CS4402 P01 Report

matriculation number: 160021429

1. Overview

The main aim of this practical is to design and create a model of Late Binding Solitaire for Accordion Solitaire in Essence Prime. The formulated model should be tested its performance on a number of supplied instances.

1 - 1. Savile Row

To test the created model, the Savile Row was used. A Savile Row is a constraint modelling tool that provides a high-level language for the user to specify their constraint problem, and automatically translates that language to the input language of a constraint solver.

2. Design and Implementation

If you see the main.eprime file, you could find decision variables, domains, and constraints for the Late Binding Solitaire.

2 - 1. Domains

1) LENGTH

```
letting LENGTH be domain int(1..n)
```

The “LENGTH” is a domain of integer values that contains integers from 1 to n. It is used for iterating the cards, since the number of given cards is n.

2) RANGE

```
letting RANGE be domain int(1..n-1)
```

The “RANGE” is a domain of integer values that contains integers from 1 to (n - 1). Basically, the number of moves should be (n - 1), thus, I declared this domain to iterate a list of moves.

3) MOVES

```
letting MOVES be domain int(1, 3)
```

The “MOVES” is a domain of 1 and 3. Basically, in the Accordion Solitaire, the source pile has to be to the right and either adjacent or 3 positions away from the destination pile. This means that in each state, the model is permitted to move a card either 1 or 3 indexes to its left. That is the main reason that the domain “MOVES” only contains 1 and 3.

4) CARD_DOMAIN

```
letting CARD_DOMAIN be domain int(-1, cards)
```

The “CARD_DOMAIN” is a domain that contains either -1 or values in the “cards”. The “-1” is used for representing the empty spaces. Basically, when the model makes a move, then the card on the destination pile should be removed. So, the model should be able to record all empty spaces in each step. Due to this reason, my model uses -1 for the empty space. Also, the model never add new cards during playing the game, values of all cards in each state should be either -1 (an empty space) or one of the values in the initial state. This is the reason that the domain “CARD_DOMAIN” only contains -1 and values in the “cards”.

2 - 2. Decision Variables

1) moves

```
find moves : matrix indexed by [RANGE] of MOVES
```

The “moves” is a list that contains (n - 1) integers. This decision variable is used for finding the number of moved indexes for all (n - 1) moves. Since the game starts from n cards, and ends when the number of remaining cards is 1, the number of moves should be (n - 1). Furthermore, for each move, the model is permitted to move a card either 1 or 3 indexes to its left. Thus, a decision variable “moves” should be a list that has (n - 1) elements, where each element is either 1 or 3.

2) movesFrom

```
find movesFrom : matrix indexed by [RANGE] of int(2..n)
```

The “movesFrom” is a list that contains (n - 1) integers. When the model moves a card, the model should memorise the index that the card moves from, and the index that the card moves to. The aim of this variable is to store all the indexes that the cards moves from.

Clearly, the model is not permitted to move a card from left to right. Thus, the “movesFrom” cannot contain 1 in it. Similarly, the maximum value that the “movesFrom” can store is n, since the number of initial cards is n. Thus, the “movesFrom” is a list that contains (n - 1) elements, where the value of each element should be an integer between 2 to n.

3) movesTo

```
find movesTo : matrix indexed by [RANGE] of int(1..n-1)
```

The “movesTo” is a list that contains (n - 1) integers. When the model moves a card, he model should memorise the index that the card moves from, and the index that the card moves to. The aim of this variable is to store all the indexes that the cards moves to.

The model is not permitted to move a card from left to right. Hence, every elements that the “movesTo” contains should be a positive integer that is less than n. Therefore, the “movesTo” should be a list of (n - 1) elements, where each element should be a positive integer that is less than n.

4) states

```
find states : matrix indexed by [LENGTH, LENGTH] of CARD_DOMAIN
```

The “states” is a 2D array that contains the all states. The size of the “states” is n x n, which means that it has n rows and n columns. The first row of the “states” should be equal to the “cards”, since it is an initial state. And the k^{th} row is a result of $(k-1)^{th}$ move. For example, the second row is a result of first move.

When the model makes a move, then the card on the destination pile should be removed. This means that the number of cards will be keep decreasing, and the number of empty space should be keep increasing. Since the variable “states” records all states, it should be able to represent the empty spaces. Thus, my model uses “-1” to represent the empty space. For example, if the 2^{nd} row of the states is “1 5 4 -1”, then this means that there are 3 cards 1, 5, and 4, and there is 1 empty space.

2 - 3. Constraints

1) Constraint for initial state

```
forAll j : LENGTH .
    states[1, j] = cards[j],
```

This constraint sets the starting state by filling the 1st row of the 2D array “states” with the values in the “cards”. Because, the first row of the “states” should be an initial state, which should be identical with the variable “cards”.

2) Constraints for the indexes of source piles and destination piles

```
forAll i : RANGE .
    movesFrom[i] = movesTo[i] + 1
    \/
    movesFrom[i] = movesTo[i] + 3,

forAll i : RANGE .
    movesFrom[i] = movesTo[i] + 1
    ->
    moves[i] = 1,

forAll i : RANGE .
    movesFrom[i] = movesTo[i] + 3
    ->
    moves[i] = 3,


forAll i : RANGE .
    movesFrom[i] > movesTo[i],
```

As I mentioned above, the model is permitted to move a card either 1 or 3 indexes to its left. This means that for all i from 1 to (n - 1), the value of movesFrom[i] should be either (movesTo[i] + 1) or (movesTo[i] + 3). If the value of movesFrom[i] is equal to (movesTo[i] + 1), then that means that the source pile is adjacent to the destination pile. Similarly, if the value of movesFrom[i] is equal to (movesTo[i] + 3), then that means that the source pile is 3 positions away from the destination pile. Based on these facts, we could derive 2 constraints:

a) $movesFrom[i] == (movesTo[i] + 3) \rightarrow moves[i] == 3$

b) $movesFrom[i] == (movesTo[i] + 1) \rightarrow moves[i] == 1$

The last constraint in the image makes the model not to move the card from left to right. Also, this constraint makes sure that the model does not move the card to the same position.

```
forAll i : RANGE .
    movesFrom[i] <= n - (i - 1),
```

The image above shows the constraint that regulates the model not to move “-1”, which represents the empty space. For all i from 1 to (n - 1), the value of $n-(i-1)$ should be the index of the rightmost non-empty position in the i^{th} row. For example, the value of the movesFrom[2] should be less than or equal to (n - 1), since the n^{th} position is an empty space.

3) Constraints for the decision variable “states”

```
forAll i : LENGTH .
    forAll j : LENGTH .
        j > (n - i + 1)
        ->
        states[i, j] = -1,
```

All rows from 2^{nd} row to n^{th} should contain suitable number of empty spaces. The constraint above sets all positions whose index is greater than the index of rightmost non-empty space with -1. In fact, the Savile Row automatically fills all not-modified cells in the list with default value, which is -1 in this case. However, I implemented this constraint, since I just wanted to make sure that the model sets all empty spaces with -1.

```
forAll a : RANGE .
    states[a, movesFrom[a]] % 13 = states[a, movesTo[a]] % 13
    \/
    states[a, movesFrom[a]] / 13 = states[a, movesTo[a]] / 13,
```

When the model makes a move, the source card and destination card should either have same shape or have same number. To regulate this, I used the modulo operator and division operator. Since there are total 51 cards (13 cards for each shape), the model could check if the shape of 2 cards are same by using the result of division by 13. Similarly, the model is able to check if 2 cards have same number by using the modulo of 13.

```
forAll i : RANGE .
    forAll j : LENGTH .
        j < movesTo[i]
        ->
        states[i + 1, j] = states[i, j],

forAll i : RANGE .
    states[i + 1, movesTo[i]] = states[i, movesFrom[i]],
```

When the model makes moves, the model should update the positions of all cards. The first constraint fills all positions that are located at left of the destination pile of previous move with same cards. For example, if the first row of the states is “1 2 4 5”, and the model moved

5, and removed 4, then the second row of the states should be “1 2 5”. In this case, first 2 positions did not changed, since the position of those 2 cards are left of the position of the destination pile. And the second constraint in the image above removes the destination pile, and replace that position with the source pile.

```
forAll i : RANGE .
    forAll j : RANGE .
        (j > movesTo[i] /\ moves[i] = 1)
        ->
        states[i + 1, j] = states[i, j + 1],

forAll i : RANGE .
    forAll j : RANGE .
        (j > movesTo[i] /\ j < movesFrom[i] /\ moves[i] = 3)
        ->
        states[i + 1, j] = states[i, j],


forAll i : RANGE .
    forAll j : LENGTH .
        (moves[i] = 3 /\ movesFrom[i] < (n - (i - 1)) /\ j <= (n - (i - 1)) /\ j > movesFrom[i])
        ->
        states[i + 1, j - 1] = states[i, j],
```

When the model moved the card to the adjacent position, then the model should move every card (all cards that are on the right side of the destination pile) to left. This is done by the first constraint in the image above.

The second and third constraints in the image above updates the states when the model moved the source pile to the position that is 3 positions away from the source pile. The third constraint does the similar thing with the first constraint, however, the third constraint moves the cards that are on the right side of the source pile.

The second constraint fills all positions that are located between the source pile and the destination pile with the same values in the previous state. For example, if the current state is “1 31 35 27”, then the model will move the card 27 to the first column, and remove the card 1. Then, the next state will be “27 31 35”, where the 31 and 35 did not changed. This is done by the second constraint in the image above.

3. Evaluation

3 - 1. Example Solutions

1) Solution for LBS5_0

```
language ESSENCE' 1.0
$ Minion SolverNodes: 6
$ Minion SolverTotalTime: 0.002807
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.369
letting moves be [1, 1, 1, 1]
letting movesFrom be [3, 2, 2, 2]
letting movesTo be [2, 1, 1, 1]
letting states be [[44, 29, 31, 5, 4],
[44, 31, 5, 4, -1],
[31, 5, 4, -1, -1],
[5, 4, -1, -1, -1],
[4, -1, -1, -1, -1]]
```

2) Solution for LBS7_47

```
language ESSENCE' 1.0
$ Minion SolverNodes: 38
$ Minion SolverTotalTime: 0.005165
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.492
letting moves be [1, 1, 3, 1, 1, 1]
letting movesFrom be [5, 5, 5, 3, 2, 2]
letting movesTo be [4, 4, 2, 2, 1, 1]
letting states be [[50, 9, 45, 18, 5, 44, 6],
[50, 9, 45, 5, 44, 6, -1],
[50, 9, 45, 44, 6, -1, -1],
[50, 6, 45, 44, -1, -1, -1],
[50, 45, 44, -1, -1, -1, -1],
[45, 44, -1, -1, -1, -1, -1],
[44, -1, -1, -1, -1, -1, -1]]
```

3) Solution for LBS9_0

```
language ESSENCE' 1.0
$ Minion SolverNodes: 1042
$ Minion SolverTotalTime: 0.03742
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.662
letting moves be [1, 1, 1, 3, 1, 1, 1, 1]
letting movesFrom be [8, 7, 6, 5, 3, 2, 2, 2]
letting movesTo be [7, 6, 5, 2, 2, 1, 1, 1]
letting states be [[44, 29, 31, 5, 15, 27, 41, 28, 8],
[44, 29, 31, 5, 15, 27, 28, 8, -1],
[44, 29, 31, 5, 15, 28, 8, -1, -1],
[44, 29, 31, 5, 28, 8, -1, -1, -1],
[44, 28, 31, 5, 8, -1, -1, -1, -1],
[44, 31, 5, 8, -1, -1, -1, -1, -1],
[31, 5, 8, -1, -1, -1, -1, -1, -1],
[5, 8, -1, -1, -1, -1, -1, -1, -1],
[8, -1, -1, -1, -1, -1, -1, -1, -1]]
```

3 - 2. Empirical Evaluation

For the Empirical Evaluation, I did 2 testings - 1) run the Savile Row with all given param files, and 2) run the Savile Row with larger n and suitable time out value.

3 - 2 - 1. Test all given param files

According to the specification, the key features for the Empirical Evaluation are SolverNodes, SolverSolveTime, and SavileRowTotalTime. These are available in the corresponding ".info" file.

Basically, I created a csv file called "output.csv" that has 8 columns : cards (the number of cards, which is n), seed, SolverSolveTime, SolverNodes, SolverSolutionsFound, SavileRowTotalTime, optimization, and option. The main reason that it contains columns like "optimization" and "option" is because this csv file contains the results of different optimization levels and "-all-solutions" option. These will be covered in the Extension section.

To visualise the csv file, I wrote a simple python script called "evaluate.py" by using pandas. The instruction about how to run this script will be mentioned in the Appendix section, so

please read the Appendix before run the python files. The image below is the output of the “evaluate.py” file.

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
4	13	0.000342	1	1	0.295	None	None
4	27	0.001318	3	1	0.302	None	None
4	33	0.001277	4	1	0.325	None	None
5	0	0.004277	6	1	0.322	None	None
5	30	0.004706	13	1	0.322	None	None
5	44	0.022300	80	1	0.004778	None	None
6	12	0.000700	0	0	0.334	None	None
6	20	0.000662	0	0	0.378	None	None
7	20	0.000751	0	0	0.533	None	None
7	21	0.005237	200	0	0.526	None	None
7	47	0.008801	42	1	0.001333	None	None
8	8	0.027875	529	0	1.023	None	None
8	20	0.023363	623	0	0.859	None	None
8	47	0.018559	257	1	0.008457	None	None
9	0	0.048573	1042	1	0.035719	None	None
9	12	0.075066	1941	1	0.159656	None	None
9	16	0.212430	5892	1	0.388	None	None
9	21	0.352429	6641	1	0.611	None	None
9	40	0.033426	687	1	0.577	None	None
9	41	0.023034	244	1	0.74	None	None
10	12	0.639905	17804	1	0.768	None	None
10	16	0.087242	1985	1	0.856	None	None
10	19	0.082181	1429	1	0.807	None	None
10	21	1.608380	40474	1	0.547	None	None
10	39	0.320675	7879	1	0.602	None	None

As you could see above, for the solvable problems, the number of “SolverNodes” increases dramatically when the number of cards is increasing. For the parameters with 4 cards, the Savile Row only creates less than 5 nodes. However, for the parameters that have more than 10 cards, the value of “SolverNodes” is greater than 1,400. Furthermore, for the parameters like 10_12 and 10_21, the number of nodes are greater than 10,000. Unlike solvable problems, the number of solver nodes for the unsolvable problems is pretty small. For example, the number of solver nodes for unsolvable parameters that have 6 cards is 0, where the number of solver nodes for solvable parameters that have 5 cards, which is less than 6, is greater than 5 (6, 13, and 80).

Also, as you could see in the image above, the values of SolverSolveTime and SavileRowTotalTime are proportional to the SolverNodes. For example, the value of SolverNodes for L9_12.param is 1941, and the value of SolverNodes for L9_16.param is 5892. Since the L9_16.param’s value of SolverNodes is greater than L9_12.param’s, the SolverSolveTime value and the SavileRowTotalTime value of L9_16.param are greater than L9_12.param’s.

3 - 2 - 2. Larger n with suitable time out value

The main aim of this section is to find the biggest n value that the model could be able to deliver a solution in a practical amount of time. I was also tried to write a python script that executes Savile Row with -solver-options “cpulimit <timeout>”, however, for some reason, the subprocess does not work with the “cpulimit <timeout>”. Thus, I tested the files manually.

In the submission directory, you could find a directory called “test”. This directory contains the param files and output files (“.info”, “.minion”, “.infor”, “.solution”).

First, I run the Savile Row to check if my model could deliver a solution within 10 minutes by using the -sover-options “cpulimit 600” (600 seconds = 10 minutes).

1) n = 11, timeout = 10 minutes

	SolverNodes	SolverSolveTime	SavileRowTotalTime
LBS11_0	108279	3.07712	1.046
LBS11_4	92428	3.04338	1.072

2) n = 12, timeout = 10 minutes

	SolverNodes	SolverSolveTime	SavileRowTotalTime
LBS12_0	516617	14.2892	1.301
LBS12_30	248277	6.55223	1.113
LBS12_33	656051	19.3413	1.704
LBS12_42	436329	13.2534	1.729
LBS12_47	632125	19.042	0.981

3) n = 13, timeout = 10 minutes

	SolverNodes	SolverSolveTime	SavileRowTotalTime
LBS13_0	4548245	107.25	0.877
LBS13_3	1021819	23.4666	0.832
LBS13_4	495400	11.9444	0.839
LBS13_5	10543883	255.03	0.753
LBS13_9	7160260	345.871	1.031
LBS13_13	985264	25.8058	1.072
LBS13_21	3023110	82.8431	0.694
LBS13_23	285372	10.0621	1.097
LBS13_44	512322	13.7552	0.757
LBS13_51	960299	26.0101	0.946

For all n <= 13, regardless of seed value, my model was able to find solution (or discover that the given parameter is unsolvable) within 10 minutes. However, from n=14, the model was not able to deliver a solution within 10 minutes.

When I tested my model with LBS14_20, I found that the timeout was occurred. Henceforth, the maximum n value that my model could be able to deliver a solution in 10 minutes is 13.

Just for curious, I wanted to check what would happen if I set the timeout value with 1 hour rather than 10 minutes.

4) n = 14, timeout = 1 hour (3600 seconds)

	SolverNodes	SolverSolveTime	SavileRowTotalTime	SolverTimeOut
LBS14_20	155334312	3551.18	1.147	yes
LBS14_24	6619190	174.314	1.051	no

As you could see above, the model gets the timeout for 14_24 even if I use 1 hour as a timeout value. Again, it would be possible to say that the maximum n value that my model could be able to deliver a solution in 10 minutes is 13.

4. Extension

4 - 1. Different Optimisation levels

The Savile Row provides 4 optimisation levels: -O0, -O1, -O2, and -O3. According to the Savile Row manual, the optimisation levels provide an easy way to control how much optimisation Savile Row does, without switching on and off the individual optimisations. The default optimisation level is 2, which provides the most generally recommended optimisation sets.

-O0:

Turns off all optional optimisations

-O1:

-O1 does the optimisations that are very efficient for both time and space.
Variables that are equal are unified
A form of common subexpression elimination is applied.

-O2:

Provides all optional optimisations that are done by -O1.
Performs filtering variable domains and aggregation.

-O3:

Provides all optional optimisations that are done by -O2.
Enables tabulation and associative-commutative common subexpression elimination.

As I mentioned above (in the Evaluation section), the “output.csv” file contains outputs for different optimisation levels. Thus, you could also print out the results of all optimisation levels by using the “evaluate.py”. Again, please read the Appendix section before run the program.

```
(cards,seed)=(4, 13)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
    4    13         0.000342            1                     1              0.295         None   None
    4    13         0.008839            1                     1              0.193           O0   None
    4    13         0.004639            1                     1              0.211           O1   None
    4    13         0.000249            1                     1              0.272           O2   None
    4    13         0.000236            1                     1              0.312           O3   None

(cards,seed)=(4, 27)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
    4    27         0.001318            3                     1              0.302         None   None
    4    27         0.011573            5                     1               0.18           O0   None
    4    27         0.004572            3                     1              0.235           O1   None
    4    27         0.001932            3                     1              0.365           O2   None
    4    27         0.005879            3                     1              0.328           O3   None
```

(cards,seed)=(4, 33)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
4	33	0.001277	4	1	0.325	None	None
4	33	0.007485	5	1	0.186	O0	None
4	33	0.002843	4	1	0.214	O1	None
4	33	0.002152	4	1	0.343	O2	None
4	33	0.000164	5	1	0.657	O3	None

(cards,seed)=(5, 0)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
5	0	0.004277	6	1	0.322	None	None
5	0	0.013021	7	1	0.249	O0	None
5	0	0.009165	6	1	0.352	O1	None
5	0	0.004145	6	1	0.409	O2	None
5	0	0.000203	7	1	1.93	O3	None

(cards,seed)=(5, 30)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
5	30	0.004706	13	1	0.322	None	None
5	30	0.015298	27	1	0.22	O0	None
5	30	0.007376	13	1	0.286	O1	None
5	30	0.003179	13	1	0.402	O2	None
5	30	0.001255	25	1	2.869	O3	None

(cards,seed)=(5, 44)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
5	44	0.022300	80	1	0.004778	None	None
5	44	0.014966	80	1	0.294	O0	None
5	44	0.007018	29	1	0.264	O1	None
5	44	0.004823	29	1	0.418	O2	None
5	44	0.001914	80	1	2.151	O3	None

(cards,seed)=(7, 47)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
7	47	0.008801	42	1	0.001333	None	None
7	47	0.047527	345	1	0.33	O0	None
7	47	0.021230	42	1	0.331	O1	None
7	47	0.010768	42	1	0.575	O2	None
7	47	0.005396	81	1	3.109	O3	None

(cards,seed)=(8, 47)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
8	47	0.018559	257	1	0.008457	None	None
8	47	0.680320	7248	1	0.301	O0	None
8	47	0.033457	257	1	0.435	O1	None
8	47	0.022193	257	1	0.577	O2	None
8	47	0.019860	568	1	2.898	O3	None

(cards,seed)=(9, 0)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
9	0	0.048573	1042	1	0.035719	None	None
9	0	11.215600	146539	1	0.382	O0	None
9	0	0.055395	1042	1	0.367	O1	None
9	0	0.057620	1042	1	0.739	O2	None
9	0	0.105355	2450	1	4.141	O3	None

(cards,seed)=(9, 12)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
9	12	0.075066	1941	1	0.159656	None	None
9	12	6.945090	103798	1	0.475	O0	None
9	12	0.105751	1939	1	0.432	O1	None
9	12	0.059514	1939	1	0.575	O2	None
9	12	0.232101	5238	1	3.608	O3	None

(cards,seed)=(9, 16)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
9	16	0.212430	5892	1	0.388	None	None
9	16	7.684590	87079	1	0.271	O0	None
9	16	0.144574	3639	1	0.448	O1	None
9	16	0.167535	3639	1	0.61	O2	None
9	16	0.520856	12097	1	3.02	O3	None

(cards,seed)=(9, 21)

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
9	21	0.352429	6641	1	0.611	None	None
9	21	45.255600	465308	1	0.393	O0	None
9	21	0.302845	6614	1	0.36	O1	None
9	21	0.284683	6614	1	0.59	O2	None
9	21	0.699894	18928	1	2.753	O3	None

```
(cards,seed)=(9, 40)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
    9    40         0.033426          687                     1              0.577         None   None
    9    40         4.251810        41340                     1              0.403           O0   None
    9    40         0.066648          687                     1              0.564           O1   None
    9    40         0.031909          687                     1               0.57           O2   None
    9    40         0.058611         1733                     1              3.257           O3   None

(cards,seed)=(9, 41)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
    9    41         0.023034          244                     1               0.74         None   None
    9    41         0.813342         6170                     1              0.834           O0   None
    9    41         0.041325          244                     1              0.363           O1   None
    9    41         0.014016          244                     1               0.37           O2   None
    9    41         0.020169          559                     1              2.921           O3   None

(cards,seed)=(10, 12)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
   10    12         0.639905        17804                     1              0.768         None   None
   10    12       140.533000      1776529                     1               0.32           O0   None
   10    12         0.637893        17804                     1              0.539           O1   None
   10    12         0.611584        17804                     1              0.681           O2   None
   10    12         2.322630        55613                     1              3.993           O3   None

(cards,seed)=(10, 16)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
   10    16         0.087242         1985                     1              0.856         None   None
   10    16        20.779900       209008                     1              0.581           O0   None
   10    16         0.089694         1985                     1              0.491           O1   None
   10    16         0.088353         1985                     1                0.7           O2   None
   10    16         0.291802         4910                     1              3.765           O3   None

(cards,seed)=(10, 19)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
   10    19         0.082181         1429                     1              0.807         None   None
   10    19        31.884100       369101                     1              0.635           O0   None
   10    19         0.076081         1429                     1              0.379           O1   None
   10    19         0.065344         1429                     1              0.666           O2   None
   10    19         0.134336         3842                     1              3.335           O3   None
```

```
(cards,seed)=(10, 21)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
   10    21          1.60838        40474                     1              0.547         None   None
   10    21       1574.47000     14129850                     1              0.326           O0   None
   10    21          1.54844        40474                     1              0.356           O1   None
   10    21          1.82874        40474                     1              0.652           O2   None
   10    21          4.34134       116706                     1              2.706           O3   None

(cards,seed)=(10, 39)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization option
   10    39         0.320675         7879                     1              0.602         None   None
   10    39       154.451000      1626207                     1             0.0.37           O0   None
   10    39         0.371039         7991                     1              0.384           O1   None
   10    39         0.336504         7879                     1              0.751           O2   None
   10    39         0.677851        21771                     1              3.438           O3   None
```

As the images above shown, the evaluate.py loads the data, and group the loaded by all combination of cards(=n) and seed. Then, it prints out each grouped dataframe. The optimization column depicts the applied optimization level. As you could see, in general, the -O1 option solves the problem within the shortest time. The values of SolverSolveTime, SolverNodes, and SavileRowTotalTime of -O1 are usually less than the values of -O2 optimisation level. I assume this is because the some of the optimisation options that is provided by the -O2 and -O3 make the solver to waste more time.

Thus, I would say that for the Late Binding Solitaire problems, the -O1 is the best optimisation level (at least for my model).

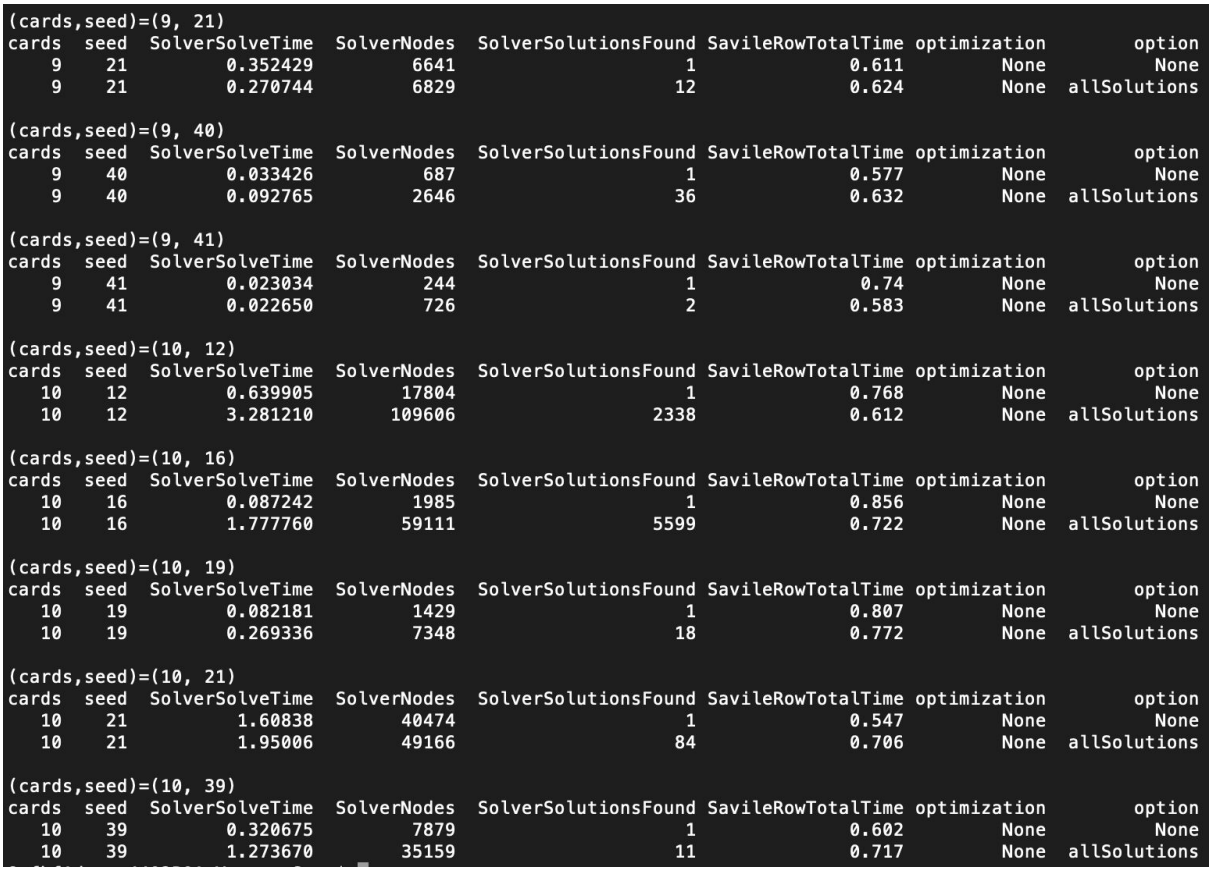
4 - 2. Evaluation for -all-solutions flag

The Savile Row has a flag called “-all-solutions”, which causes the solver to search for all solutions, and make the Savile Row to parse all solutions. I was pretty sure that some parameters might have multiple solutions, thus, I applied the -all-solutions flag to all given solvable files. Moreover, I was curious what would happen if I use the -all-solutions flag for the unsolvable parameters. Thus, I also tested with all given unsolvable files.

Again, all outputs were recorded in the “output.csv” file, thus, the “evaluate.py” file is used for printing out the results.

```
(cards,seed)=(4, 13)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    4    13         0.000342            1                     1              0.295         None          None
    4    13         0.000342            1                     1              0.295         None  allSolutions

(cards,seed)=(4, 27)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    4    27         0.001318            3                     1              0.302         None          None
    4    27         0.001599            5                     3              0.312         None  allSolutions

(cards,seed)=(4, 33)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    4    33         0.001277            4                     1              0.325         None          None
    4    33         0.001659            4                     8              0.364         None  allSolutions

(cards,seed)=(5, 0)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    5     0         0.004277            6                     1              0.322         None          None
    5     0         0.003757           22                     2              0.358         None  allSolutions
```

```
(cards,seed)=(5, 30)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    5    30         0.004706           13                     1              0.322         None          None
    5    30         0.005399           20                     1              0.388         None  allSolutions

(cards,seed)=(5, 44)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    5    44         0.022300           80                     1           0.004778         None          None
    5    44         0.003179           30                     1              0.317         None  allSolutions

(cards,seed)=(7, 47)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    7    47         0.008801           42                     1           0.001333         None          None
    7    47         0.010804          289                    16              0.486         None  allSolutions

(cards,seed)=(8, 47)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    8    47         0.018559          257                     1           0.008457         None          None
    8    47         0.086421         2326                    62              0.505         None  allSolutions

(cards,seed)=(9, 0)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9     0         0.048573         1042                     1           0.035719         None          None
    9     0         0.298810         8251                     2              0.602         None  allSolutions

(cards,seed)=(9, 12)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9    12         0.075066         1941                     1           0.159656         None          None
    9    12         0.407326        13017                   338              0.668         None  allSolutions

(cards,seed)=(9, 16)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9    16         0.212430         5892                     1              0.388         None          None
    9    16         0.254986         5819                    36              0.652         None  allSolutions
```



```
(cards,seed)=(9, 21)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9    21         0.352429         6641                     1              0.611         None          None
    9    21         0.270744         6829                    12              0.624         None  allSolutions

(cards,seed)=(9, 40)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9    40         0.033426          687                     1              0.577         None          None
    9    40         0.092765         2646                    36              0.632         None  allSolutions

(cards,seed)=(9, 41)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
    9    41         0.023034          244                     1               0.74         None          None
    9    41         0.022650          726                     2              0.583         None  allSolutions

(cards,seed)=(10, 12)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
   10    12         0.639905        17804                     1              0.768         None          None
   10    12         3.281210       109606                  2338              0.612         None  allSolutions

(cards,seed)=(10, 16)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
   10    16         0.087242         1985                     1              0.856         None          None
   10    16         1.777760        59111                  5599              0.722         None  allSolutions

(cards,seed)=(10, 19)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
   10    19         0.082181         1429                     1              0.807         None          None
   10    19         0.269336         7348                    18              0.772         None  allSolutions

(cards,seed)=(10, 21)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
   10    21          1.60838        40474                     1              0.547         None          None
   10    21          1.95006        49166                    84              0.706         None  allSolutions

(cards,seed)=(10, 39)
cards  seed  SolverSolveTime  SolverNodes  SolverSolutionsFound SavileRowTotalTime optimization        option
   10    39         0.320675         7879                     1              0.602         None          None
   10    39         1.273670        35159                    11              0.717         None  allSolutions
```

As the images above shown, the evaluate.py loads the data, and group the loaded by all combination of cards(=n) and seed. Then, it prints out each grouped dataframe. Each grouped dataframe has 2 rows, where the first row is the result without applying the -all-solutions flag, and the second row is the result with the -all-solutions flag.

After running the Savile Row with the -all-solutions, I was surprised that most solvable parameter files have multiple solutions, where some of them has more than 1000 solutions. Especially, if you see the result of 10_16, you could find that the solver found 5599 solutions. The most interesting was that the longest solver solve time was 1.95006 seconds (10_19), where the Savile Row took more than 30 seconds to find a single solution with the -O0 optimisation level.

Due to the time limit, I was not able to test the -all-solutions flag with different optimisation levels. However, I am pretty sure that it would be worth to try it.

5. Conclusion

While doing this practical, I was able to learn how to design the constraint models. At the beginning, I had hard times for implementing the constraints, because unlike other general programming languages, the Essence Prime does not provide debugging. As I was not able to print things, it was really hard to understand what is actually going on.

After spending a lot of time for reading the description file and running the example files, I was able to understand what to do. It was really tough practical, however, at the same time it was pretty interesting practical.

6. Appendix

If you are using the lab machine, you might need to set up the virtualenv to install required python packages. First use “python3 -m venv venv” to create the virtualenv.
Then use either “. venv/bin/activate” or “source venv/bin/activate” to execute the virtualenv.
Then, install all packages in the requirements.txt file by using “pip3 install -r requirements.txt”.

To execute the “test_all_files.py” file, you need to input 3 command line arguments.

```
Usage: python3 test_all_files.py <file_path_of_savile_row> <1 or 2> <1 or 2>
```

The first argument is the path of the Savile Row directory. For example, if you use “../savilerow-1.8.0-linux” as a first command line argument, then the program will try to find the executable file “../savilerow-1.8.0-linux/savilerow”.

The second argument should be either 1 or 2. If it is 1, then the program will create or remove output files for all param files in the “solvable” directory. Otherwise, if the second argument is 2, then the program will create or remove output files for all param files in the “unsolvable” directory.

The third argument should be either 1 or 2. If it is 1, then it will create output files for all param files in the target directory (either “solvable” or “unsolvable” - depends on the second argument). Otherwise, it will remove all output files in the target directory.

For example, if you use “python3 ./savilerow_dir 1 1”, then the “test_all_files.py” will create subprocesses to execute “./savilerow_dir/savilerow ./main.eprime <param_file> -run-solver” for all param files in the “solvable” directory. And if you use “python3 ./savilerow_dir 1 2”, then the “test_all_files.py” will remove all generated output files in the “solvable” directory. You could do the same things with the “unsolvable” directory by replacing the second command line argument with 2.

To execute the “evaluate.py” file, you need to pass 1 command line argument.

```
Usage: python3 evaluate.py <mode_num>
       mode_num should be one of 1 ~ 3
```

The first command line argument should be one of 1, 2, and 3.

If you use 1, then the program will print out the dataframe that contains the results of all param files in the both “unsolvable” directory and “solvable” directory. Below is the result of “python3 evaluate.py 1”.

cards	seed	SolverSolveTime	SolverNodes	SolverSolutionsFound	SavileRowTotalTime	optimization	option
4	13	0.000342	1	1	0.295	None	None
4	27	0.001318	3	1	0.302	None	None
4	33	0.001277	4	1	0.325	None	None
5	0	0.004277	6	1	0.322	None	None
5	30	0.004706	13	1	0.322	None	None
5	44	0.022300	80	1	0.004778	None	None
6	12	0.000700	0	0	0.334	None	None
6	20	0.000662	0	0	0.378	None	None
7	20	0.000751	0	0	0.533	None	None
7	21	0.005237	200	0	0.526	None	None
7	47	0.008801	42	1	0.001333	None	None
8	8	0.027875	529	0	1.023	None	None
8	20	0.023363	623	0	0.859	None	None
8	47	0.018559	257	1	0.008457	None	None
9	0	0.048573	1042	1	0.035719	None	None
9	12	0.075066	1941	1	0.159656	None	None
9	16	0.212430	5892	1	0.388	None	None
9	21	0.352429	6641	1	0.611	None	None
9	40	0.033426	687	1	0.577	None	None
9	41	0.023034	244	1	0.74	None	None
10	12	0.639905	17804	1	0.768	None	None
10	16	0.087242	1985	1	0.856	None	None
10	19	0.082181	1429	1	0.807	None	None
10	21	1.608380	40474	1	0.547	None	None
10	39	0.320675	7879	1	0.602	None	None

If you use 2 as a command line argument, then the program will print out the grouped dataframes where each dataframe is grouped by combination of cards and seed. Each dataframe contains 2 rows, where the first row is the result without applying the -all-solutions flag, and the second row is the result with the -all-solutions flag.

If you use 3 as a command line argument, then the program will print out the grouped dataframes, where each dataframe is grouped by combination of cards and seed values. Those grouped dataframes are the results for the “different optimisation level” testing, which is mentioned in the seciton 4 - 1.

The “test” directory contains the param files with larger n (n > 10), and the result files. And the main.eprime is the file that contains the constraint model.