Human Resource Machine

200024105
CS4402 P1

October 2020

1 Introduction

This report will begin with an in depth analysis of an Essence' constraint programme (hrm1.eprime) for the human resource game with an evaluation of the variables, domains and constraints. It will then use empirically evaluate the performance of the programme. The report will then compare this optimized version to a more basic version (hrm0.eprime) to demonstrate the effect of optimization constraints. For this planning problem I will use a state and action focused approach. I will use 2D matrices to track the states of decision variables over time. 2D matrices are required because at each step the decision variables have a length dimension. I believe that the state and action approach will be simpler for handling the 2D matrices as opposed to the state only approach seen in the waterbucket model.

2 Variables and Domains

For this programme I used the following parameters (see lines 4 - 7 fig1) all of which are taken from the parameter file:

- `MAX_OPS` - dictates the maximum number of operations. This is implemented through the `STEPS` domain.
- `num_registers` - dictates the number of available floor registers. This is implemented in the `REG_NO` domain.
- `input` - matrix which represents the initial input from the game conveyor. It is treated as an ordered stack to the first item must be removed before the second and so on. Maximum integer allowed in game is 100. `input` is set to equal the first `inbox` step.
- `output` - matrix which represents the target output on the game conveyor. It is also treated as an ordered stack to the first item must be placed before the second and so on. `output` is the target final `outbox`. Follow that maximum integer allowed is 100.

For this programme I used the following domains (see fig1):

- `STEPS` - limited by the number of operations `MAX_OPS` determined in the parameter file. Used to construct `action`, `inbox`, `hand` and `reg`.
- `STEPS1` - subset of `STEPS`, is iterated through by the main set of constraints. The 1st, 2nd and penultimate steps are done manually to improve efficiency.
- `IN_LEN` - for the length of the input list. Used in matrix list searches later in the programme and the construction of the inbox matrix.

```
4  given MAX_OPS: int(1..)
5  given num_registers : int(0..)
6  given input: matrix indexed by [int(1..input_length)] of int(0..100)
7  given output: matrix indexed by [int(1..output_length)] of int(0..100)
8
9  letting STEPS be domain int(0..MAX_OPS)
10 letting STEPS1 be domain int(2..(MAX_OPS-1))$first 2 and final steps are automated.
11
12 letting IN_LEN be domain int(1..input_length)
13 letting OUT_LEN be domain int(1..output_length)
14 letting REG_NO be domain int(1..num_registers)
15
16 $different integer value for each action at each step.
17 find action: matrix indexed by [int(1..MAX_OPS)] of int(1..8)
18 $Ordered and represents items in the inbox at each step.
19 find inbox : matrix indexed by [STEPS,  IN_LEN] of int(-1..100)
20 $Ordered and represents items in the outbox at each step.
21 find outbox: matrix indexed by [ STEPS, OUT_LEN ] of int(-1..100)
22 $Unordered an represents items in register slots at each step.
23 find hand  : matrix indexed by [ STEPS ] of int(-1..100)
24 $Represents item in player's hand at each step.
25 find reg   : matrix indexed by [ STEPS , REG_NO ] of int(-1..100)
```

Figure 1: Code extract of parameters, domains and decision variables.

- `OUT_LEN` - for the length of the output list. Used in matrix list searches later in the programme and the construction of the outbox matrix.
- `REG_NO` - for the length of the registers. Used in matrix list searches later in the programme and the construction of the reg matrix.

For this programme I used the following decision variables (see lines 18 - 26 fig(1)).

- `action` - this matrix records which action (see section 3) is taken at each step. Each action has a unique integer from 1 to 8. This matrix is useful for debugging as you can identify from the terminal output which constraint may be causing issues. Indexed from 1 as there is not action for the first step.
- `inbox` - used to track the game's `input` conveyor belt at each `stage`. One of it's dimensions is the same length as the input and this dimension is also ordered as a stack such that `inbox[i]` can only be added to the player's hands if `inbox[j]`$= -1 \forall$ `j` $<$ `i`. Have used a separate inbox matrix to track the changes so that the input variable is unchanged throughout the programme.
- `outbox` - similar to inbox above but used to track the game's `output` conveyor belt at each stage. One of it's dimensions is the same length as the output and is also ordered as a stack such that `outbox[i]` can only be added from the player's hands if `outbox[j]`$= -1 \forall$ `j` $>$ `i`. Have used a separate outbox matrix to track the changes so that the output variable is unchanged throughout the programme.
- `hand` - matrix which represents integer in the player's hands at each step. Is a 1D matrix as the player can only hold one item at each step.
- `reg` - matrix which represents floor registers at each step. One dimension is equal to the number of registers determined by `REG_NO` from parameter file. These slots are not ordered as a stack and can be accessed in any order by the player.

For `inbox, outbox, hand & reg` I have used -1 as a dummy variable to represent an empty space. Therefore the domains range from -1 to the maximum integer size of 100. -1 was an appropriate

selection as it was easy to identify empty spaces using a < 0 condition and was outside the domain of possible input integers. Although this was not implemented in my program I considered it could also be used in further amendments to count how many empty spaces were in a 1D matrix for a specific step e.g *number of empty positions = sum(output) - sum(outbox[step]).*

3 Constraints

This section will discuss the constraints of the programme. The constraints section of the programme can be divided into 5 stages; first the target output is defined, then the inital state for all the decision variables is defined, next the first step is done manually, the programme then loops through `STEPS1` and finally the last step is completed outside the loop.

3.1 End State

The programme first sets the target end state. From the constraints of the Human Resource Game the final outbox should be equal to the output from the parameter file. This is represented in the programme as seen in fig(2). Since the first dimension of the 2D `outbox` matrix is the number of steps then the final step will be `MAX_OPS`.

```
30  $target final state.
31  (forAll i:OUT_LEN .
32   outbox[MAX_OPS,i] = output[i]),
```

Figure 2: Constraint for final state of program.

3.2 Step 0

Step 0 sets up the decision matrices. The steps determined by MAX_OPS are numbered 1 - MAX_OPS so this initial step 0 is not included in this count. At the beginning of the game the player's hands, the outbox and the registers are all empty. This is represented with assigning the empty place value holder -1 for each of these slots as seen in fig(3). The only non-empty slots are in the inbox which is set to equal to the input variable.

```
35  $sets up initial state with all empty except inbox filled with input.
36  forAll o:OUT_LEN .
37    outbox[0,o]= -1,
38
39  forAll r:REG_NO .
40    reg[0,r]= -1,
41
42  forAll i:IN_LEN .
43    inbox[0,i]= input[i],
44
45  hand[0] = -1,
```

Figure 3: Code extract for initial stage.

3.3 Step 1

To improve the efficiency of this programme I have manually programmed the first step, to save the programme from searching through all of the available actions. This can be done because the first step will always be the player picking up the first item from the inbox, the implementation of this can be seen in fig(4).

3.4 Steps (2..MAX_OPS-1)

This loop continues the steps from the constraints described above and iterates through every step until the penultimate step according to the `MAX_OPS` variable. Each set of constraints in this loop represent an action in the game and one action occurs for each step. As explained in the following

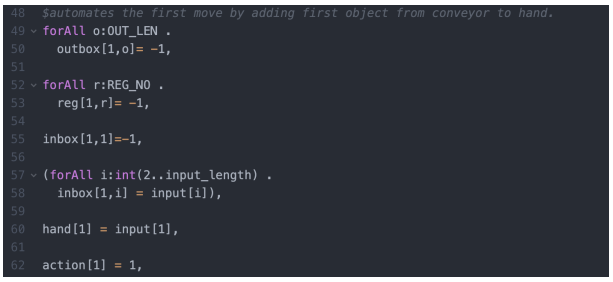


Figure 4: Code extract for step 1.

section, the last step is a special case. Fig(5) shows the start of the loop using the `STEPS1` domain defined at the start.

Inside this loop, before the set of the constraints, `out_count` is found. This is so that it can be used in the `inbox` action to increase efficiency as explained in section 3.4.1. `out_count` uses 2 constraints to find an integer for the location of the top of the `outbox` stack. It does this by finding the location of the empty slot (-1) with a non-empty slot to the left of it (see line 71). This will not work in the case of the first slot where all the slots will be empty, so this has a special case (see line 73).

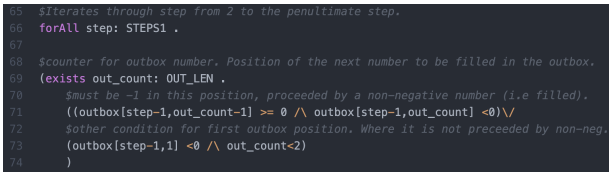


Figure 5: Code extract of the beginning of the loop.

The following actions described this section have some structural similarities. These will be explained generally here to avoid repetition. For each step, these actions will repopulate every decision variable with it's values from the previous step, except the decision variable which the action is performed on and the `action` matrix. As mentioned before, the `action` decision matrix tracks which action happens at each stage. The matrix is updated accordingly at each step and each action has a unique integer matching to their subsection numbers in this report e.g 1 - Inbox , 2 - Outbox...

3.4.1 Inbox

INBOX action takes the item from the top of the inbox stack and assigns it to the players hands. The `exist` function with the constraint in line 80 in fig(6) works to find the integer value of the top of the stack for the previous step. The `in_count`> 2 because the first step of assigning the first inbox value to the player's hand has been done before this loop.

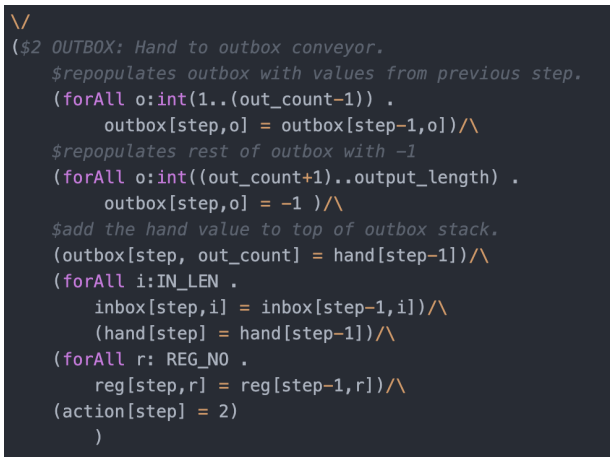
This INBOX action uses the predetermined `out_count` to improve the programme's efficiency (see line 80 fig(6)). `inbox` is the only action which discards the integer in the player's hand, therefore this action should never be done if the integer in the player's hand is the same as the next value needed on the `outbox` stack according to the `output` variable. This added constraint prevents the programme from unnecessarily testing this action in this situation.

```
/\
( $1 INBOX: from inbox conveyor to hand.
  $First condition avoids returning to inbox and
  $dropping hand value if hand value is equal to the next item needed in outbox.
  ((hand[step-1]!=output[out_count])/\
  $finds position top of inbox stack, where it is non-empty but has an empty slot
  $directly to the right.
  (exists in_count: int(2..input_length).
    (inbox[step-1,in_count-1] <0 /\ inbox[step-1,in_count] >= 0)/\
    $repopulates empty inbox with -1
    (forAll i:int(1..in_count) .
        inbox[step,i] = -1)/\
    $repopulates the rest of the imbox with values from the previous step.
    (forAll i:int((in_count+1)..input_length) .
        inbox[step,i] = inbox[step-1,i])/\
    $adds the hand from the top of the inbox stack to the player's hand.
    (hand[step] = inbox[step-1,in_count])/\
    (forAll o:OUT_LEN .
        outbox[step,o] = outbox[step-1,o])/\
    (forAll r: REG_NO .
          reg[step,r] = reg[step-1,r])/\
    (action[step] = 1)
        ))
    )
```

Figure 6: Code extract of INBOX action constraints.

3.4.2 Outbox

OUTBOX action copies the integer from the player's hand to the top of the outbox stack. This action takes `out_count` from the constraint at the start of the loop.



```
\/
($2 OUTBOX: Hand to outbox conveyor.
    $repopulates outbox with values from previous step.
    (forAll o:int(1..(out_count-1)) .
         outbox[step,o] = outbox[step-1,o])/\
    $repopulates rest of outbox with -1
    (forAll o:int((out_count+1)..output_length) .
         outbox[step,o] = -1 )/\
    $add the hand value to top of outbox stack.
    (outbox[step, out_count] = hand[step-1])/\
    (forAll i:IN_LEN .
        inbox[step,i] = inbox[step-1,i])/\
        (hand[step] = hand[step-1])/\
    (forAll r: REG_NO .
        reg[step,r] = reg[step-1,r])/\
    (action[step] = 2)
        )
```

Figure 7: Code extract of OUTBOX action constraints.

3.4.3 CopyTo

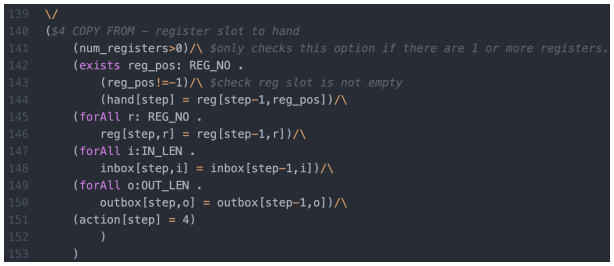
COPYTO action as seen in fig(8) copies the integer from the player's hand to a register slot. As mentioned previously the registers are not in a stack so any slot can be chosen. This action and all the following actions in this section have a check (see line 121 fig(8) that the number of registers is greater than 0. This check improves efficiency because it prevents the model from testing these actions unnecessarily if there are no register slots. Line 121 fig(8) enforces the constraint that this action cannot be performed if the hands are empty (-1).

3.4.4 CopyFrom

COPYFROM has a similar functionality to COPYTO as explained in the previous subsection, however copies the integer from a register slot into the player's hand as seen in line 144 fig(9).

```
\/
($3 COPY TO - hand to register slot
    (hand[step-1]!=output[out_count])/\
    (num_registers>0)/\ $only checks this option if there are 1 or more registers.
    (hand[step-1]!=-1)/\  $checks hand is not empty
    (exists reg_pos: REG_NO . $integer in player's hand to a register slot.
      (reg[step,reg_pos] = hand[step-1] )/\
    (forAll r:int(1..(reg_pos-1)).
       reg[step,r] = reg[step-1,r])/\
    (forAll r:int((reg_pos+1)..num_registers).
       reg[step,r] = reg[step-1,r])/\
       (hand[step] = hand[step-1])/\
    (forAll i:IN_LEN .
      inbox[step,i] = inbox[step-1,i])/\
    (forAll o:OUT_LEN .
      outbox[step,o] = outbox[step-1,o])/\
    (action[step] = 3)
        )
    )
```

Figure 8: Code extra of COPYTO action constraints.

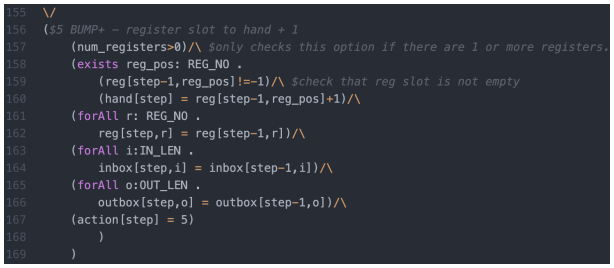


```
\/
($4 COPY FROM - register slot to hand
    (num_registers>0)/\ $only checks this option if there are 1 or more registers.
    (exists reg_pos: REG_NO .
        (reg_pos!=-1)/\ $check reg slot is not empty
        (hand[step] = reg[step-1,reg_pos])/\
    (forAll r: REG_NO .
        reg[step,r] = reg[step-1,r])/\
    (forAll i:IN_LEN .
        inbox[step,i] = inbox[step-1,i])/\
    (forAll o:OUT_LEN .
        outbox[step,o] = outbox[step-1,o])/\
    (action[step] = 4)
        )
    )
```

Figure 9: Code extract of COPYFROM action constraints.

3.4.5 Bump+

BUMP+ copies the value from a register slot into the player's hands and adds 1 to the value, as seen in line 159 fig(10).



```
\/
($5 BUMP+ - register slot to hand + 1
    (num_registers>0)/\ $only checks this option if there are 1 or more registers.
    (exists reg_pos: REG_NO .
        (reg[step-1,reg_pos]!=-1)/\ $check that reg slot is not empty
        (hand[step] = reg[step-1,reg_pos]+1)/\
    (forAll r: REG_NO .
        reg[step,r] = reg[step-1,r])/\
    (forAll i:IN_LEN .
        inbox[step,i] = inbox[step-1,i])/\
    (forAll o:OUT_LEN .
        outbox[step,o] = outbox[step-1,o])/\
    (action[step] = 5)
        )
    )
```

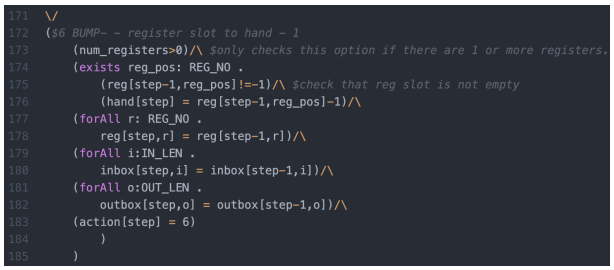
Figure 10: Code extract of BUMP+ action constraints.

3.4.6 Bump-

BUMP is similar to BUMP above but copies the value from a register slot into the player's hands and adds 1 to the value, as seen in line 176 fig(11).

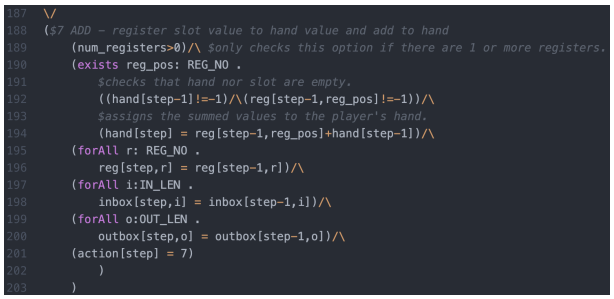
3.4.7 Add

ADD chooses a register slot and adds the number in the register to the number in the player's hand for the previous step and assigns the sum value to the players hand for the current step. This action is implemented in line 194 fig(12).



```
171  \/
172  ($6 BUMP- - register slot to hand - 1
173      (num_registers>0)/\ $only checks this option if there are 1 or more registers.
174      (exists reg_pos: REG_NO .
175          (reg[step-1,reg_pos]!=-1)/\ $check that reg slot is not empty
176          (hand[step] = reg[step-1,reg_pos]-1)/\
177      (forAll r: REG_NO .
178          reg[step,r] = reg[step-1,r])/\
179      (forAll i:IN_LEN .
180          inbox[step,i] = inbox[step-1,i])/\
181      (forAll o:OUT_LEN .
182          outbox[step,o] = outbox[step-1,o])/\
183      (action[step] = 6)
184          )
185      )
```

Figure 11: Code extract of BUMP action constraints.

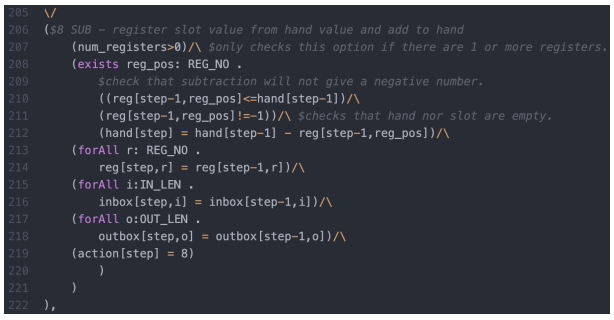


```
187  \/
188  ($7 ADD - register slot value to hand value and add to hand
189      (num_registers>0)/\ $only checks this option if there are 1 or more registers.
190      (exists reg_pos: REG_NO .
191          $checks that hand nor slot are empty.
192          ((hand[step-1]!=-1)/\(reg[step-1,reg_pos]!=-1))/\
193          $assigns the summed values to the player's hand.
194          (hand[step] = reg[step-1,reg_pos]+hand[step-1])/\
195      (forAll r: REG_NO .
196          reg[step,r] = reg[step-1,r])/\
197      (forAll i:IN_LEN .
198          inbox[step,i] = inbox[step-1,i])/\
199      (forAll o:OUT_LEN .
200          outbox[step,o] = outbox[step-1,o])/\
201      (action[step] = 7)
202          )
203      )
```

Figure 12: Code extract of ADD action constraints.

3.4.8 Sub

SUB chooses a register slot and subtracts the number in the register from the number in the player's hand for the previous step and assigns the new value to the players hand for the current step, this is implemented in line 212 fig(13). This includes an extra check in line 210 that the operation will give a positive number in the player's hand.



```
205  \/
206  ($8 SUB - register slot value from hand value and add to hand
207      (num_registers>0)/\ $only checks this option if there are 1 or more registers.
208      (exists reg_pos: REG_NO .
209          $check that subtraction will not give a negative number.
210          ((reg[step-1,reg_pos]<=hand[step-1])/\
211          (reg[step-1,reg_pos]!=-1))/\ $checks that hand nor slot are empty.
212          (hand[step] = hand[step-1] - reg[step-1,reg_pos])/\
213      (forAll r: REG_NO .
214          reg[step,r] = reg[step-1,r])/\
215      (forAll i:IN_LEN .
216          inbox[step,i] = inbox[step-1,i])/\
217      (forAll o:OUT_LEN .
218          outbox[step,o] = outbox[step-1,o])/\
219      (action[step] = 8)
220          )
221      )
222  ),
```

Figure 13: Code extract of SUB action constraints.

3.5 Final Step

The final step can be done outside to improve the efficiency of the programme and prevent the model from having to search through all the possible actions listed above. If the `outbox` is not already equal to the `output` then if the variables have a solution, then the final action must be placing the integer in the player's hands onto the outbox. This is implemented in the code in fig(14).

```
((outbox[MAX_OPS-1,output_length]!= output[output_length])/\
(outbox[MAX_OPS, output_length] = hand[MAX_OPS-1])/\
(forAll i:IN_LEN .
  inbox[MAX_OPS,i] = inbox[MAX_OPS-1,i])/\
(forAll o:int(1..(output_length-1)) .
    outbox[MAX_OPS,o] = outbox[MAX_OPS-1,o])/\
  (hand[MAX_OPS] = hand[MAX_OPS-1])/\
  (forAll r: REG_NO .
    reg[MAX_OPS,r] = reg[MAX_OPS-1,r])/\
    (action[MAX_OPS] = 2)
  ),
```

Figure 14: Code extract for the final step.

4 Example Solutions

This section will discuss the sample solutions for 3.param and 5.param when these variables are used in the programme described in the previous sections, the terminal outputs can be seen in fig(15) and fig(16).

I will start with a general discussion about the 2 outputs. Each row in the `inbox`, `outbox` and `reg` matrix represent the states at each step from 0 to `MAX_OPS`.

Each terminal output has an `warning` message caused by the loop in each of the actions where the decision variables are repopulated for the next step, e.g. lines 86 - 90 for fig(6). In the `INBOX` action this will create a similar error message when `in_count = input_length` as the domain in line 89 will be in the wrong order. In practice this means that there is nothing left on the inbox conveyor so the programme does not need to repopulate the inbox with any parameters from the previous step. Since Savile Row treats the misordered array as empty this works perfectly.

```
WARNING: interval 5..4 is out of order. Rewriting to empty interval.
WARNING: interval 2..1 is out of order. Rewriting to empty interval.
WARNING: interval 1..0 is out of order. Rewriting to empty interval.
WARNING: interval 1..0 is out of order. Rewriting to empty interval.
WARNING: interval 3..2 is out of order. Rewriting to empty interval.
Created output file for domain filtering sat-instances/3.param.minion
Created output file sat-instances/3.param.minion
language ESSENCE' 1.0
$ Minion SolverNodes: 12136
$ Minion SolverTotalTime: 0.190857
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.836
letting action be [1, 3, 1, 7, 2, 1, 3, 1, 7, 2]
letting hand be [-1, 3, 3, 2, 5, 5, 9, 9, 12, 21, 21;int(0..10)]
letting inbox be [[3, 2, 9, 12],
[-1, 2, 9, 12],
[-1, 2, 9, 12],
[-1, -1, 9, 12],
[-1, -1, 9, 12],
[-1, -1, 9, 12],
[-1, -1, -1, 12],
[-1, -1, -1, 12],
[-1, -1, -1, -1],
[-1, -1, -1, -1],
[-1, -1, -1, -1];int(0..10)]
letting outbox be [[-1, -1],
[-1, -1],
[-1, -1],
[-1, -1],
[-1, -1],
[5, -1],
[5, -1],
[5, -1],
[5, -1],
[5, -1],
[5, 21];int(0..10)]
letting reg be [[-1],
[-1],
[3],
[3],
[3],
[3],
[3],
[9],
[9],
[9],
[9];int(0..10)]

---------
Created information file sat-instances/3.param.info
```

Figure 15: Terminal output for 3.param.

```
WARNING: interval 4..3 is out of order. Rewriting to empty interval.
WARNING: interval 2..1 is out of order. Rewriting to empty interval.
WARNING: interval 1..0 is out of order. Rewriting to empty interval.
WARNING: interval 1..0 is out of order. Rewriting to empty interval.
WARNING: interval 4..3 is out of order. Rewriting to empty interval.
Created output file for domain filtering sat-instances/5.param.minion
Created output file sat-instances/5.param.minion
language ESSENCE' 1.0
$ Minion SolverNodes: 721494
$ Minion SolverTotalTime: 13.6874
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 1.108
letting action be [1, 3, 1, 7, 2, 3, 3, 7, 2, 1, 3, 7, 7, 2]
letting hand be [-1, 4, 4, 8, 12, 12, 12, 12, 24, 24, 5, 5, 10, 15, 15;int(0..14)]
letting inbox be [[4, 8, 5],
[-1, 8, 5],
[-1, 8, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, 5],
[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1];int(0..14)]
letting outbox be [[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1],
[-1, -1, -1],
[12, -1, -1],
[12, -1, -1],
[12, -1, -1],
[12, -1, -1],
[12, 24, -1],
[12, 24, -1],
[12, 24, -1],
[12, 24, -1],
[12, 24, -1],
[12, 24, 15];int(0..14)]
letting reg be [[-1],
[-1],
[4],
[4],
[4],
[4],
[12],
[12],
[12],
[12],
[12],
[5],
[5],
[5],
[5];int(0..14)]

---------
Created information file sat-instances/5.param.info
```

Figure 16: Terminal output for 5.param.

4.1 3.param

This input file had the following variables:

```
letting input=[3,2,9,12]
letting output=[5,21]
letting MAX_OPS=10
letting num_registers=1.
```

From the `action` decision variable in fig(15) we can see that solution uses the maximum number of operations and the order of actions were;

```
[INBOX, COPYTO, INBOX, ADD, OUTBOX, INBOX, COPYTO, INBOX, ADD, OUTBOX]
```

4.2 5.param

This input file had the following variables:

```
letting input=[4,8,5]
letting output=[12,24,15]
letting MAX_OPS=14
letting num_registers=1
```

From the `action` decision variable in fig(15) we can see that solution uses the maximum number of operations and the order of actions were;

`[INBOX, COPYTO, INBOX, ADD, OUTBOX, COPYTO , COPYTO, ADD, OUTBOX, INBOX, COPYTO, ADD, ADD, OUTBOX]`

4.3 Comparison

Both solutions use the same set of actions in the solution {`1,2,3,7`} however 3.param has a smaller number of `MAX_OPS`. 3.param also has a smaller `output` length but a larger `input`. We can see that for 5.param the model uses almost 6 times as many solver nodes and has a much larger Saville Row and Solver Total time. The possible reasons for this will be explained in more detail in section 5.

5 Empirical Evaluation

In the following comparisons I have used `Total Time = Solver Solve Time + SavileRow Time`. This is not necessarily the entire time taken for the entire search but is a useful metric for the following comparisons.

5.1 10 minute Run Test Results

Param No	Complete?	Solver Nodes	Solver Solve Time	SavilleRow Time	Total Time	No. Operators	Actions
1		1	0.504	0.000405	0.504405	6	[1, 2, 1, 2, 1, 2]
2		1157	0.014912	0.999	1.013912	2	[1, 1, 1, 2, 3, 5, 2, 7, 2]
3		8443	0.015328	0.828	0.843328	10	[1, 3, 1, 7, 2, 1, 3, 1, 7, 2]
4		1	0.000074	0.000431	0.000505	11	[1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 2]
5		405822	8.52591	1.402	9.92791	14	[1, 3, 1, 7, 2, 3, 3, 7, 2, 1, 3, 7, 7, 2]
6		1020952	22.1087	1.781	23.8897	14	[1, 3, 7, 3, 7, 3, 7, 2, 3, 7, 2, 1, 7, 2]
7	x	x	x	x	x	11	x
8		6138	0.381457	3.494	3.875457	15	[1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 2]
9		43	106.258	1.823	108.081	10	[1, 1, 3, 6, 2, 1, 2, 1, 1, 2]
10		1929680	103.434	2.54	105.974	20	[1, 3, 1, 7, 2, 3, 3, 7, 2, 1, 3, 1, 7, 2, 3, 7, 2, 3, 7, 2]
11		47	0.007128	41.392	41.399128	24	[1, 1, 1, 1, 2, 3, 5, 2, 2, 4, 2, 5, 2, 2, 4, 2, 5, 2, 4, 2, 5, 2, 4, 2]
12		211934	10.1071	9.831	19.938	40	[1, 1, 1, 1, 3, 1, 4, 5, 3, 7, 2, 5, 2, 4, 2, 6, 2, 3, 6, 2, 5, 7, 2, 5, 2, 4, 2, 6, 2, 5, 2, 4, 2, 6, 2, 4, 2, 6, 2, 2]
13		2432777	78.7711	1.711	80.4821	14	[1, 3, 1, 1, 2, 7, 7, 3, 7, 2, 7, 7, 7, 2]
14		1470854	55.509	2.968	58.477	16	[1, 1, 1, 3, 3, 7, 2, 3, 7, 7, 7, 7, 7, 2, 6, 2]
15		236015638	2195.640	2.107	2197.75	12	[1, 1, 1, 3, 7, 3, 8, 2, 6, 2, 4, 2]
16		2712405	152.839	3.622	156.461	17	[1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 3, 2]
17	x	0	0.000147	7.717	7.717147	11	x
18	x	0	0.00261	4.834	4.83661	14	x
19	x	0	2.30E-05	0.776	0.776023	3	x
20	x	0	0.00014	6.363	6.36314	13	x

Figure 17: Table showing results for each test case. Tests were run for a maximum of 10 minutes and are marked with an x if they did not complete in this time. 1 - 16 are satisfiable and 17 - 20 are unsatisfiable.

Can see in the table in fig(17) that param 15 took a significantly longer time to solve (2195s) compared to the other parameters. As a result this has been discarded as an outlier in some of the following graphs. Param 15 is also the only one which used action 8 (`SUB`) suggesting that there is an issue with this constraint.

The table in fig(17) also shows that a solution was not found for param 7, suggesting another issue with the code. The table in fig(18) shows the results when the programme was run on 7v2, the only modification compared to 7 is the number of operators has been increased by 1.

Param No	Complete?	Solver Nodes	Solver Solve Time	SavilleRow Time	Total Time	No. Operators	Actions
7v2		97	0.036339	1.648	1.684339	**12**	[1, 1, 1, 1, 3, 8, 2, 1, 6, 2, 1, 2]

Figure 18: Table showing results for alternative version of 7.param with an increased number of maximum operations.

The following graphs have been generated using the data from fig(17).

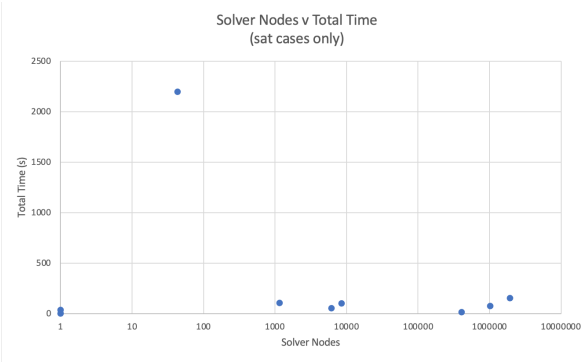


Figure 19: Graph showing total time (Savile Row time + Solver Solve time) compared to the number of solver nodes for the satisfiable test cases.

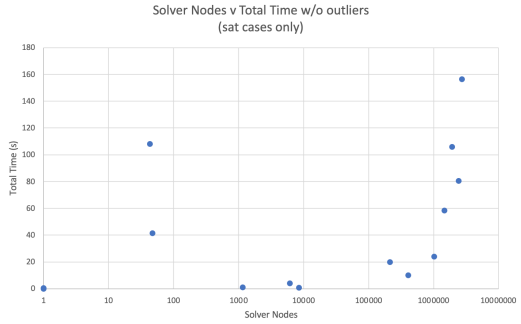


Figure 20: Graph showing total time taken compared to the number of solver nodes, with the outlier removed. No strong correlation. Larger range in total times when solver nodes are very large.

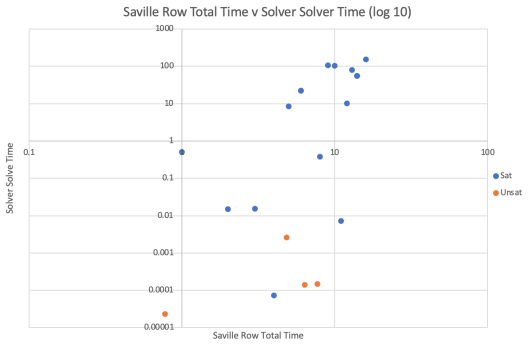


Figure 21: Graph showing how Savile Row time compares to Solver Solve time. Both axis have been adjusted to $\log_{10}$ to show the relationship more clearly. This graph shows a weak positive correlation between the two.

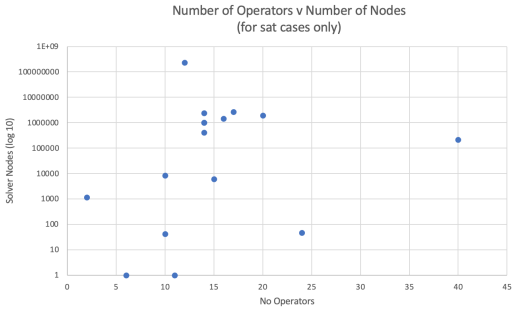


Figure 22: Graph showing how the number of nodes increases with the number of operators (as determined by `MAX_OPS`). . This graph shows a weak correlation. Only the satisfiable cases have been included here because the unsatisfiable cases all have 0 nodes.

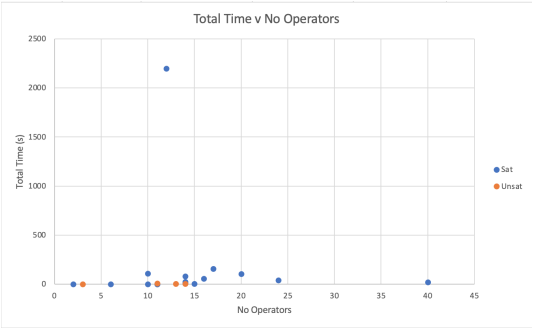


Figure 23: Graph showing how the total time for the solver to run compared to the number of operators (as determined by `MAX_OPS`).

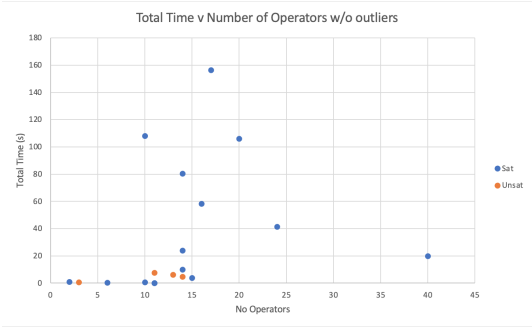


Figure 24: Graph showing how the total time for the solver to run compared to the number of operators with the outlier removed. There is no correlation.

5.2 Bump Removal

The table in fig(25) can be compared to param no 5 in fig(17). Although in the either case the `BUMP+` [4] is not used we can see that by removing this action as an option, the total run time has decreased and the number of solver nodes has decreased. The run time and number of nodes will have decreased because the model has had to search fewer possible actions. As we have seen in fig(22) there is a weak positive correlation between the total run time and the number of nodes. This experiment has demonstrated that the option of unnecessary actions can decrease the model's efficency.

Param No	Complete?	Solver Nodes	Solver Time	Saville Row Time	Total Time	No. Operators	Actions
5		110188	2.39911	1.142	3.541	13	[1, 3, 7, 3, 7, 3, 7, 2, 3, 7, 2, 1, 7, 2]

Figure 25: Table showing results for 5.param when `BUMP` is removed.

5.3 Scalability

Using generator.java I produced some more test cases of varying input and output lengths (also with a range of seed values to ensure varied complexity). The results when the solver was run on each of these test cases for 10 minutes can be seen in fig(26). For borderline combinations {(30,5) , (5,30) , (15,10) , (10,15)} I ran the tests for 3 different seed values, non of which were successful. This experiment could be improved by automating the generation of the variables for a variety of seed values and then running this test several times for each combination of input/output length and using a mean time for the table.

The table in fig(26) shows the limitations of this model's ability to be scaled within the 10 minute limit. The run time increases as both the output and input lengths decrease however the length of the output has a more significant impact on the increase in time. Intuitively it follows that the more items required in the outbox the more actions the player must complete to fill it. The generation of numbers, especially if they are not already in the inbox will require many steps such as `ADD`, `SUB` ect and dramatically increase the complexity of the solution search.

\Input Length Output Length	5	10	15	20	25	30
5	1.488	10.171	15.666	18.971	19.526	x
10	10.171	22.748	x	x	x	x
15	35.746	x	x	x	x	x
20	62.7091	x	x	x	x	x
25	24.133	x	x	x	x	x
30	x	x	x	x	x	x

Figure 26: Table showing SavilleRow Time + Solver Solve Time for parameters with a variety of input and output lengths. Times are in seconds and x shows that no result was found after 10 minutes (neither satisfiable nor unsatisfiable).

6 Previous Version

The Essence Prime code discussed in this report so far is my final version (v1), contained additional constraints to improve the efficiency. These are discussed in depth in section 3. The purpose of this section is to demonstrate the optimization due to the additional constraints in v1. This section will compare v1 to my first basic version (v0) of the programme by highlighting key differences in the constraints and comparing test results.

6.1 Constraints

Comparing fig(27) to fig(1) the key difference is that `STEPS1` is defined from `1..MAX_OPS` instead of `2..MAX_OPS-1` in v1. Optimization constraints in v1 save 2 loops through `STEPS1`.

```
given MAX_OPS: int(1..)
given num_registers : int(0..)
given input: matrix indexed by [int(1..input_length)] of int(0..100)
given output: matrix indexed by [int(1..output_length)] of int(0..100)

$ Excludes initial state
letting STEPS be domain int(0..MAX_OPS)
letting STEPS1 be domain int(1..MAX_OPS)

letting IN_LEN be domain int(1..input_length)
letting OUT_LEN be domain int(1..output_length)
letting REG_NO be domain int(1..num_registers)

$different integer value for each action at each step.
find action: matrix indexed by [int(1..MAX_OPS)] of int(1..8)
$Ordered and represents items in the inbox at each step.
find inbox : matrix indexed by [STEPS,  IN_LEN] of int(-1..100)
$Ordered and represents items in the outbox at each step.
find outbox: matrix indexed by [ STEPS, OUT_LEN ] of int(-1..100)
$Unordered an represents items in register slots at each step.
find hand  : matrix indexed by [ STEPS ] of int(-1..100)
$Represents item in player's hand at each step.
find reg   : matrix indexed by [ STEPS , REG_NO ] of int(-1..100)

```

Figure 27: Parameter, domain and decision variable declaration.

Fig(28) shows the set up for the initial step 0 similar to fig(3) for v1. However this older version does not set up the step 0 as seen in fig(4) for v1, therefore this model must do a timely search through all the actions for step 0.

```
$target final state.
(forAll i:OUT_LEN .
 outbox[MAX_OPS,i] = output[i]),

forAll o:OUT_LEN .
  outbox[0,o]= -1,

forAll r:REG_NO .
  reg[0,r]= -1,

forAll i:IN_LEN .
  inbox[0,i]= input[i],

hand[0] = -1,

```

Figure 28: Step 0.

Figure 29: Start of steps loop.

Fig(29) shows the start of the loop which iterates through `STEPS1` for each of the 8 actions. Only `INBOX` and `OUTBOX` are shown here as the codes in v1 and v0 for the remaining actions are identical.

The difference between fig(29) and fig(5) is that for v0 `out_count` is determined in the `OUTBOX` step. This means that the programme can still try `INBOX` even if the player has the next item for the outbox in their hand, which may increase the search time.

The final difference between the 2 versions is the automation of OUTBOX for `step[MAX_OPS]` in v1. Instead for v0 `step[MAX_OPS]` is included in the `STEPS1` loop above, this will further increase search time.

6.2 Results Comparison

The results for v0 in fig(30) can be compared the results for v1 in fig(17) to see the effect of the programme optimization. The table in fig(30) shows that in 10 minutes the v0 code was only able to find solutions for 4 satisfiable and 2 unsatisfiable parameters. This clearly shows the impact on search time of v1 compared to v0 due to the optimization constraints. For the parameters that v0 was able to find a solution for, it did so in a shorter time. This may be due to the fact that the v0 code is simpler and does not require `out_count` to be calculated at each stage. However the run time clearly increases dramatically as the complexity of the parameters increases, since it was unable to find a solution for the majority of the parameters within 10 minutes.

The terminal output in fig(31) can be compared to the terminal output as seen in fig(15). Here we can see that both v0 and v1 returned a solution with the same decision variable matrices however v0 used 30 times as many nodes, demonstrating the additional searches required in v0 compared to v1.

Param No	Complete?	Solver Nodes	Solver Solve	SavilleRow Time	Total Time	No. Operators	Actions
1		1	6.50E-05	0.481	0.48	6	[1, 2, 1, 2, 1, 2]
2		40520	0.000217	0.826	0.83	2	[1, 1, 1, 2, 3, 5, 2, 7, 2]
3		403991	0.091389	0.628	0.72	10	[1, 3, 1, 7, 2, 1, 3, 1, 7, 2]
4		142	9.40E-05	0.822	0.82	11	[1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 2]
5	x	x	x	x	x	14	x
6	x	x	x	x	x	14	x
7	x	x	x	x	x	11	x
8	x	x	x	x	x	15	x
9	x	x	x	x	x	10	x
10	x	x	x	x	x	20	x
11	x	x	x	x	x	24	x
12	x	x	x	x	x	40	x
13	x	x	x	x	x	14	x
14	x	x	x	x	x	16	x
15	x	x	x	x	x	12	x
16	x	x	x	x	x	17	x
17		0	9.50E-05	1.803	1.803095	11	x
18	x	x	x	x	x	14	x
19		0	5.40E-05	0.53	5.30E-01	3	x
20	x	x	x	x	x	13	x

Figure 30: Table showing results for v0 for each test case. Tests were run for 10 minutes and are marked with an x if did not complete in this time. 1 - 16 are satisfiable and 17 - 20 are unsatisfiable.

```
$ Minion SolverNodes: 403991
$ Minion SolverTotalTime: 6.89658
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.779
letting action be [1, 3, 1, 7, 2, 1, 3, 1, 7, 2]
letting hand be [-1, 3, 3, 2, 5, 5, 9, 9, 12, 21, 21;int(0..10)]
letting inbox be [[3, 2, 9, 12],
[-1, 2, 9, 12],
[-1, 2, 9, 12],
[-1, -1, 9, 12],
[-1, -1, 9, 12],
[-1, -1, 9, 12],
[-1, -1, -1, 12],
[-1, -1, -1, 12],
[-1, -1, -1, -1],
[-1, -1, -1, -1],
[-1, -1, -1, -1];int(0..10)]
letting outbox be [[-1, -1],
[-1, -1],
[-1, -1],
[-1, -1],
[-1, -1],
[5, -1],
[5, -1],
[5, -1],
[5, -1],
[5, -1],
[5, 21];int(0..10)]
letting reg be [[-1],
[-1],
[3],
[3],
[3],
[3],
[3],
[9],
[9],
[9],
[9];int(0..10)]
```

Figure 31: Terminal output for 3.param with v0.

7 Conclusion

in this report I have explained my approach to the planning problem and use of domains and decision variables. I have also evaluated the performance of the model through testing and discussed it's limitations. This report also compared the model to a more basic version to demonstrate the optimization of additional constraints.

During this project a significant issue was long run times for models, when an unnecessary loop or over complicated constraint would prevent any solution (satisfiable or non-satisfiable) being found. I found the most effective way to avoid this was to build the code gradually and regularly test each amendment for a variety of parameters.

I have also mentioned in this report the need for possible improvements for this model due to scalability and long run times as seen for action 8 and parameter 7 in section 5. Some ideas for how these could be implemented include;

- Use the out_count loop as seen in `INBOX` could be used for other steps to help break more symmetries.
- In this current model, an action must occur for each step. Therefore the `MAX_OPS` number of moves is always used, adding a functionality to the model where it can use less than the maximum number of moves could be explored.

  There may be also be a more favourable viewpoint for this model.Finally, as mentioned previously, the generation of test cases and running of them could be automated to increase the data set of test data.