CS4402 - Practical 1: Bombastic

170008773

18th February 2018

Abstract

The planning of sequences of moves is a common problem in constraint solving, applicable in problems like scheduling to route planning. To be able to solve these problems efficiently they have to be modelled in such a way that solvers can work with the data efficiently. Here it is critical to note that the way these problems are modelled can have potentially huge impacts on how efficiently they can be solved. As an exercise in constraint modelling, I have modelled the game Bombastic in Essense Prime. In this report, I will detail the design choices, the testing of the model and the results of benchmarking the model with different levels of optimisation performed by the solver. I found that for the smaller problems there is not that much difference between the levels of optimisation but as the problems become more complex both using no optimisation and all the possible optimisation have significantly worse performance than using a moderate amount of optimisation.

1 Problem description

I will first start with a description of the problem I modelled.

1.1 The game

Bombastic is played on an $N \times M$ grid of cells. The cells in this grid can either be `dead`, `ice`, or `normal`. `dead` cells can be entered by neither the blocks nor the avatar, and other cells can accommodate at most one block or the avatar. For simplicity's sake, we assume that every grid is surrounded by a wall of dead cells. On this grid, there is an avatar and one or more blocks and a number of goals, *equal to the number of blocks*. The objective of the avatar is to walk around the grid and push the blocks around until all blocks are at a goal. In this scenario, we are not interested in which block ends up at which goals.

1.2 Avatar logic

The avatar is allowed to move around the grid, moving the cell it is currently occupying to any of the adjacent empty cells that are not dead. The avatar is only allowed to move purely horizontally or vertically (i.e. not at the same time) and not moving is also disallowed. The player is allowed to move onto an ice cell but when it moves off that cell again, the ice will crack, turning the cell into a dead one.

1.3 Box logic

The avatar can move blocks by moving into their square. This will move the block one square in the direction in which the avatar is travelling. This is only allowed if the cell into which the block is moving is not dead and does not contain another block. The avatar may only allowed to push one block at a time.

1.4 Objectives

Given the grid layout, the positions of the blocks, the position of the goals, and the initial position of the avatar, the objective is to find a sequence of legal avatar moves that move all the blocks such that every goal is occupied by a block. In this instance, the number of moves is provided. The problem is then to find whether there is a sequence of the given length that satisfies all the objectives and constraints.

2 Modelling the problem

I will now detail how the assignment was set up, the major modelling ethos used and the new instances I designed.

2.1 Setup

In this instance, I was required to use the modelling language `Essence prime` in conjunction with the constraint solver Savile Row. I was also provided with the decision variables, their domains and several sets of parameters to test the system. I was required to add the constraints to accurately model the problem and solve the provided instances, as well as design several new instances.

2.2 Modelling decisions

Redundancy As mentioned previously, the way the problems are modelled can have big impacts on how efficiently they are solved. The fewer constraints there are to check, the less work the solver has to perform. So there is an incentive to make the constraints as simple and as small in number as possible. On the other hand, however, these models, even ones that are well constructed, can be very complex to interact with on a human level. To mitigate this, extra constraints that overlap with, or even imply other constraints can be used as sanity checks. Ultimately, these are judgement calls as either extreme is likely to cause problems. In the modelling of this problem, I opted to lean towards redundant constraints, feeling that an extreme focus on performance was not needed considering the scope of the project.

Note however that this can significantly increase the work in heavily optimised runs, especially if the constraints are simple, checking them might not be a problem. However, it might be less simple to reduce them to common expressions etc. which is what the highest level of optimisation tries to do. Later in section 3 we will see that in the more complex instances, the heavily optimised runs can struggle hugely. This is one of the possible causes of that.

Elegance In contrast to redundancy, it benefits both human and machine to simplify the constraints as much as possible. This practice can make the model more understandable to humans and easier to solve for the machine. Where possible I attempted to make the constraints as small as possible, trying to eliminate conjunctions or disjunctions where possible.

Global constrains It is also worth mentioning that I tried to use global constraints such as `GCC` where possible. This was done because constrains solvers can often handle global constraints more efficiently than individual constraints. This, however, was not always possible. I chose readability over the use of global constraints since I did not think that the potential performance increases of using complicated global constraints would be worth the decrease in readability.

2.3 Model description

Overview The model is roughly divided into four parts: Initialisation, Game logic, Avatar logic and Box logic, each of which I will discuss below.

Initialisation This is fairly straightforward. I was given the initial positions for the goals, the blocks, the map layout and the avatar. In this section, I simply looped over all our environment variables at time 0 and set them to the initial parameters received.

Game logic The game logic section is mainly concerned with detailing the end conditions of the game, namely that all the goals must be occupied by a block, and how the cells in the map are or are not supposed to change. Here I chose to add some redundant constraints as a sanity check, which state that normal cells stay normal and dead cells stay dead. Finally, ice cells crack when the avatar leaves them. Since not moving is disallowed, we did this by checking when the avatar enters the ice cell and killing the cell in the next step.

Avatar logic The avatar logic is also relatively straightforward. Avatars cannot enter dead cells, their position should be updated using the `moveCol` and `moveRow` variables and they are only allowed to move one square in either a vertical or horizontal direction. I chose to model the movement using the sum of the absolute values of the move variables because this simultaneously details which moves are allowed and disallow both diagonal movement and non-movement.

Box logic Box movement is the most complicated part of this model since it involves the greatest number of variables. Making a block move involves checking the blocks' position, the position of the avatar, the direction in which the avatar wants to move and the position to which the block itself would move. Here it is useful to note that we must add constraints that blocks are not allowed to move unless pushed by the player, to avoid the solver simply teleporting them to the goals at any point in time. Because this is the most complex part of the model we have opted to leave in the most redundant constraints. For example, the constraint that blocks can share a cell would have been sufficient to ensure that no two blocks can be pushed simultaneously, but since this depends on how the movement of the avatar is checked, we decided to add a separate constraint for this as a sanity check.

2.4 Designed instances

I was also required to design new instances of this problem class. I took this opportunity to design a few instances that can be used to either test specific parts of the modelled logic or some artificially difficult problem to test the performance. They are designed in pairs. Every problem has one parameter file that is solvable and one that isn't so that we can see how the performance compares to comparable problems. To make the process of visualising the results a little easier every impossible version of an instance requests a solution that is 1 step shorter than the shortest solution, even if it is made impossible by other modifications. We will discuss them below. Humanly readable maps of all the instances are provided in the files. The numbers in the list correspond to the appropriate file name.

10) This problem is a slight adaptation of the `Bombastic1_1.param`. It is a single-width corridor with two blocks in it. This is designed to test the avatar's inability to push more than one block at the same time.

11) This is another instance designed to test the ice mechanic. There is an L shaped corridor with an ice cell at the intersection on which the avatar starts and a block and goal in each branch. The possible version removes the ice cell.

12) This problem contains a square of ice in the middle with two single-width corridors with a block on either end. The possible version has one ice cell replaced with a normal one, making it possible. This instance was designed to test how the solver deals with ice planning, and whether the ice mechanic is done correctly.

13) This instance is a slightly more complicated instance to test the inability of the avatar to push more than one box at a time. It has a small room and two blocks in a row with goals on two sides. This instance requires a tiny bit of planning since the avatar will First have to walk around the blocks.

14) This instance is simply a large open room with no complications. It is designed to test how the solver performs in terms of the length of the solutions and the size of the possible moves while the actual solution is very uncomplicated.

15) This instance is a simple adaptation of `Bombastic9_17.param`. It is identical to the old instance but it has a relatively big empty space added at the bottom that should change nothing about the solution. This was done to test how well the solver would fare if a lot of useless space was added.

3 Emperical evaluation

In this section, I will describe the testing process used and discuss the results. Savile Row has several levels of optimisation (levels 0 through 3). In these levels, it performs different optimisation steps. These steps include but are not limited to common subexpression elimination, unifying equal variables and filtering domains. These optimisations can save work during the search phase but can also be quite expensive themselves, making them potentially useless or even worse. Here we perform empirical analyses to see how the different levels of optimisation compare.

3.1 Testing process

The testing process was fairly straightforward. In addition to the testing I did while designing the system, I used both the provided instances as well as my own designed instances described above. As discussed above I designed some instances with the intention of testing specific aspects of the model and others were designed to test the overall performance of the model.

I used a python script to run the solver on every instance with all the optimisation levels and automatically record the data. This data consisted of an instance id, the number of steps of the requested solution, the optimisation level, whether a solution was found and the computation time taken by the solver. I measured the computation time in CPU seconds spent in user mode (as described by the bash module `time`). The remaining data was obtained by a simple parsing of the output, the solver and parameter file. The code for the script can be found in appendix A. The script should be run as `` `python timer.py` `` and it will recursively search through the directory tree from where it was called and process all the `.param` files it finds.

3.2 Results

In this section, I will discuss the computation time of Savile Row as a function of two measures: the level of optimisation and the length of the requested solution. As can be seen in table 1 the computation time varies very little for levels 1 and 2, but can vary an enormous amount for levels 0 and 3 as will be discuss in more detail below

	compTime	
	mean	std
optLevel		
0	327.668276	931.351282
1	6.343345	4.575549
2	7.345448	4.839894
3	213.884966	679.810891

Figure 1: Computation time statistics by level of optimisation

3.2.1 Computation per step

Because of the intricate nature of constraint problems, in general, I cannot make precise claims about space or time complexity of these problems. Therefore I will only discuss the empirical results found. In figure 2 you will find the computation times of all the instances that succeeded.

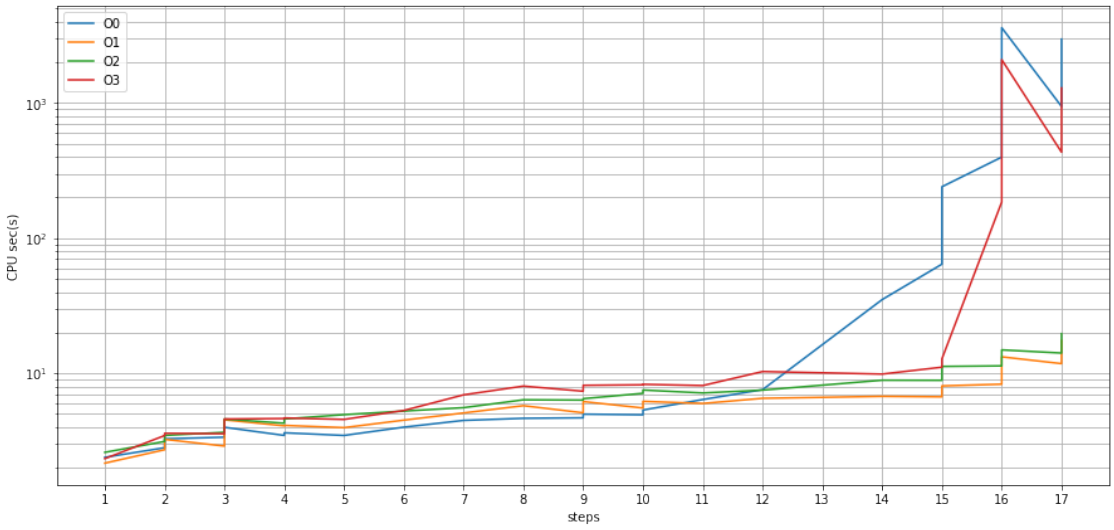

Figure 2: The number of steps of the requested solution vs. user CPU time. (lower is better) Note that the scale used is logarithmic

When the problems are relatively small, the level of optimisation doesn't matter that much. For the simpler problems, the least number of optimisation seems to be best, which makes sense since the extra overhead of optimisation might not be worth it for such small problems. As we increase the length of the requested solution we see that levels 1 and 2 remain roughly linear. Optimisation level 0 and 3 however, eventually start exhibiting exponential behaviour. For level 0 this is what we expected since the search space grows too fast if we don't make smart optimisations for this.

Heavy optimisation What happens to level 3, however, is slightly more counter-intuitive. This can explained as follows: Optimisation is a lot of work and if the search space is less conducive to optimisation it can waste a lot of effort. I assume that this is what happened since I was not able to model the problem using many global constraints. These global constraints are easiest the optimise away, so the fact that we didn't use them that much might make the heavy optimisation not worthwhile. As we mentioned earlier this is where the solver can spend a lot of extra time trying to optimise our redundant constraints away.

3.2.2 Complexity vs length

When looking at the tail end of the graph above, it is clear that while levels 1 and 2 perform fairly well with longer solutions, levels 0 and 3 start performing quite badly. Now I will examine the difference between level 0 and level 3 more closely.

Here it is useful to look at the instances I designed (number 14 and 15 specifically). These contain large open spaces that do not serve any special purpose. Here we can see that Level 3 solidly outperforms level 0. Presumably, because it doesn't get stuck walking around in the large spaces, offsetting the cost of the optimisation. I presume that this trend would continue as the length of the requested solutions continues to grow.

Relative performance of instances In this context it is also useful to compare the performances of the same instance between the possible and impossible versions since these have similar (or identical) complexity but differing solution lengths. The relative difference between the computation times of the same instance can be found in figure 3 below.

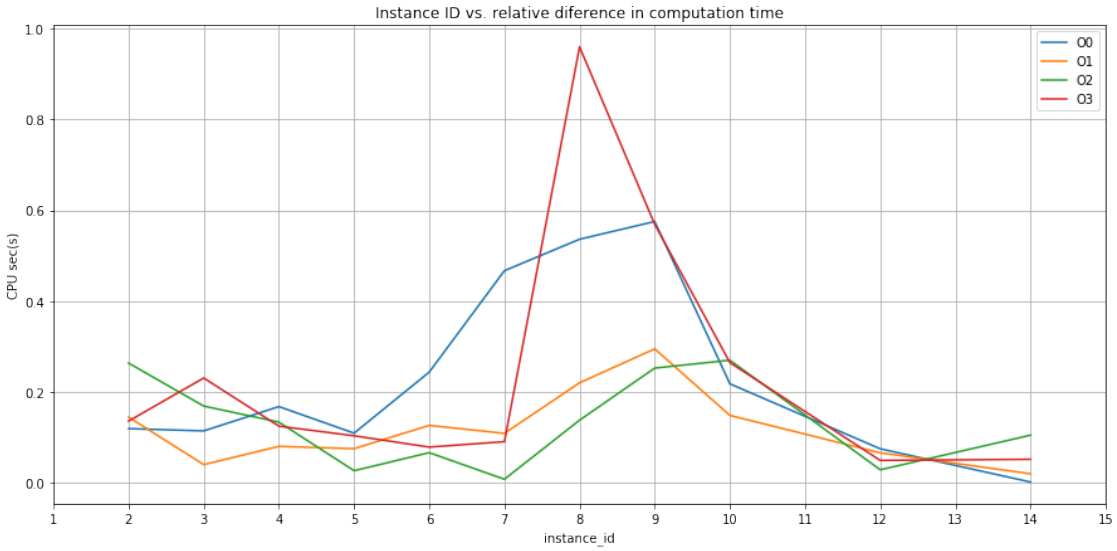

Figure 3: The instance id vs. absolute difference of user CPU time between the possible and impossible variant of every instance. User IDs of the designed instances are arranged to have increasing step sizes.

Looking at the graph, we see that the relative difference in computing time roughly increases with the `instance_id` i.e. with higher solution length. This also makes sense since at higher steps adding another step increases the possible state space to search, but much more than adding steps to small spaces. The dip that happens around 10 is because that is the threshold for the designed instances which start out with a low step length. It is clear that again level 0 and level 3 have the highest difference, with 3 having the highest of all. This makes sense since level 0 must traverse almost all of the search space, a single step increase can

have big consequences. Conversely, heavily optimised runs could possibly find solutions much faster when a solution is possible creating a big difference to when there is no solution and it has to search an entire space anyway, effectively wasting the optimisation time.

4 Conclusion

For this assignment, I modelled an abstracted version of the game Bombastic. In this report, I discussed the rules and how I modelled them using Savile Row. Eventually,I found that both optimisation levels 0 and 3 produced less than optimal results, with level 2 performing best across the board. word count:

Appendices

A Data collection script

```
1  import subprocess as sub
2  import os
3  import re
4
5  solutionPatern = re.compile("solution")
6  paramFilePatern = re.compile(r'.*\.param$')
7  timePatern = re.compile(r'(\d+\.\d+)')
8  fileNamePatern = re.compile(r'.*Bombastic(\d+)\_(\d+).*')
9  logFile = "compTimes.log"
10
11
12 optFlags = ['-O' + str(x) for x in range(4)]
13
14 sub.call("rm -f " + logFile + " && touch " + logFile + " && echo
       \"instance_id,steps,optLevel,foundSolution,compTime\" >> "+
       logFile,shell=True)
15
16
17 for root, dirs, files in os.walk("."):
18   for file in files:
19     filePath = os.path.join(root,file)
20     if not paramFilePatern.match(filePath) or re.search(r'stac',
          filePath):
21       continue
22     else:
23       fileSearch = fileNamePatern.search(filePath)
24                         instance_id = fileSearch.group(1)
25                         steps  = fileSearch.group(2)
26
27       for flag in optFlags:
28         lineOut = str(instance_id) + "," + str(steps) + ","
29         print(file,flag)
30         subOut = sub.check_output("TIMEFORMAT='%U'; { time ../
              savilerow Bombastic.eprime "+filePath+" -run-solver "+
```

```
            flag+"; } 2>&1",shell=True)
31          lineOut += flag[-1] + ","
32          if solutionPatern.search(subOut):
33            lineOut += "true,"
34          else:
35            lineOut += "false,"
36
37          match = timePatern.search(subOut)
38          lineOut += match.group(0)
39          sub.call("echo \"" + lineOut + "\" >> "+logFile,shell=
              True)
```