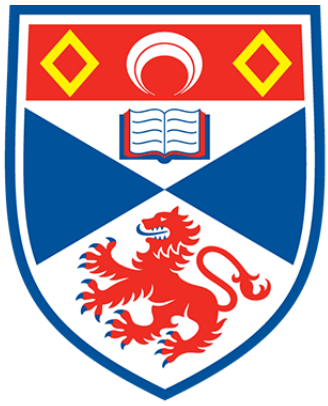

University of
St Andrews

CS4402 Constraint Programming

The Bombastic Modelling Problem

March 2, 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.

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 and 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.

```
1  $ All blocks are in a goal
2  forall block : int(1..numBlocks) .
3      exists goal : int(1..numBlocks) .
```

```
4         blocksCurrentRow[steps,block] = blocksGoalRow[goal] /\
5         blocksCurrentCol[steps,block] = blocksGoalCol[goal],
```

Listing 2: Constraints for the goal state

Because it does not matter which blocks 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.

```
1  $ Avatar current row/col cannot be on dead cells
2  forall step : int(0..steps) .
3      forall row : int(1..r) .
4          forall col : int(1..c) .
5              gridCurrent[step,row,col] = 0
6                  -> avatarCurrentRow[step] != row \/
7                     avatarCurrentCol[step] != col,
8
9
10 $ Blocks and avatar cannot share same cell
11 forall step : int(0..steps) .
12     forall block : int(1..numBlocks) .
13         avatarCurrentRow[step] != blocksCurrentRow[step,block] \/
14         avatarCurrentCol[step] != blocksCurrentCol[step,block],
15
16 $ Block cannot be on dead cells
17 forall step : int(0..steps) .
18     forall block : int(1..numBlocks) .
19         forall row : int(1..r) .
20         forall col : int(1..c) .
21             gridCurrent[step,row,col] = 0
22                 -> blocksCurrentRow[step,block] != row \/
23                    blocksCurrentCol[step,block] != col,
24
25 $ Blocks cannot share same cell
26 forall step : int(0..steps) .
27     forall checkBlock : int(1..numBlocks) .
28         forall otherBlock : int(1..numBlocks) .
29             checkBlock != otherBlock ->
30                 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 $\wedge$because TODO

2.1.3 Movement

Now for 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       moveRow[step] = avatarCurrentRow[step] - avatarCurrentRow[step-1]
            /\
4       moveCol[step] = avatarCurrentCol[step] - avatarCurrentCol[step-1],
```

Listing 4: Updating `moveRow` and `moveCol`

Updating `moveRow` and `moveCol` 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 of `moveRow`/`moveCol` as that is just moving in the other direction.

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

Listing 5: Prevent diagonal movement and force movement every turn

The 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 /\
```

```
14              gridCurrent[step-1,row,col] = 1 ->
15                  gridCurrent[step,row,col] = 0,
```

Listing 7: Constraints for grid cells

The first constraint here ensures 0s and 2s never change, however this doesn't take into account the 1s that need to stay the same as long as the avatar has not stepped on them yet.

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

Listing 8: Additional constraint to prevent ice cells from changing

For this constraint, in addition to the grid of the previous step having to be an ice cell (`gridCurrent[step-1,row,col] = 1`), the avatar must also not have stepped on the cell in the previous turn. Only then should the ice cells stay the same.

2.2 Model improvements

After some testing and initial results, improvements to the original model were made that substantially reduced the number of solver nodes.

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.

```
1  $ Avatar current row/col cannot be on dead cells
2  forall step : int(0..steps) .
3      forall row : int(1..r) .
4          forall col : int(1..c) .
5              gridCurrent[step,row,col] = 0 ->
6                  avatarCurrentRow[step] != row \/
7                  avatarCurrentCol[step] != col,
8
9  $ Optimisation
10 forall step : int (0..steps) .
11     gridCurrent[step, avatarCurrentRow[step], avatarCurrentCol[step]] !=
           0,
```

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

Other constraints of this pattern, such as checking a block on a dead cell were all changed the same way.

3 Experimentation

The constraint models were tested on the given parameter files and the results of the `.info` files taken. The results are then displayed and analysed to test differences in optimisation and heuristic options in Savile Row. Finally, new instances were created and tested TODO

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 in which the models find solutions without manually refining the constraints.

The time taken by the solver is not the best 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 will always stay the same for the same model and problem instance.

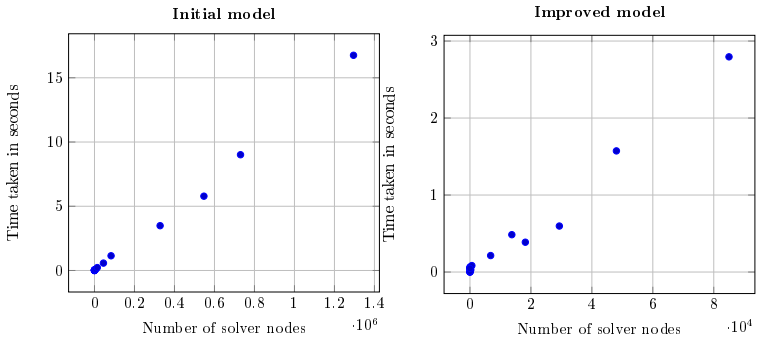

Figure 1: Number of solver nodes plotted against time for both initial and improved models. We can see a strong linear relationship between the time taken and number of solver nodes. This shows both are equivalent metrics for measuring the performance of our model.

To run the experiments, a simple python script was written to run Savile Row automatically with various options and the results gathered.

3.2 Results

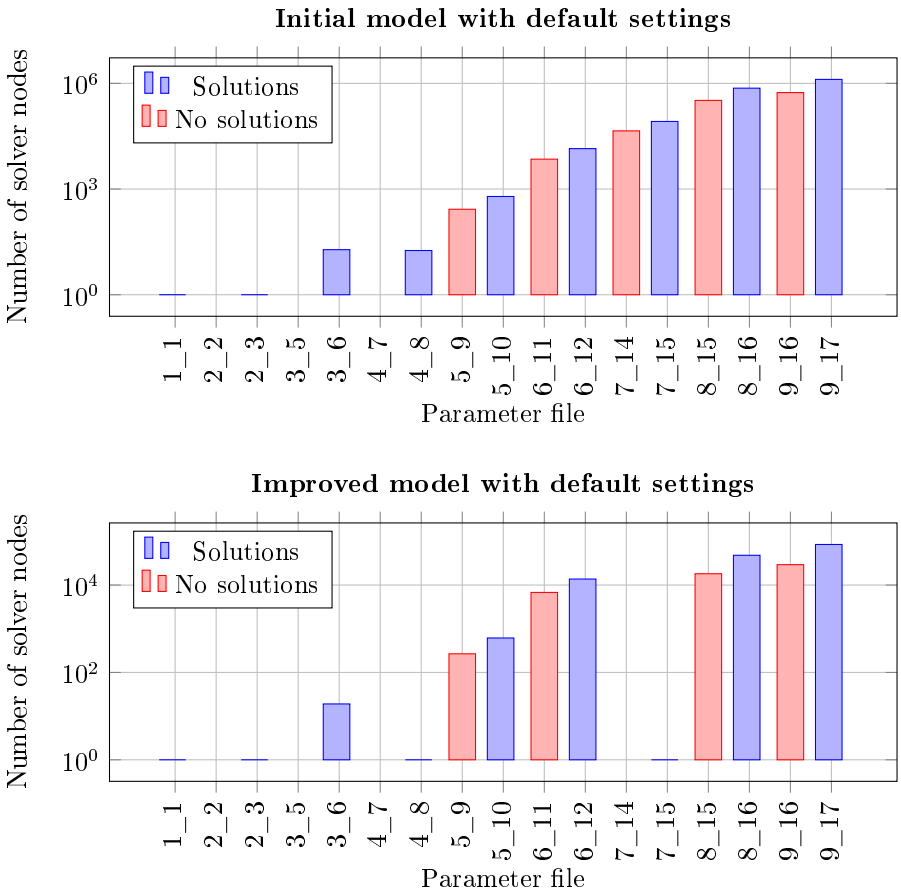

Figure 2: Time taken and number of solver nodes for all given parameters

3.2.1 Optimisations

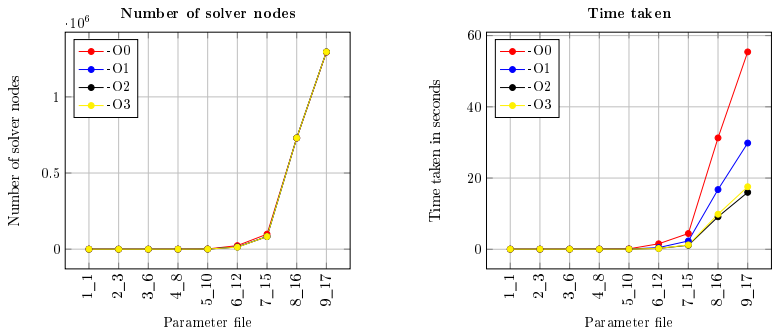

Figure 3: Comparing the effect of optimisation flags on the number of nodes and time taken on the initial model for parameter files with solutions. See appendix TODO for full results.

An interesting result from using different optimisation flags in Savile Row 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 equivalent 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`.

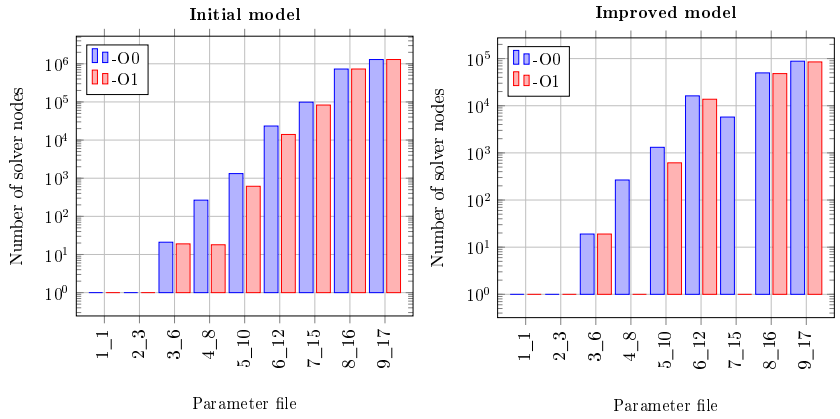

Figure 4: Effect of optimisation flags on number of solver nodes for parameter files with solutions.

This shows that parts of the optimisation is optimising Savile Row like a program, similar to how optimisation flags work in compilers like `gcc` rather than optimising it as a constraint solver. These optimisations would work well for most general programs, speeding up the run time but does little to improve our model from a constraint programming point of view. That is not to say the optimisations have no effect at all on the constraints. Figure 4 shows that for some parameter files, the optimisation does well enough to eliminate many solver nodes. 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.

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`), they should only affect the number of solver nodes and not the time taken by Savile Row.

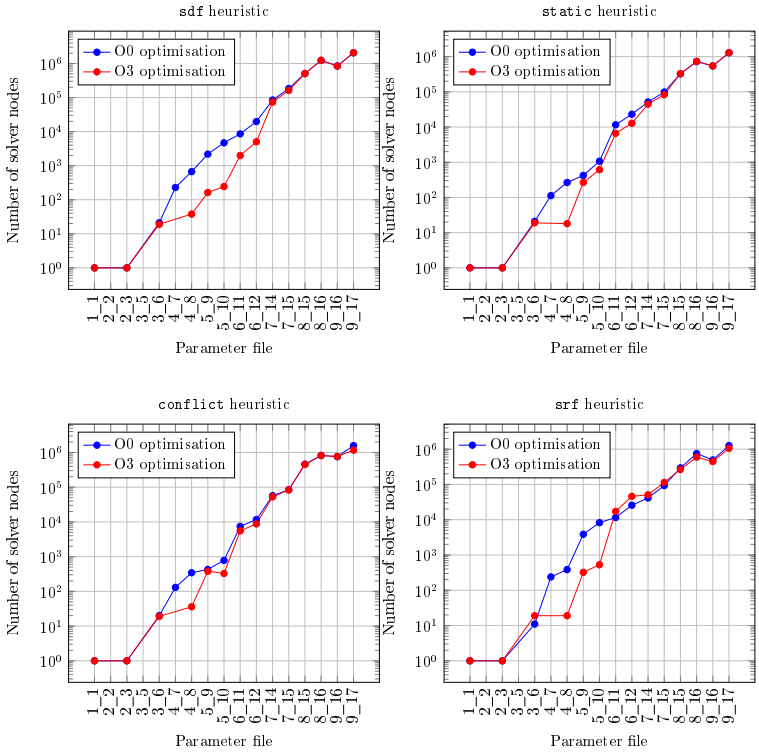

Figure 5: Number of solver nodes for different heuristics