and the (directed) edges the ordering constraints. A solution to the problem is

[5] But I am perfectly happy with using implicit top down priority, but then that's the way most business people actually think

then a ‘topological ordering’ of the vertices, which is simply a list of the vertices in which each vertex which is at the ‘to’ end of an edge, is to the right of the vertex at the ‘from’ end of that edge. Here for example the ordering *Prancer, Rudolph, Cupid, Dasher* fits the bill.

Such a topological ordering exists if and only if there are no cycles in the graph (so it is a directed *acyclic* graph – commonly known as a DAG). In reindeer terms a cycle means the constraints lead to the conclusion that at least one reindeer must be behind itself – which means no ordering of the reindeer can satisfy Santa’s rules.

A topological ordering of a DAG is not in general unique – in this case *Prancer, Rudolph, Dasher*, *Cupid* and *Prancer, Cupid, Rudolph, Dasher* are also valid. If we modify the second of Santa’s rules to ‘Rudolph behind Prancer and Cupid’, adding another edge to the graph, only one solution exists. This is shown below with the green edges tracing out the unique topological sorting.



Wikipedia is as usual a fount of knowledge and provides details of the two well-known algorithms for generating a topological sorting – Kahn’s and depth first (see https://en.wikipedia.org/wiki/Topological_sorting). The algorithm used in both my Drools and Python solutions is essentially a variant of Kahn’s, although it could probably be made more efficient using a proper graph based object model rather than separately managing the edges and vertices.

Another thought – which I was intending to pursue before I found that solving the problem using rules was easier than I expected. In the full problem, Santa’s fifteen rules give rise to thirty-six ordering constraints (including the duplicates). However only eight of them are actually important. If we take the list:

Constraint 1 Rudolph is in front of Comet
Constraint 2 Prancer is in front of Comet
Constraint 3 Cupid is in front of Comet
Constraint 4 Cupid is in front of Blitzen
Constraint 5 Blitzen is in front of Donder
Constraint 6 Blitzen is in front of Vixen
Constraint 7 Blitzen is in front of Dancer
Constraint 8 Cupid is in front of Comet
Constraint 9 Cupid is in front of Blitzen
Constraint 10 Cupid is in front of Vixen
Constraint 11 Vixen is in front of Donder

Constraint 12 Dasher is in front of Donder
Constraint 13 Prancer is in front of Donder
Constraint 14 Prancer is in front of Rudolph
Constraint 15 Rudolph is in front of Donder
Constraint 16 Rudolph is in front of Dancer
Constraint 17 Rudolph is in front of Dasher
Constraint 18 Vixen is in front of Dancer
Constraint 19 Vixen is in front of Comet
Constraint 20 Donder is in front of Dancer
Constraint 21 Rudolph is in front of Dancer
Constraint 22 Blitzen is in front of Dancer
Constraint 23 Prancer is in front of Cupid
Constraint 24 Prancer is in front of Donder
Constraint 25 Prancer is in front of Blitzen
Constraint 26 Prancer is in front of Dasher
Constraint 27 Dasher is in front of Vixen
Constraint 28 Dasher is in front of Dancer
Constraint 29 Dasher is in front of Blitzen
Constraint 30 Comet is in front of Donder
Constraint 31 Cupid is in front of Donder
Constraint 32 Cupid is in front of Rudolph
Constraint 33 Cupid is in front of Dancer
Constraint 34 Rudolph is in front of Vixen
Constraint 35 Prancer is in front of Vixen
Constraint 36 Dasher is in front of Vixen

We only need the highlighted constraints, because the other twenty-eight can be deduced from them (for example “Prancer is in front of Rudolph” follows from Prancer is in front of Cupid and Cupid is in front of Rudolph). It should be possible to produce an inference rule with identifies and then eliminates the redundant constraints. One reason I didn’t take this idea too far was I couldn’t quite see what to do if the problem did not have a unique solution. But maybe it’s worth pursuing.

5 Conclusion

Well first of all, I need to apologise. I was very wrong in thinking you can’t use business rules to solve the generalised problem. It turns out with Drools you can get a simple and I think quite elegant solution. The solution reads in Santa’s rules in his original format and is indifferent to changes in names and numbers of reindeer. Also, having pulled the problem inside a rules engine it opens the opportunity to see how to integrate other kinds of ordering constraints like:

> “Blizten and Dancer cannot be next to one another”

Which would be difficult to cope with in using a programming language approach.

But it is not by any means perfect. For one thing, it gives me the chance to rant about unguarded reliance on ‘declarative’ as opposed to ‘procedural’ logic. It also is a little less robust than my Python solution with regard to poorly posed problems. While it can detect if there is no solution, it is unable to say if there is a unique solution – though probably with a little more work one could add something in to do this.

Then there is the question of performance and scaling. There are some of us who may have programmed in Prolog in our youth. One of the earliest examples one learns in Prolog is an almost magically neat trick for reversing lists. But then one is told never to use it because it is horrendously inefficient. And I suspect this solution is similar in that it looks very slick, but doesn't really perform very well.

The focused algorithms of graph theory provide a solution which scales linearly (or indeed better) as the size of the problem grows. While I've not benchmarked the Drools solution against larger problems, I suspect run times will grow at least quadratically and probably much worse as the number of reindeer and ordering constraints increase.

Even as things stand, there is a stark difference in performance. Simple timing probes show the Python solution executes on my laptop in less than 2ms, while the Drools solution takes around 1400ms, getting on for three orders of magnitude slower (and bear in mind Python while not exactly slow, is a fully interpreted language, while in appropriate circumstances modern Java can match C++ for performance). One can argue this is deceptive as about 1300ms of the time is the Drools system initialising, but even if one ignores this part of the processing, the Python solution is still about 50 times faster than Drools. Not very important for the elves with only nine reindeer to worry about, but if we wanted to pull out a topological ordering of a large DAG it would be very significant.

Anyway, it's time for a mince pie. So:

6 Appendix – Java Classes & Drools Rules

6.1 Java Classes

6.1.1 FindOrder Class

```
1   package jetsetbc.reindeer;
2
3   import java.io.*;
4   import java.util.*;
5   import java.util.regex.*;
6
7   import org.kie.api.KieServices;
8   import org.kie.api.runtime.KieContainer;
9   import org.kie.api.runtime.KieSession;
10
11  import jetsetbc.reindeer.domain.*;
12
13  public class FindOrder {
14
15    private static final Pattern RAW_CONSTRAINT =
16            Pattern.compile("(.*) (behind|in front of) (.*)",     Pattern.CASE_INSENSITIVE);
17
18    public static void main(String[] args) {
19      try {
20        System.out.println("Starting");
21        long t0 = System.currentTimeMillis();
22        // load up the knowledge base
23        KieServices ks = KieServices.Factory.get();
24        KieContainer kContainer = ks.getKieClasspathContainer();
25        KieSession kSession = kContainer.newKieSession("ksession-rules");
26        long t1 = System.currentTimeMillis();
27
28        // create list to hold the results and inject into working memory
29        List<String> result = new ArrayList<String>();
30        kSession.setGlobal("result", result);
31
32        // load Santa's rules from the input file and insert them into working memory
33        ArrayList<SantaRule> santaRules = parseFileString(args[0]);
34        for (SantaRule s : santaRules) {
35          System.out.println("Santa rule is : " + s);
36          kSession.insert(s);
37        }
38        long t2 = System.currentTimeMillis();
39
40        // execute the rules
41        kSession.fireAllRules();
42
43        // print the outcome
44        int idx = 1;
45        for (String s : result) {
```

```
46          System.out.println(idx++ + " " + s);
47         }
48
49         // timing info
50         long t3 = System.currentTimeMillis();
51         System.out.println("Total Execution time is : " + (t3 - t0));
52         System.out.println("KIE/Drools init time is : " + (t1 - t0));
53         System.out.println("Build of Working memory time is : " + (t2 - t1));
54         System.out.println("Rule execution time is : " + (t3 - t2));
55        } catch (Throwable t) {
56         t.printStackTrace();
57        }
58       }
59
60       // simple logic to read and parse a file containing Santa's rules
61       private static ArrayList<SantaRule> parseFileString(String fileName) throws Exception {
62        ArrayList<SantaRule> santaConstraints = new ArrayList<SantaRule>();
63        BufferedReader reader = new BufferedReader(new FileReader(fileName));
64        String inputline = null;
65        while (null != (inputline = reader.readLine())) {
66         inputline = inputline.trim().replace(".", "");
67         Matcher match = RAW_CONSTRAINT.matcher(inputline);
68         if (match.find()) {
69          String name = match.group(1).trim();
70          String direction = match.group(2);
71          String[] relations = match.group(3).trim().
72                    replaceAll(" and ", ",").replaceAll(" ", "").split(",");
73          santaConstraints.add(new SantaRule(name, direction, relations));
74         }
75        }
76        reader.close();
77        return santaConstraints;
78       }
79
80      }
```

6.1.2 SantaRule Class

```
1       package jetsetbc.reindeer.domain;
2
3       public class SantaRule {
4               private String name;
5               private String direction;
6               private String[] relations;
7
8               public SantaRule(String name, String direction, String[] relations) {
9                       this.name = name;
10                      this.direction = direction;
11                      this.relations = relations;
12              }
13
14              public String getName() { return name; }
15
```

```
16              public String getDirection() { return direction; }
17
18              public String[] getRelations() { return relations; }
19
20              public String toString() {
21                      String relationsList = "";
22                      for(String relation: relations) relationsList += relation + ",";
23                      return name + " " + direction + " " +
24                          relationsList.substring(0, relationsList.length()-1);
25              }
26      }
```

6.1.3 OrderingConstraint Class

```
1       package jetsetbc.reindeer.domain;
2
3       public class OrderingConstraint {
4               private String behind;
5               private String inFront;
6
7               public OrderingConstraint(String inFront, String behind) {
8                       this.inFront = inFront;
9                       this.behind = behind;
10              }
11
12              public String getBehind() { return behind; }
13
14              public String getInFront() { return inFront; }
15      }
```

6.1.4 Reindeer Class

```
1       package jetsetbc.reindeer.domain;
2
3       public class Reindeer {
4               private String name;
5               private boolean positioned;
6
7               public Reindeer(String name) { this.name = name; }
8
9               public String getName() { return name; }
10
11              public boolean getPositioned() { return positioned; }
12
13              public void setPositioned(boolean positioned) {
14                      this.positioned = positioned;
15              }
16      }
```

6.2 Drools Rules File

```
1   package jetsetbc.reindeer
2
3   import jetsetbc.reindeer.domain.*;
4   import java.util.List;
5
6   global List<String> result;
7
8   // ***************** phase 1 rules *************
9   rule "Build 'behind' constraints"
10  salience 20
11   when
12    s : SantaRule(direction == "behind")
13   then
14    for(String inFront: s.getRelations()) {
15     insert(new OrderingConstraint(inFront, s.getName()));
16     System.out.println("Added constraint - " + inFront + " must be in front of " + s.getName());
17    };
18  end
19
20  rule "Build 'in front of' constraints"
21  salience 20
22   when
23    s : SantaRule(direction != "behind")
24   then
25    for(String behind: s.getRelations()) {
26     insert(new OrderingConstraint(s.getName(), behind));
27     System.out.println("Added constraint - " + s.getName() + " must be in front of " + behind);
28    };
29  end
30
31  rule "Find reindeer who are the 'in front' part of constraints"
32  salience 20
33   when
34    oc : OrderingConstraint()
35    not Reindeer(name == oc.inFront)
36   then
37    insert(new Reindeer(oc.getInFront()));
38    System.out.println("Added reindeer " + oc.getInFront());
39  end
40
41  rule "Find reindeer who are the 'behind' part of constraints"
42  salience 20
43   when
44    oc : OrderingConstraint()
45    not Reindeer(name == oc.behind)
46   then
47    insert(new Reindeer(oc.getBehind()));
48    System.out.println("Added reindeer " + oc.getBehind());
49  end
50
51  // ***************** phase 2 rules *************
```

```
52  rule "Find first"
53  salience 10
54    when
55      r : Reindeer()
56      not OrderingConstraint(behind == r.name)
57    then
58      System.out.println("Determined that " + r.getName() + " is next Reindeer to add");
59      modify(r) { setPositioned(true) }
60      result.add(r.getName());
61  end
62
63  rule "When positioned"
64  salience 10
65    when
66      r : Reindeer(positioned)
67      oc : OrderingConstraint(inFront == r.name)
68    then
69      delete(oc);
70  end
71
72  // ***************** phase 3 rules *************
73  rule "Check for inconsistent constraints"
74  salience 0
75    when
76      r : Reindeer(!positioned)
77    then
78      result.add("Inconsistent constraints - cannot find anywhere to put " + r.getName());
79  end
```