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

(Bob Moore, JETset Business Consulting, 1 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 Getting Started

So, what are we trying to do? Surely we are doing a bit more that just sorting out the order of the reindeer for Christmas 2017? If that was it, it makes more sense to do it with pencil and paper. My interpretation is that to satisfy the challenge I need to produce some kind of automated system that can work out the correct order for years to come, on the basis that the 'type' of rules remains the same, but the actual reindeer and rules vary year to year. It ought also to be able to tell if the rules are consistent (so an order actually exists) and if the solution is unique (only one order satisfies all the rules).

I'm pretty sure problems equivalent to this challenge form part of the standard literature, but it's more fun to work it out from first principles, so I decided to forgo Google and try and do it by myself. However, last night I found myself speculating about to merge multiple partial orderings into a single total ordering instead of going to sleep. I had little luck figuring how to solve it (and not as much sleep as I'd have liked either).

Setting about it this morning, I decided the first step to finding a way to solve the problem was to get the 'rules' into some kind of uniform structure, on the grounds I'd definitely need to do that at some stage and if I did it first, I could try solving things manually which could give me insights into build an automated solution.

The first observation is that the ‘rules’ we are given are not the ‘production rules’ of the type we see in decision tables, or tools like ODM and Blaze (which allow me to deduce new information when their conditions are satisfied). Instead we are looking at a list of constraints – things we use to determine if a proposed solution is correct or not. Furthermore, they all can be expressed in a canonical form each stating that in a valid solution particular reindeer are in front of particular other reindeer. In fact, ‘rule 1’ is three distinct and independent constraints:

Rudolph is in front of Comet
Prancer is in front of Comet
Cupid is in front of Comet

When we expand the 15 ‘rules’ we are given, we end up with 36 ordering constraints (well not quite as we see in a moment). I could have rearranged everything using cut and paste in a text editor, but I’m trying to improve my skills with Python, so thought I’d practise a bit, by using it to build a primitive parser to read in the rules and format them in a uniform manner. After 30 minutes or so and 30 or so lines of Python, I got Python to read in the problem text and spit out the following neatly formatted list of constraints:

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

Which is a start. However, there are a couple of issues with this. Firstly, there seem to be duplicates. Constraints 4 and 9 both tell us that Cupid is in front of Blitzen. If we look back to the original rules we see that rule 2 tells us Blitzen is behind Cupid, while rule 4 tells us Cupid is in front of Blitzen (as well Comet and Vixen) so we have some redundancy. The second issue is the ordering of the constraints is random (like the original rules) so it's hard to see if there a pattern of some sort we can exploit to find our way to a solution

Deduplicating is not difficult – and it quickly emerges there are only 30 unique constraints[1]. As for ordering I decided simply to order the constraints alphabetically – at which point the solution leapt out at me. The ordered, deduplicated constraints are:

Constraint 1 Blitzen is in front of Dancer
Constraint 2 Blitzen is in front of Donder
Constraint 3 Blitzen is in front of Vixen
Constraint 4 Comet is in front of Donder
Constraint 5 Cupid is in front of Blitzen
Constraint 6 Cupid is in front of Comet
Constraint 7 Cupid is in front of Dancer
Constraint 8 Cupid is in front of Donder
Constraint 9 Cupid is in front of Rudolph
Constraint 10 Cupid is in front of Vixen
Constraint 11 Dasher is in front of Blitzen
Constraint 12 Dasher is in front of Dancer
Constraint 13 Dasher is in front of Donder
Constraint 14 Dasher is in front of Vixen
Constraint 15 Donder is in front of Dancer
Constraint 16 Prancer is in front of Blitzen
Constraint 17 Prancer is in front of Comet
Constraint 18 Prancer is in front of Cupid
Constraint 19 Prancer is in front of Dasher
Constraint 20 Prancer is in front of Donder
Constraint 21 Prancer is in front of Rudolph
Constraint 22 Prancer is in front of Vixen
Constraint 23 Rudolph is in front of Comet
Constraint 24 Rudolph is in front of Dancer
Constraint 25 Rudolph is in front of Dasher
Constraint 26 Rudolph is in front of Donder
Constraint 27 Rudolph is in front of Vixen
Constraint 28 Vixen is in front of Comet
Constraint 29 Vixen is in front of Dancer
Constraint 30 Vixen is in front of Donder

[1] The constraints are 'unique' but that does not mean they are 'necessary'. It's highly likely that some of the 30 constraints could be removed and the same unique solution would still emerge – but that is a different challenge!

Looking down the list what struck me was that there is only one constraint telling me Donder is in front of anyone. So, I thought – Donder must be near the back[2]. Then it occurred to me the constraint says Donder is in front of Dancer, and there are *no* constraints saying Dancer is in front of anyone. So, I deduced Dancer must actually be at the back!

For this conclusion to logically follow, we strictly need an extra assumption. We hope from the wording of the challenge that there is only one ordering of the reindeer which satisfies all the constraints. However, in theory there might be no arrangement of Santa's reindeer which satisfy all 30 constraints, or there may be several.

What is clear though is that if we do have a solution to the problem where Dancer is not the last reindeer, then we can get another solution by simply taking that solution and moving Dancer to the end of the line. If the first solution satisfies all the 'is in front' constraints for reindeers other than Dancer, so will the second, and a solution with Dancer at the rear satisfies automatically all the constraints involving Dancer since all the other reindeer are in front of her[3]. So, if the solution is unique as we hope, Dancer must be the last reindeer in the line.

3 Solving The Problem

While mulling the problem over in the wee hours, I almost at once discarded the idea of using decision tables to attack this, and wasn't happy about trying to use inference rules. They might work, but I didn't see things being sufficiently focused to be efficient. I wanted an approach which would scale reasonably. After all if scaling wasn't part of the challenge one might as well go for a brute force approach and just try every permutation – 9 factorial isn't really all that big.

At one stage I did contemplate doing some sort of backtracking search and perhaps blowing the dust off my Prolog skills (or lack thereof). However, by now my insights had me convinced was the problem was substantially about lists of things (constraints and reindeer) and comparisons, and whatever technology I was going to use ought to be good with lists.

At which point I still had my Python editor open fresh from formatting the constraints. One of Python's strengths is the ease with which it manipulates lists and other sorts of collections. So why not practise my Python a bit more?

The full code – including my noddy little parser is in the appendix, but on the assumption some readers will have even more difficulty understanding Python than I do, I'll step through the solution in stages here.

Firstly, here is the top-level Python function[4]:

---

[2] Someone else might have spotted 'Prancer' must be close to the front – indeed she is at the front. My solution works back to front, but an equivalent solution exists working front to back

[3] Choosing this pronoun diverted me for a couple of hours googling debates on the gender of Santa's reindeer. Some think there is a mix, some they are all female. My wife is reading a book which argues the does are all pregnant at Christmas, so shouldn't be flying, while the bulls are all worn out by rutting. So, the book argues, they are probably all gelded males

[4] Note that there are some comments in the real code in the appendix. I've just edited them out in the text here.

```
1    def solve(fileName):
2        constraintSet = readConstraints(fileName)
3        unpositionedItemSet = findItems(constraintSet)
4        solution = nextStep(unpositionedItemSet, constraintSet, [])
5        print('Solution is ')
6        printSolution(solution)
```

This takes as a parameter an input text file containing the 'rules' in the original format presented in the challenge (less the numbering and the trailing full stop). The function readConstraints called in line 2 parses the file and extracts and deduplicates the individual constraints. The constraints are returned as a set (not a list) of ordered pairs (tuples in Python) of reindeer names each pair expressing the constraint that the first reindeer, must be in front of the second reindeer in any valid solution. For the specific constraint set we are working with, there are 30 such pairs.

In line 3, the function findItems[5] is called to extract all the distinct reindeer names in the constraints, of which there are nine[6].

The next step is to call the function nextStep. This is where all the clever stuff takes place as we will see shortly. This takes three parameters, the set of reindeers we haven't worked out a position in the line for, the set of constraints that are currently of interest, and the list[7] of reindeers we do know the position of in the line.

When we call nextStep for the first time, evidently the first set will be all the reindeers, the second set will be all the constraints, and the list will be empty because we haven't worked out any positions yet. When nextStep returns though it will have the list of reindeer names in the correct order for hitching up, (with Prancer at the front and Dancer at the rear in this case).

Finally, the last two lines simply print out the solution to the console.

I'm not going into the gory details of readConstraints since it's just basic I/O and is in the appendix for those really interested. Suffice to say we get back a set where the elements look like:

('Dasher', 'Blitzen')

Which expresses the fact that Dasher must be in front of Blitzen (see 'rule' 12). The logic of findItems is very simple, but is a good introduction to the kind of power you get from Python's collection capabilities. The code is as follows:

```
1    def findItems(constraintSet):
2        return {c[0] for c in constraintSet}.union({c[1] for c in constraintSet})
```

[5] I envisage the 'solution' as being for an abstract ordering problem, so I've just not been able to bring myself to use the word 'reindeer' in the code. However, wherever you see 'item' you can think 'reindeer'
[6] Note if there was a reindeer whose name didn't appear in one of the constraints, that reindeer could go anywhere in the line-up without violating any constraints, so we'd have multiple valid arrangements
[7] Most of the Python code in the solution works with sets, because order is unimportant and because they are implemented as hash tables, searching sets is (in theory) faster than searching lists. However, sets have no order, so the result must be a list.

The whole of the logic is in line 2. The expression {c[0] for c in constraintSet} is a 'set comprehension' – it says create a set by taking the first element of each ordered pair in the set of constraints (constraintSet). The rest of this line union({c[1] for c in constraintSet}) is saying create another set from the second element and then form the union with the first set.

And this one line of code identifies all the reindeers mentioned in all the constraints. Which you can think of either as pretty slick or too clever by half.

Before we move onto detailed examination of the function nextStep let's go back to where we were at the end of section 2 to check we know how to do proceed.

Our starting point is that we have:

- a set of reindeer we need to work out the position of
- a set of constraints
- an ordered list of the reindeer we have already worked out where to put (which is initially empty)

Now we worked out in section 2, that if we could find a reindeer which was not mentioned in the first half of any of our constraints, then we would know that that reindeer must be behind all the other reindeer we hadn't found a position for. For our specific example in the beginning there are no constraints of the form "Dancer is in front of X", so Dancer is at the end of the line.

Now once we have worked this out, we can put Dancer in our list of reindeer we know the position of, and remove her from the set of un-positioned reindeer (one down eight to go), but – and this is the crucial bit – we can also forget about any constraints which refer to Dancer – we've already positioned her, so they are no longer of any relevance to us. So now we can reduce the number of constraints by six to only twenty four.

Now let's look at our reduced set of un-positioned reindeer and our reduced set of constraints. If you go back to the list of constraints on page 3 you can see we will by now have got rid of the one constraint of the form "Donder is in front of X", because in that, the constraining 'X' was Dancer. So, we see Donder must be the next to last reindeer immediately in front of Dancer. We eliminate references to Donder from our two sets and then find Comet is our only option for the next reindeer and so on.

So, I've described it words – what's it look like in code?

```
1   def nextStep(unpositionedItemSet, constraintSet, order):
2       if len(unpositionedItemSet) == 0:
3           return reversed(order)
4
5       unresolvedPositionSet = {c[0] for c in constraintSet}
6       found = unpositionedItemSet.difference(unresolvedPositionSet)
7
8       if len(found) == 0:
9           raise Exception('whoops can not find an item - constraints are inconsistent')
10      elif len(found) > 1:
```

```
11          print('whoops found more than one item ' + str(found))
12
13      item = found.pop()
14      order.append(item)
15
16      unpositionedItemSet.remove(item)
17      stillRelevantConstraintSet = {c for c in constraintSet if c[1] != item}
18
19      return nextStep(unpositionedItemSet, stillRelevantConstraintSet, order)
```

Considering what it is doing, this code packs a lot into very few lines, which is why I'm getting to like Python even if commits the vile sin of being layout sensitive.

The astute reader will have anticipated before we got this far that our solution is going to use recursion[8]. The first step (lines 1 and 2) are to see if the recursion is over. We check if the set of un-positioned reindeer is empty. If it is, we have positioned all the reindeer, so we return our list of positions unwinding the recursion. Before we do this, we reverse the list as we build it back to front (we add Dancer first, Donder second etc). We append items to the list on line 14, rather than pushing them onto the front.

The next step – line 5 – is to work out which reindeer still have a constraint of the form "I'm in front of x". The logic we worked out above means we don't (yet) know what to do with these reindeer. This gives a set of things (unresolvedPositionSet) which we know we don't know what to do with. We also have a set (unpositionedItemSet) of things we haven't decided what to do with. If we take the set difference of these we end up with the set of things we (now) *do* know what do with. This is done on line 6.

This new set we have created (found) should hold exactly one element. Line 8 to 11 check this is the case. If it is not the problem is not well formed. If the set found is empty, then each reindeer has some other reindeer (other than the ones we've already determined the position of) it is to be in front of. This can only be if there is some circular dependency in the constraints, and so no solution exists. In this case the program raises an exception (line 9) and terminates. If there is more than one entry in the set found, then the solution to the problem is not unique. In practise what it means is that there are at least two reindeer such that they can be interchanged on the line while preserving all the constraints. The program carries on but outputs a warning (line 11) that only one solution out of many is being given[9].

In line 13 and 14 we pick an element out of the set found (ideally the only element), and append it to our partial solution. Then, in lines 16 and 17, we remove our selected reindeer from the list of unplaced reindeer and filter out of the remaining constraints which refer to it.

---

[8] This is in deference to good functional programming practise, but in truth Python is probably better optimised for an iterative style. My first solution was all for loops and while loops, but I felt obliged to be true to the proper style of this kind of solution and replaced as much of these as possible by 'comprehensions' and recursion.

[9] So, an obvious enhancement is to modify the logic to find all solutions

Finally, in line 19 we call `nextStep` recursively to find the next reindeer to put into the line. Since we reduce the size of the set `unpositionedItemSet` with each call, the recursion is guaranteed to complete with one more call than there are reindeer.

4 And the answer is?

The reindeer should be arranged in the order front to back as follows:

1 Prancer
2 Cupid
3 Rudolph
4 Dasher
5 Blitzen
6 Vixen
7 Comet
8 Donder
9 Dancer

5 Conclusion

Looking at the solution, it is at heart a very simple constraint engine, able to solve a very simple kind of constraint problem. The first half of the `nextStep` function is basically doing constraint propagation, using the constraints to squeeze out the next step in reaching a solution, while the second half then reduces the search space. It is missing any functionality to 'backtrack' to find alternate solutions, but as long as the solution is unique this is not an issue.

Constraint problems fit well with using DMN to model the components of a decision-making process in a diagrammatic manner, but at the lower level, decision tables and production rules are rarely the right technology for solving these kinds of problems. Techniques of constraint propagation and search space reduction – which this solution uses in a somewhat oblique manner are more appropriate

The approach taken here is simple, direct and specific. The problem of Santa's reindeer is constrained, so a simple direct approach is justified. The solution is indifferent to the number of constraints or reindeer we have, but depends heavily on the simplicity of the ordering constraints. It is not immediately obvious how to cope if we wanted to add in a constraint like:

> "Blizten and Dancer cannot be next to one another"

If we want to handle more general kinds of ordering constraints we really need to look at a more general type of constraint/optimization engine. I'm hoping to have another go at this with Optiplanner if I have time, to see how I could add in things like a constraint of this nature.

In the meanwhile:

Good Luck with the Sleigh Santa

And

Season's Greetings to One and All

6 Appendix – Constraints & Full Python Code

6.1 Data

Just copy and paste the data into some suitable file. The code assumes it is in C:\tmp\data.txt – see the last line of the Python code

Comet behind Rudolph, Prancer and Cupid
Blitzen behind Cupid
Blitzen in front of Donder, Vixen and Dancer
Cupid in front of Comet, Blitzen and Vixen
Donder behind Vixen, Dasher and Prancer
Rudolph behind Prancer
Rudolph in front of Donder, Dancer and Dasher
Vixen in front of Dancer and Comet
Dancer behind Donder, Rudolph and Blitzen
Prancer in front of Cupid, Donder and Blitzen
Dasher behind Prancer
Dasher in front of Vixen, Dancer and Blitzen
Donder behind Comet and Cupid
Cupid in front of Rudolph and Dancer
Vixen behind Rudolph, Prancer and Dasher

6.2 Python Code

Note in Python layout is very important as it uses layout and in particular indenting to delimit code blocks (why not use something sensible braces or BEGIN and END I ask). So, if you copy and paste from the PDF document it might introduce some little problems you'll needing fixing afterwards. If you want to try it for yourselves, you can always e-mail me for the code in plain text. Additionally it worth mentioning that the code is for Python 3, not Python 2.

```
import os
import re

# Print Routines to dump data

# lists reformated constraints
def printConstraints(constraintSet):
   constraintList = sorted(list(constraintSet))
   idx = 0
   for item in constraintList:
      idx +=1
      print('Constraint %2d %s is in front of %s' % (idx, item[0], item[1]))

# prints a solution
def printSolution(order):
   idx = 0
```

```
    for item in order:
        idx +=1
        print('%2d %s' % (idx, item))

# adds in the constraints represented by a constraint line to the overall
# constraint set
def buildConstraints(constraintSet, match):
    if match:
        fromRel = match.group(1) # constrained item
        position = match.group(2) # is this item behind or in front
        toRel = match.group(3) # items doing the contraining (one or more)

        # if there are more than one items in the 'constraining' part
        # we replace the 'and' by a comma, strip out the spaces
        # to give a resulting comma separated set of values
        pattern = re.compile('(.*) and ([a-z]+)', re.IGNORECASE)
        match = pattern.search(toRel)
        if match:
            toRel = (match.group(1) + ',' + match.group(2)).replace(' ', '')

        # now create a list from the constraining items
        toRelList = toRel.split(',')

        # iterate over constraining items (using list comprehension
        # and the 'position' information to create a set of tuples
        # representing the constraints from this line of input
        if position == 'behind':
            newRelSet = {(toRel, fromRel) for toRel in toRelList }
        else:
            newRelSet = {(fromRel, toRel) for toRel in toRelList }

        # finally add them in - since we are using sets, duplicates are
        # automatically eliminated
        return constraintSet.union(newRelSet)

    # catch all in case the constraints cannot be parse
    print('Could not do match on line')
    return constraintSet

# simple function to extract constraints from file - most of the work is
# delegated to buildConstraints
def readConstraints(fileName):
    file = open(fileName, 'r')
    pattern = re.compile('([a-z]+) (behind|in front of) (.*)', re.IGNORECASE)
    constraintSet = set()
    for line in file:
        constraintSet = buildConstraints(constraintSet, pattern.search(line))
    printConstraints(constraintSet)
```

```
    return constraintSet

# find the set of items of interest - which is basically anything mentioned
# in a constraint
def findItems(constraintSet):
    itemSet = {c[0] for c in constraintSet}.union({c[1] for c in constraintSet})
    print(itemSet) # this will be randomly ordered
    return itemSet

# this is the clever bit! The idea is we have a set of constraints,
# a set of unpositioned items and a partial solution of items which
# have been placed. We hunt through the constraints to find all items
# where there is a constraint which says they must be in front of
# another item.
# Then we look for an unpositioned item which is not in this list (for a
# unique solution there should only be one, if there are none, there is
# no solution). This item is the next one to put into our partial solution.
# We then remove any constraints relating to this item from the list of
# constraints and list of unpositioned items. Once the latter is empty we
# are done. Finally we call nextStep again to do the next step
def nextStep(unpositionedItemSet, constraintSet, order):
    # if the if unpositioned Item Set is empty we are done, so return
    # and unwind the recursion stack
    if len(unpositionedItemSet) == 0:
        # reverse order so it is front to back not back to front
        return reversed(order)

    # find items which have a constraint saying they are ahead of some
    # other item, then diff with the unplaced items to find one where
    # no such constraint exists
    unresolvedPositionSet = {c[0] for c in constraintSet}
    found = unpositionedItemSet.difference(unresolvedPositionSet)

    # sanity check that a solution exists and is unique
    if len(found) == 0:
        printConstraints(constraintSet)
        raise Exception('whoops can not find an item - constraints are inconsistent')
    elif len(found) > 1:
        print('whoops found more than one item ' + str(found))

    # pop the next item into the solution NOTE this list is built in reverse
    item = found.pop()
    order.append(item)
    print('Adding ' + item + ' to partial solution')

    # filter the unpositioned item and constraints to remove irrelevant ones
    unpositionedItemSet.remove(item)
    stillRelevantConstraintSet = {c for c in constraintSet if c[1] != item}
```

```
    # make next step with recursive call
    return nextStep(unpositionedItemSet, stillRelevantConstraintSet, order)

# top level routine - call with file containing constraint list
def solve(fileName):
    constraintSet = readConstraints(fileName)
    unpositionedItemSet = findItems(constraintSet)
    solution = nextStep(unpositionedItemSet, constraintSet, [])
    print('Solution is ')
    printSolution(solution)

# kick off code for module
solve('C:\\tmp\\data.txt')
```