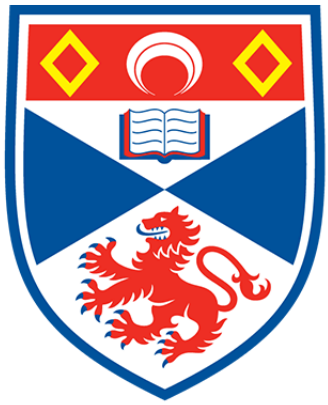

University of
St Andrews

CS4402 Constraint Programming

The Bombastic Modelling Problem

March 12, 2018

Lecturer:
Ian Miguel

Submitted By:
140011146

1 Introduction

Bombastic is a Capcom video game which involves pushing dice on a grid into certain configurations. In this practical we take an abstraction of this game, modelling it in Essence Prime using the Savile Row tool and writing constraints to the problem.

The model and constraints are then run against a set of parameter files with different processing options in Savile Row to evaluate the performance of the model and explore the effects of heuristics and optimisations of the tool.

Finally, custom problem instances were defined to investigate and identify specific aspects of the Bombastic problem that affect the efficiency of our model.

2 Design and Implementation

An initial model was created to pass all tests from the given parameter files with little thought for efficiency. Afterwards, small changes were made to try and optimise the model to improve the time taken and reduce the number of solver nodes to search.

2.1 Initial model

2.1.1 Initial and goal states

There are three sets of state variables that need to be set up as the initial states: the avatar's position, the locations of the blocks and the cells of the grid.

```
1  $ Avatar's initial position
2  avatarCurrentRow[0] = avatarInitRow,
3  avatarCurrentCol[0] = avatarInitCol,
4
5  $ Initial locations for blocks
6  forall block : int(1..numBlocks) .
7    blocksCurrentRow[0,block] = blocksInitRow[block] /\
8    blocksCurrentCol[0,block] = blocksInitCol[block],
9
10 $ Initial cells of grid
11 forall row : int(1..r) .
12   forall col : int(1..c) .
13     gridCurrent[0,row,col] = gridInit[row,col],
```

Listing 1: Constraints for setting initial state variables

This sets all `current` decision variables for step 0 based on the given `init` parameter variables. All further constraints will be based on these `current` matrices and their values. Next is the constraint for the goal state.

```
$ All blocks are in a goal
forall block : int(1..numBlocks) .
  exists goal : int(1..numBlocks) .
    blocksCurrentRow[steps,block] = blocksGoalRow[goal] /\
    blocksCurrentCol[steps,block] = blocksGoalCol[goal],
```

Listing 2: Constraints for the goal state

Because it does not matter which block is pushed into which goal, we can say that for every block there must exist a goal it is in. This combined with the constraint that blocks cannot be in the same position means each block must be in a different goal.

2.1.2 Invalid states

Next are the constraints for invalid states of the game. This restricts the model to not have states such as having the avatar and a block be in the same position.

```
$ Avatar cannot be on dead cells
forall step : int(0..steps) .
  forall row : int(1..r) .
    forall col : int(1..c) .
      gridCurrent[step,row,col] = 0 ->
        avatarCurrentRow[step] != row \/
        avatarCurrentCol[step] != col,


$ Blocks and avatar cannot share same cell
forall step : int(0..steps) .
  forall block : int(1..numBlocks) .
    avatarCurrentRow[step] != blocksCurrentRow[step,block] \/
    avatarCurrentCol[step] != blocksCurrentCol[step,block],

$ Block cannot be on dead cells
forall step : int(0..steps) .
  forall block : int(1..numBlocks) .
    forall row : int(1..r) .
      forall col : int(1..c) .
        gridCurrent[step,row,col] = 0 ->
          blocksCurrentRow[step,block] != row \/
          blocksCurrentCol[step,block] != col,

$ Blocks cannot share same cell
forall step : int(0..steps) .
  forall checkBlock : int(1..numBlocks) .
    forall otherBlock : int(1..numBlocks) .
      checkBlock != otherBlock ->
        blocksCurrentRow[step, checkBlock] != blocksCurrentRow[step,
            otherBlock] \/
```

```
31        blocksCurrentCol[step, checkBlock] != blocksCurrentCol[step,
              otherBlock],
```

Listing 3: Constraints to prevent invalid game states

All these constraints are quite similar and simply deal with not allowing the avatar or any blocks to share position or be in dead cells. An $\vee$ is used instead of an $\wedge$ to convey the constraint, though they are equivalent to one another by DeMorgan's Law:

$$\texttt{!(avatarCurrentRow[step] = row} \wedge \texttt{avatarCurrentCol[step] = col)} \tag{1}$$

$$\texttt{avatarCurrentRow[step] != row} \vee \texttt{avatarCurrentCol[step] != col} \tag{2}$$

These constraints look rather inefficient from the number of nested loops required. We shall we later in section 2.2.1 how we can improve this. Additionally, the results found in section 3.2 show the difference in performance between the initial and improved constraints.

2.1.3 Movement

We need to ensure that if the `avatarCurrentRow` and `avatarCurrentCol` have different positions in different steps (i.e the avatar has moved its position), then `moveRol` and `moveCol` must be updated. Furthermore, the movement cannot be more than one step vertically or horizontally and not diagonally and there cannot be no movement every turn.

```
1  $ Update moveRow/moveCol for avatar movement
2  forall step : int(1..steps) .
3    avatarCurrentRow[step-1] < avatarCurrentRow[step] -> moveRow[step] = 1,
4
5  forall step : int(1..steps) .
6    avatarCurrentRow[step-1] > avatarCurrentRow[step] -> moveRow[step] =
         -1,
7
8  forall step : int(1..steps) .
9    avatarCurrentRow[step-1] = avatarCurrentRow[step] -> moveRow[step] = 0,
10
11 forall step : int(1..steps) .
12   avatarCurrentCol[step-1] < avatarCurrentCol[step] -> moveCol[step] = 1,
13
14 forall step : int(1..steps) .
15   avatarCurrentCol[step-1] > avatarCurrentCol[step] -> moveCol[step] =
         -1,
16
17 forall step : int(1..steps) .
18   avatarCurrentCol[step-1] = avatarCurrentCol[step] -> moveCol[step] = 0,
```

Listing 4: Updating `moveRow` and `moveCol`

These initial constraints on both `avatarCurrentRow` and `avatarCurrentCol` checks all the cases for 1, -1 and 0 depending on whether or not the avatar's position on the next step is greater, less or equal respectively. For example if the avatar's previous position is less than the avatar's current position, then the avatar must have moved a step to the right, so `moveRow` for that step is 1.

```
$ Diagonal movement not allowed and must move each turn
forall step : int(1..steps) .
  | moveRow[step] | + | moveCol[step] | = 1,
```

Listing 5: Prevent diagonal movement and force movement every turn

This second constraint restricts both diagonal movement and forces the avatar to move every turn. This works because the absolute value of `moveRow` and `moveCol` is how much the avatar has moved by. To move diagonally, the sum of `moveRow` and `moveCol` must be at least 2, as one has to move at least one row *and* column. Additionally, the avatar must move every turn with this constraint as the sum is equal to 1, so `moveRow` and `moveCol` cannot both be 0 on each turn. This also constrains the avatar to only move a distance of 1 each turn.

Next, the blocks must be pushed by the avatar must be moved. To do this, two constraints were used.

```
$ If block has moved, avatar must have moved into block's previous
    location
forall step : int(1..steps) .
  forall block : int(1..numBlocks) .
    blocksCurrentRow[step-1,block] != blocksCurrentRow[step,block] \/
    blocksCurrentCol[step-1,block] != blocksCurrentCol[step,block] ->
      avatarCurrentRow[step] = blocksCurrentRow[step-1,block] /\
      avatarCurrentCol[step] = blocksCurrentCol[step-1,block],

$ If avatar moved into block, block move same direction
forall step : int(1..steps) .
  forall block : int(1..numBlocks) .
    avatarCurrentRow[step] = blocksCurrentRow[step-1,block] /\
    avatarCurrentCol[step] = blocksCurrentCol[step-1,block] ->
      blocksCurrentRow[step,block] = blocksCurrentRow[step-1,block] +
          moveRow[step] /\
      blocksCurrentCol[step,block] = blocksCurrentCol[step-1,block] +
          moveCol[step],
```

Listing 6: Constraints for moving blocks

The first checks if a block has moved on the next step. If the block has moved, then the avatar must have moved into the block's old position as that is the only way blocks can move. However, just this constraint is not enough as it doesn't say anything about how to move the block. The second constraint

moves the block by adding `moveRow` and `moveCol` to `blocksCurrentRow` and `blocksCurrentCol` respectively. This works as `moveRow` and `moveCol` directly represent the direction of the avatar's movement and blocks must be pushed in the same direction. We do not have to worry about pushing blocks into dead cells as previous constraints do not allow that to happen.

2.1.4 Grid and ice

Finally, we have to make sure than none of the grid changes unless it is ice and it was stepped on.

```
$ Grid 0 and 2s always stay the same
forall step : int(1..steps) .
  forall row : int(1..r) .
    forall col : int(1..c) .
      gridCurrent[step-1,row,col] != 1 ->
        gridCurrent[step,row,col] = gridCurrent[step-1,row,col],

$ Ice becomes dead cell
forall step : int(1..steps) .
  forall row : int(1..r) .
    forall col : int(1..c) .
      avatarCurrentRow[step-1] = row /\
      avatarCurrentCol[step-1] = col /\
      gridCurrent[step-1,row,col] = 1 ->
        gridCurrent[step,row,col] = 0,
```

Listing 7: Constraints for grid cells

The first constraint here ensures 0s and 2s never change as live cells and dead cells always stay the same. For ice to change, we simply check if the cell is ice on the previous step and the avatar was stepping on it. Then in the next turn the ice cells turns into a dead cell. We do not have to worry about the avatar not moving as multiple other constraints prevent this from happening. The avatar must move each turn due to the movement constraints and cannot be allowed on dead cells from the invalid state constraints. However, these constraints are not enough, as the ice cells can still freely change on every turn.

```
$Ice not stepped on doesn't change
forall step : int(1..steps) .
  forall row : int(1..r) .
    forall col : int(1..c) .
      gridCurrent[step-1,row,col] = 1 /\
      (avatarCurrentRow[step-1] != row \/ avatarCurrentCol[step-1] !=
          col) ->
        gridCurrent[step,row,col] = 1
```

Listing 8: Additional constraint to prevent ice cells from changing

This final constraint makes sure ice stays the same. This works by having the grid of the previous step be an ice cell (`gridCurrent[step-1,row,col] = 1`) and making sure the avatar must not have stepped on the cell in the previous turn. Then the ice cells must stay the same.

2.2 Model improvements

After some testing and initial results, small improvements and simplifications were made to the original model that substantially reduced the number of solver nodes and time taken to run the various problem instances.

2.2.1 Direct row/col

In a few cases, the constraints in the initial model had to loop through all steps, rows and columns, for example to check the avatar is not on a dead cell. This could be simplified to directly use the avatar's current row and column as an index rather than check every combination of step, row and col.

```
$ Avatar current row/col cannot be on dead cells
forall step : int(0..steps) .
  forall row : int(1..r) .
    forall col : int(1..c) .
      gridCurrent[step,row,col] = 0 ->
        avatarCurrentRow[step] != row \/
        avatarCurrentCol[step] != col,

$ Optimisation
forall step : int (0..steps) .
  gridCurrent[step, avatarCurrentRow[step], avatarCurrentCol[step]] != 0,
```

Listing 9: Example of optimising number of constraints by directly indexing with `avatarCurrentRow` and `avatarCurrentCol`.

This reduces the number of constraints significantly, as every case of this pattern created additional $row \times col$ constraints. Other constraints of this pattern, such as checking a block on a dead cell were all changed the same way.

2.2.2 Movement

The initial movement constraints for updating `moveRow` and `moveCol` aren't very efficient because a separate constraint had to be written for each case. This was simplified as we can notice that the value of `moveRow` and `moveCol` is exactly directly related to the `avatarCurrentRow` and `avatarCurrentCol`.

```
$Update moveRow/moveCol for avatar movement
forall step : int(1..steps) .
    moveRow[step] = avatarCurrentRow[step] - avatarCurrentRow[step-1] /\
    moveCol[step] = avatarCurrentCol[step] - avatarCurrentCol[step-1],
```

Listing 10: Refactored and improved movement constraint

This simplification works as the game grid is indexed in order both row-wise and column-wise. To explain easily, we can rearrange the formula to be as follows:

```
1 avatarCurrentRow[step] = moveRow[step] + avatarCurrentRow[step-1]
2 avatarCurrentCol[step] = moveCol[step] + avatarCurrentCol[step-1]
```

This intuitively says that the avatar's current position is its previous position plus the value of its movement. This works the same for a negative value as that is just moving in the other direction.

3 Experimentation

The constraint models were tested on the given parameter files and the results of the `.info` files taken. Except for the very first parameter file, the parameter files come in pairs of the same "problem". We will refer to these as the same problem as the grid, blocks and goals are all the same, with the only difference between the instances being the number of steps the avatar can take. The first of each pair always has not enough steps to find a solution while the second of the pair does. For example, files `2_2` and `2_3` are the same problem as they are identical except for the number of steps. This feature will be important to remember for our analysis.

The results of running the instances in Savile Row are displayed and analysed to test differences in optimisations flags, heuristic options and how the efficiency of the models and the characteristics of problem instances relate to each other.

New custom instances were also created and tested. These problem instances each have their own properties which were designed to test specific elements of the model.

3.1 Methodology

For the experiments with optimisation and heuristic options in Savile Row, both the initial model and improved model were used. This lets us see if any options can improve the efficiency of the models without manually refining the constraints and also shows the effectiveness of the optimisations and heuristics on the different models.

The time taken by the solver is not the only metric to look at, as it is not the same on every run and dependent on the load and speed of the computer running the experiment. The number of solver nodes will also be used as a metric of how well the models perform since this number should be deterministic for the

same model and problem instance.

For the custom instances, only the improved model was used as we want to test the elements that affect the efficiency of the model and not test the model itself.

To run the experiments, a simple python script was written to run Savile Row automatically with various options and the results gathered. The two fields `SolverTotalTime` and `SolverNodes` from the `.info` file are used as the time taken and number of solver nodes respectively.

3.2 Results

From running both models against all given parameter files, we can see a simple pattern which applies to both the initial model and the improved one. As the problem instances become more complicated, the number of solver nodes and the time taken both increase together. The improved model performs much better, taking less time and less nodes, but the pattern is the same.

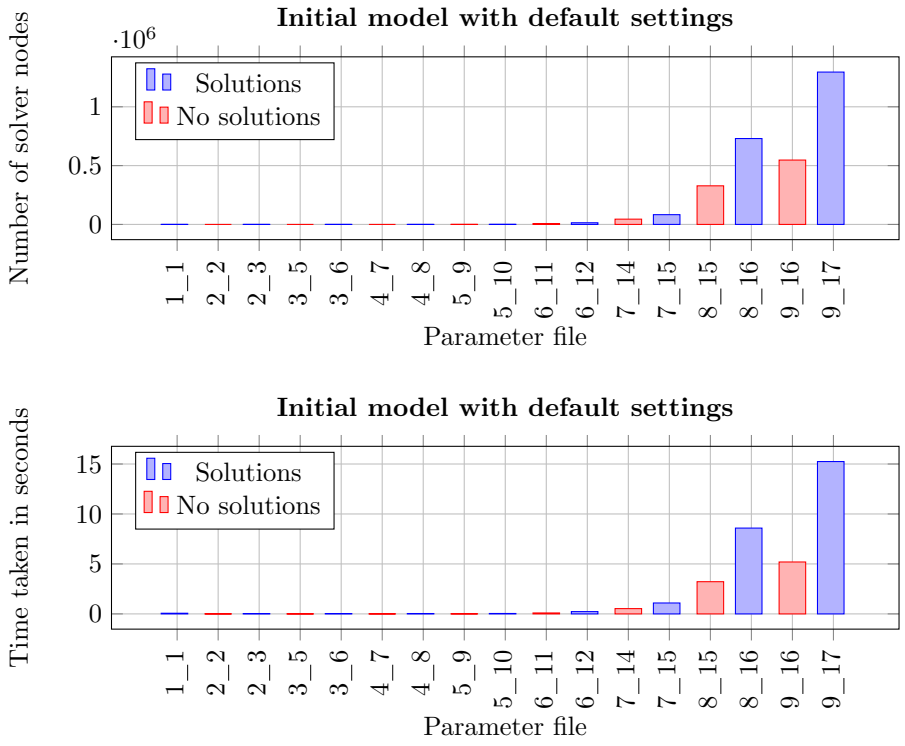

Figure 1: Number of solver nodes and time taken on all given parameters for the initial model. The two different coloured bars represent parameters which either had a solution, or the solver found no solutions.

At first glance, it appears that the instances with solutions always take more time and nodes than an instance of the same problem with no solution. This is not intuitive as logically having no solution means the solver must search through all nodes before stopping while having a solution means a solver can stop on the first solution found. However, it is unfair to make this comparison because the instance with solutions always contain one more step compared to their counterparts. We shall revisit this issue later by defining instances with multiple solutions and instances with more steps.

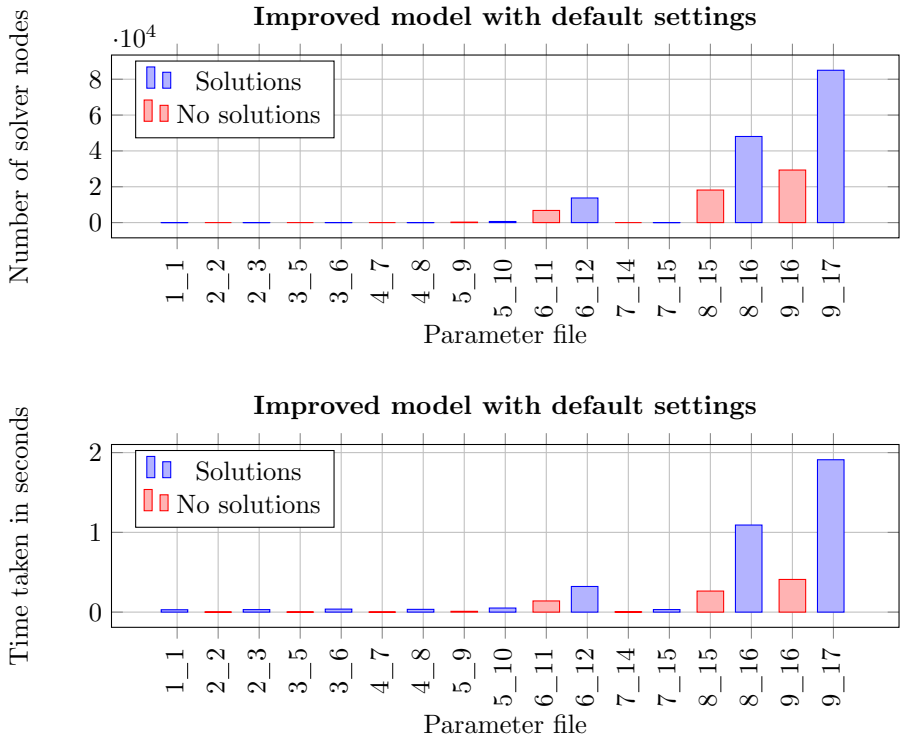

Figure 2: Number of solver nodes and time taken on all given parameters for the improved model

The improved model follows almost the same pattern as initial model, but the number of solver nodes and time taken is decreased significantly. For example the instance `9_17` took over 15 seconds and 1 million solver nodes for the initial model while the improved model only took 2 seconds and 85000 nodes. This indicates the improved constraints help reduce the effort required by the solver by being more precise. Interestingly, problem 7 is easier for the improved model to solve and is an anomaly to the pattern of increasing time and number of nodes.

3.2.1 Optimisations

To find out more about Savile Row and our models, we can specify the level of optimisation.

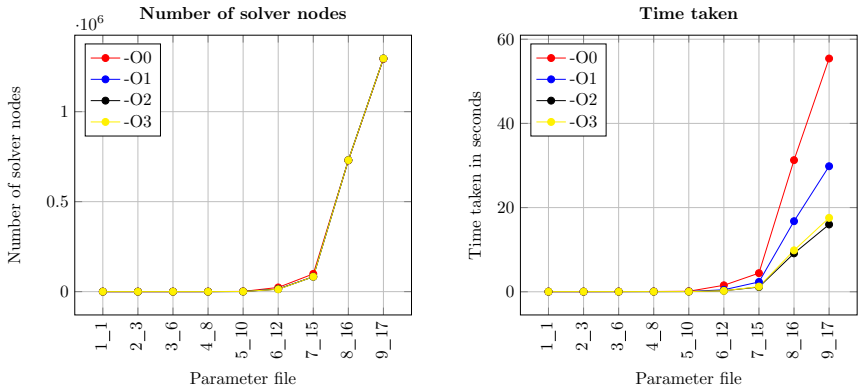

Figure 3: Comparing the effect of optimisation flags on the number of nodes and time taken on the initial model for parameter files with solutions. See appendix A for full results.

An interesting result from using different optimisation flags in is the effect it has on the performance of the model. Although it was shown earlier that the number of solver nodes and the time taken by the solver are very similar metrics for measuring the performance of the model, the optimisation flags have a much greater effect on the time taken compared to the number of nodes the solver has to search. With optimisations, the number of solver nodes only decreased by a small fraction. For many parameter files, this number did not change between the three levels of optimisation. However for the time taken by the solver, the time almost halved for most parameter files when comparing `-O0` and `-O1`. The time again decreased significantly for `-O2`, though there is little difference between `-O2` and `-O3`.

This shows us that the optimisation does less to cull search nodes, but perhaps more to reduce processing time in generating the nodes or does more to be able to search through the nodes efficiently, especially as the problem sizes and complexity grows (size and complexity here refer to larger grids and more complicated grid setups like having ice blocks or multiple blocks to push).

That is not to say the optimisations have no effect at all on the solver nodes. Figure 4 shows that for some parameter files, the optimisation does well enough to eliminate many solver nodes. For example in problems 4 and 7, the `-O1` optimisation is able to eliminate all but 1 solver node. From the difference shown between the initial model and improved model, it can be seen that the optimisations have a stronger effect on the better model for lowering the number

of solver nodes. This shows it is important to have a well defined model as optimisations are not enough to overcome an inefficient model and are also more effective for good models.

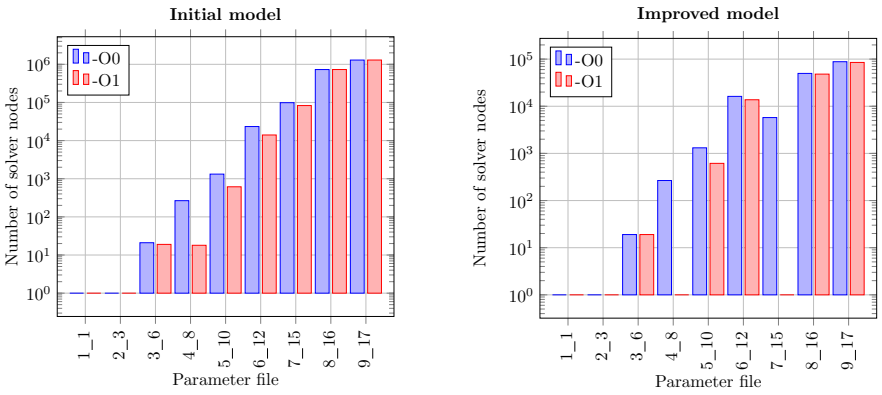

Figure 4: Effect of optimisation flags on number of solver nodes for parameter files with solutions.

3.2.2 Heuristics

Another option to look at is the heuristics that Essence Prime provides. As these are heuristics that must be specified in the model (in `Bombastic.eprime`), we are interested to see how they may affect the number of solver nodes searched by the model. Many of the problem instances follow a similar pattern for how they are affected by different heuristics. We shall focus our attention on the `9_17` problem instance as it is the most difficult problem - takes the longest time and most solver nodes - and allows us to find patterns without having too complex a graph.

From figure 5, the most interesting heuristics to take note of are `sdf` and `srf`. When using the `sdf` heuristic, we can see there is a large difference for its effect on the initial model and improved model. With optimisation flags, the heuristic causes the model to perform *worse* than without any optimisation for the initial model, but it performs well for the improved model only with optimisations on. It is impossible to say without knowing what the heuristics do how the models are actually affected, but the more concise constraints of the improved model are likely part of the cause.

In the case of `srf`, we again have some very interesting results. The effect of using `srf` combined with optimisations seem to have almost an opposite effect on the two models. `O1` optimisation performs worse than `O0` and `O2` and `O3` perform the best for the initial model. This directly contrasts the results for the improved model, where `O1` performs the best, `O0` the worst, but turning on higher optimisations (`O2` and `O3`) increases the number of solver nodes compared to `O1`.

There may be some work that the heuristic does that additional optimisations do not take advantage of or their combination conflicts with each other. Of course, even though `srt` performs better on higher optimisations for the initial model, the number of nodes is still greater.

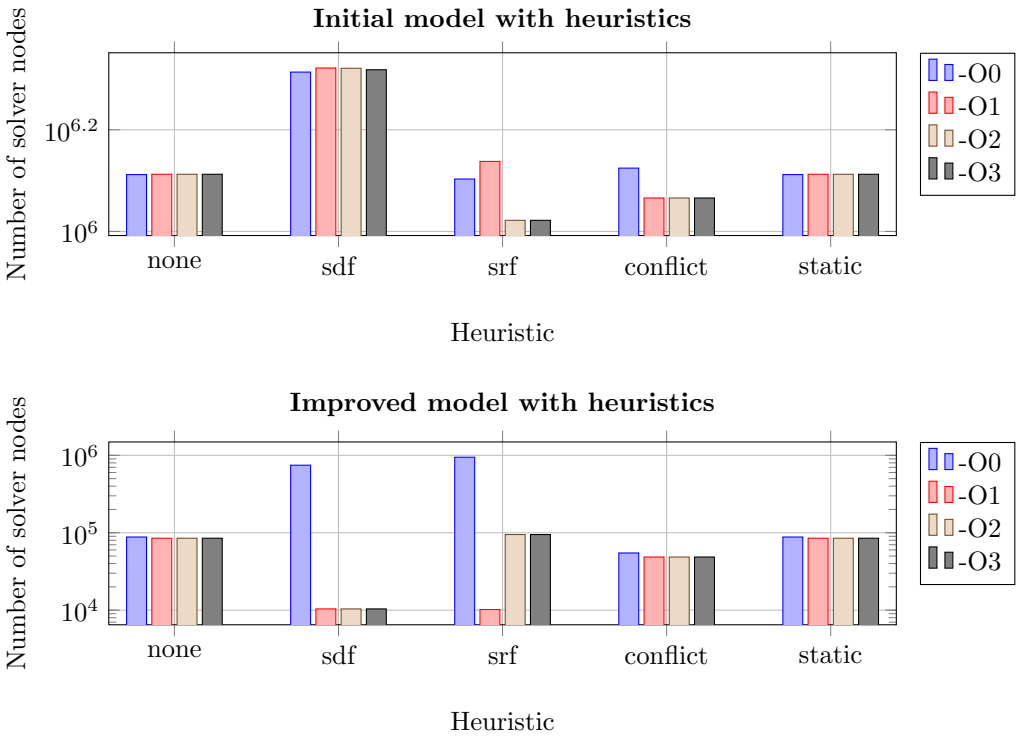

Figure 5: Effect of different heuristics on problem instance `9_17`

It is important to note that these results and patterns are not unique to the `9_17` instance and recurs in many of the given instances. Furthermore, it should also be noted that in the case of `sdf` and `srf`, using the heuristics without optimisations performs worse than not using the heuristics at all. This suggests some kind of relationship between how the heuristics and optimisations complement or contrast each other. Perhaps the heuristics create a lot of equivalent or symmetrical nodes that optimisations can easily simplify, but over-complicates the search if not simplified.

3.3 Custom instances

To look into more detail at how the effort required of the models are affected, some of the given instances have been modified and tested against the improved model. This allows us to more finely test what affects the efficiency of the model and isolate specific variables to change and test.

3.3.1 Increasing number of steps

First, a simple thing we can change is the number of steps for problems we looked at earlier. This gives us a comparison against earlier results. The number of steps is increased and not decreased as decreasing the number of steps will only lead to having no solution due to the nature of the game. We test this for both problems with solutions after increasing the number of steps, and problems with no solutions after. Examples of the latter would be grids that contain ice such that the avatar cannot move after pushing all blocks into goals, thereby causing no solution to be found. We use default settings here only on the improved model as we want to focus our attention on the effect of the parameter files and not on other options explored earlier.

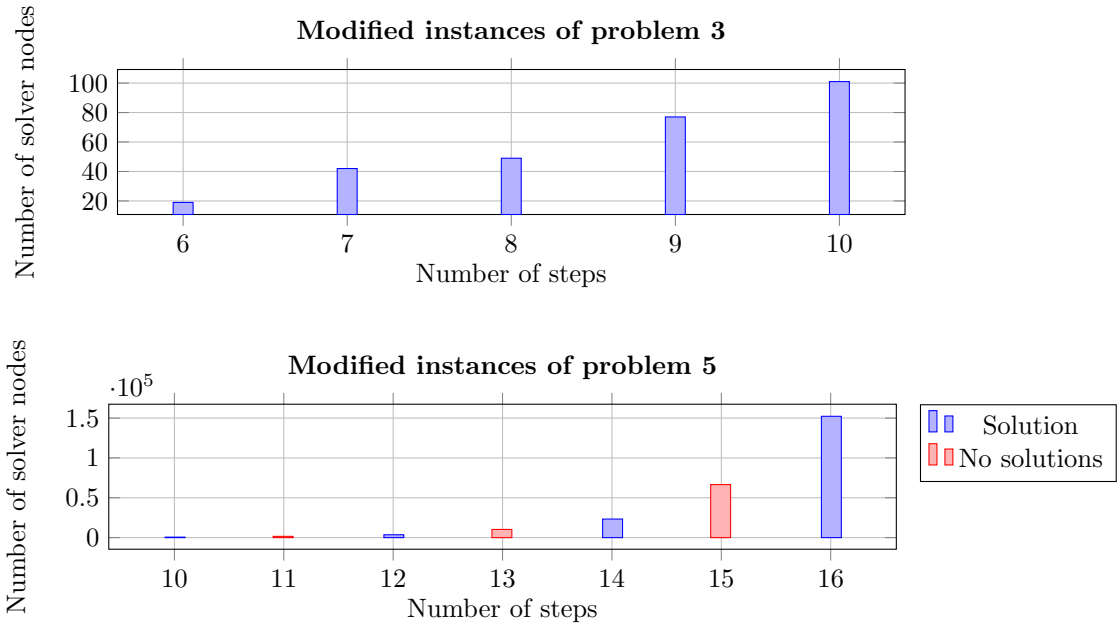

Figure 6: Number of nodes on the same problems but with varying number of steps required. Note that for problem 5, the number of steps determines whether there is a solution or not as the avatar can move back and forth along his initial position with two extra steps, but not with just one extra step. For problem 3, all instances have solutions as the avatar can move freely after pushing all blocks into the goal.

It is interesting to see how the number of solver nodes increases with the number of steps even though there is still just one or no solution. This suggests that the number of steps is an independent factor in regards to the difficulty of a problem instance. It also doesn't seem to matter whether the increased steps led to a solution or not as the number of nodes increases regardless. Sequences

of movement after the blocks are pushed into the goals for problem 3 should not increase the number of solver nodes, since any sequence of movement after the blocks are pushed is a valid solution and the solver would stop. This means the nodes must come from large numbers of backtracking combinations before the block is pushed into the goal that are tried because having more steps increased the number of combinations. The increased work from a larger grid may also dwarf the work difference between problems with and without a solution. It is difficult to test the same problem instance both with and without a solution since the problem would be a different instance by definition, but we can certainly say the number of steps is a large factor in the difficulty of the Bombastic problem.

3.3.2 Increasing cube problems

Something that we want to test is how a problem with multiple solutions would affect our model, especially when the `-all-solutions` flag is passed into Savile Row. A custom instance that was created and tested for this purpose was an $n \times n$ cube where the avatar always started at the top left, the goal is always at the bottom right and the one single block is always on the cell directly top left of the goal. The $n \times n$ here is the size of the live cells in the problem grid, not taking into account the ring of dead cells.

0	0	0	0	0
0	**P**	2	2	0
0	2	**B**	2	0
0	2	2	**G**	0
0	0	0	0	0

0	0	0	0	0	0	0	0
0	**P**	2	2	2	2	2	0
0	2	2	2	2	2	2	0
0	2	2	2	2	2	2	0
0	2	2	2	2	2	2	0
0	2	2	2	2	**B**	2	0
0	2	2	2	2	2	**G**	0
0	0	0	0	0	0	0	0

Figure 7: Examples of a 3×3 and 6×6 cube problem instance on the left and right respectively.

Cube problem sizes from 3×3 to 9×9 were created and run with the improved model. This problem is used to test two different aspects of the model and solver. First is how the size of the grid affects the time and number of nodes to find a solution. This is different from the initial testing with the given parameter files as there is no increased complexity in the problem instance from elements such as ice blocks and multiple blocks. Only the size of the grid changes between cube problem instances. Secondly we can test the effect of multiple solutions to confirm the solver takes more time and nodes as there are more solutions to a problem instance.

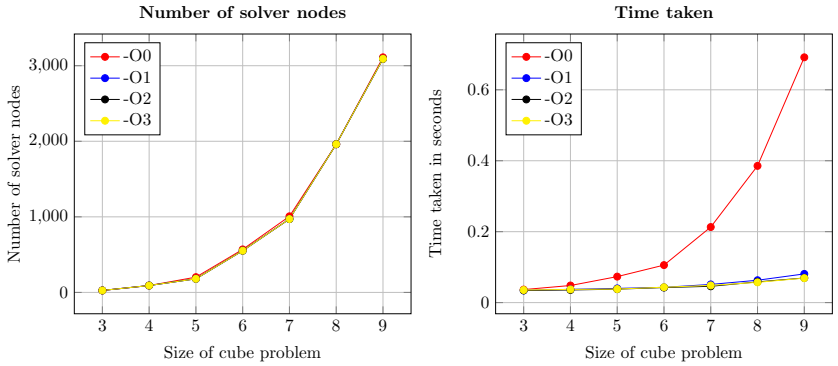

Figure 8: Results of running all cube problems without the `-all-solutions` flag.

We can see that as the size of the grid increases, the number of solver nodes searched increases exponentially. This makes sense as the size of the grid and number of steps has increased, there is a larger search space with more possible moves to search through. What is interesting is that the time taken is significantly improved by the optimisations while the number of solver nodes is not. This is similar to the pattern observed in section 3.2.1 but to a greater effect.

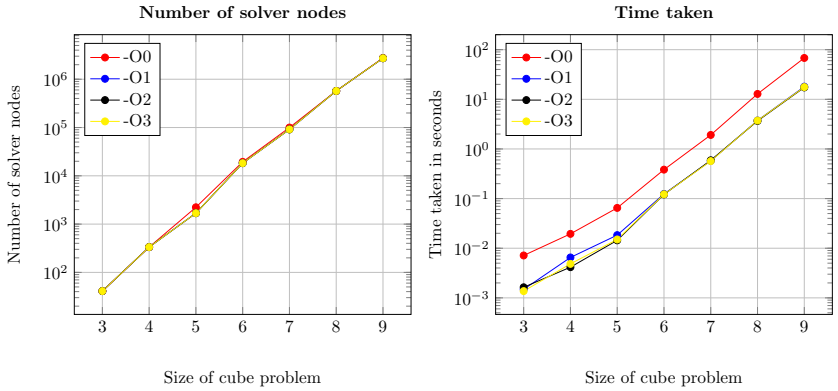

Figure 9: Results of running all cube problems finding all solutions. The y-axis is plot as a log axis to more clearly see the relation of the data.

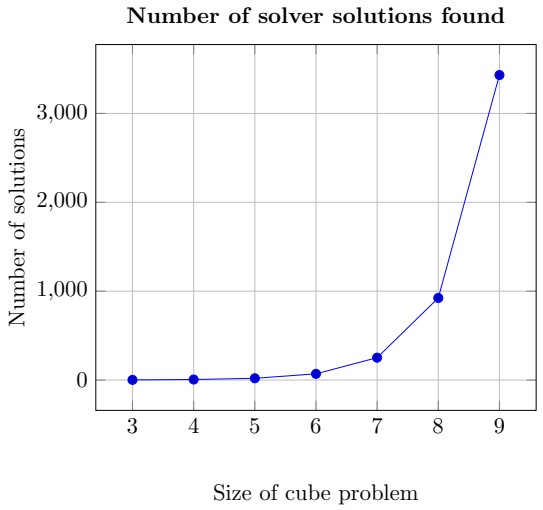

Figure 10: Number of solutions found for each cube problem instance.

Another good result we get is time and nodes searched to find all solutions is much more than just finding one solution. Though this should be obvious, it is a good result to show empirically, showing that the constraint solver will stop at the first solution found without any further searching by default.

Again the same pattern of improving time with optimisations without improving the number of nodes applies when finding all solutions as well. The log graph of figure 9 quite clearly shows no optimisations is worse for every sized cube problem instance. However, it is difficult to determine if this pattern is due solely to the increasing complexity from the size of the grid, as the number of steps had to be increased as well and it is already shown that the number of steps is an important factor where more steps requires more effort to solve.

3.3.3 Cube with ice

To determine whether the number of steps is the sole factor determining the effort required, we can change our cube problems to contain ice cells instead of normal cells. The number of steps stays constant for each cube problem, but all normal cells are replaced with ice cells. This does not affect the solution as all solutions with the given number of steps do not require walking back over any previously visited cell. In other words, it is valid to change all cells to ice without losing any solutions.

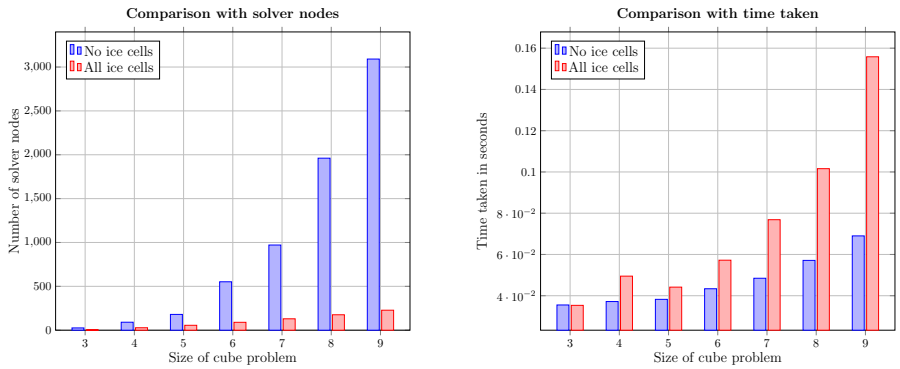

Figure 11: Comparison of the same cube problems with same steps but cells are either all normal or all ice.

Having the problem instances change to contain all ice cells dramatically decreases the number of solver nodes searched. The rest of the instance stays exactly the same, including the number of steps. This is interesting as it shows the ice creates less nodes to search through as it prevents any nodes with backtracking because it is not allowed to walk back from the ice. However, the time taken tells an opposite story. The problem instances with ice take *more* time to solve even though there are many less nodes being searched. This shows the effort by the solver isn't just on searching through the nodes to find a solution, but also includes processing time to go through the constraints to get the nodes in the first place. Having all ice cells complicates the instance in that every move causes changes to the grid and therefore required more time.

From this, we can also say that the number of steps is not the only factor. The complexity of the instance changes the effort taken by the solver and model. A more complicated problem contains less search because it is more constrained by the model, but still takes time and effort as more constraints have to be processed. More steps does seem to be a large contributing factor, as figure 11 still shows a larger grid - which requires more steps to solve - takes more work, but there are other factors like the ice cells which can change the efficiency in which the model can find solutions.

4 Conclusion

In conclusion, two models with different constraints were developed and tested against various Bombastic problem instances. The second model attempted to improve on the first by simplifying complicated constraints to be more precise. The differences between the models are shown by their performance on the given problem instances. Additionally, the solver options of Savile Row such as optimisation flags and heuristics were also explored to see their effect on the two models. Finally, custom instances were created to isolate and test how

specific elements increased the difficulty of the Bombastic problem and affected the performance of our constraints model.

The optimisation flags and heuristics both had strong effects on the model, able to eliminate solver nodes and decrease the time taken by the model. The two seem to be able to work in tandem, producing good results if both are used. Nevertheless, using heuristics blindly without understanding how they work can have a negative impact on performance, as some combinations of heuristics and optimisations can lead to decreased performance.

Furthermore, it was found that the number of steps had a large impact on the effort required to solve the problem, as it led to more solver nodes and more time taken, even if the rest of the problem instance stays the same. It was also shown that there are other factors such as complexity of the instance that contribute to different efforts required to solve the problem.

Appendices

Appendix A Results of optimisation flags

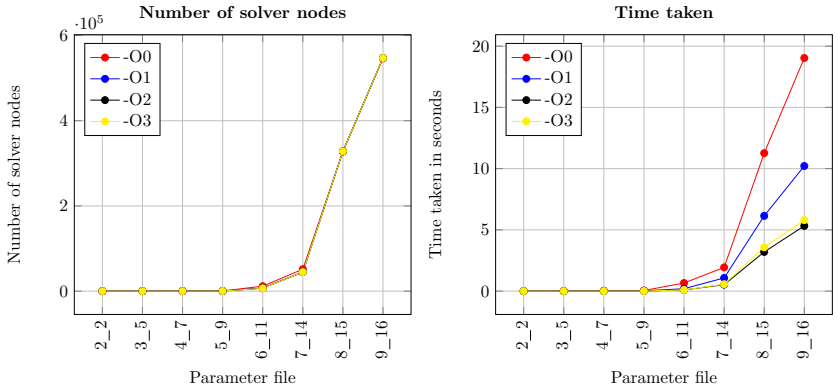

Figure A.1: Optimisations with initial model on problem files *without* solutions

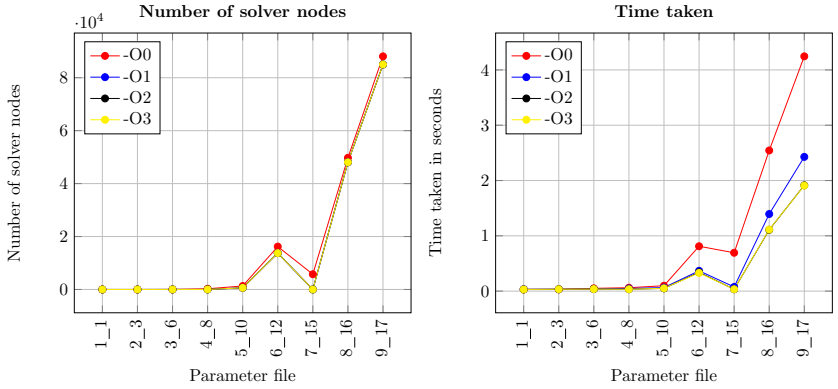

Figure A.2: Optimisations with improved model on problem files *with* solutions

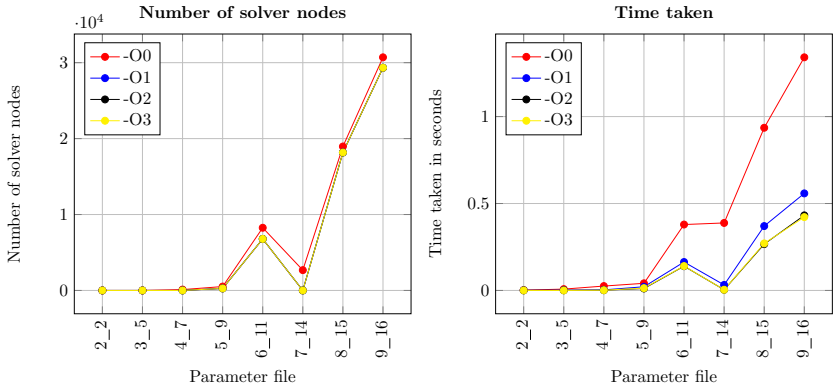

Figure A.3: Optimisations with improved model on problem files *without* solutions