Constraint Processing and Programming: Solitaire Assignment and Past Deliveries Report

Denis Ergashbaev

December 7, 2014

1 Past Deliveries

The following table lists the locations of the past deliveries and provides instructions of executing the code. The SAT assignments were developed with python 2.7 in Pycharm IDE while the CP assignments were done in Qt Creator. Both python and C++ projects can be run from the shell.

Table 1: SAT Assignments

Assignment	Location	Command
3-Coloring Problem	`cpp/SAT/3-coloring`	`PYTHONPATH=../ python main-3-coloring.py`
DPLL Own SAT-Solver	`cpp/SAT/dpll`	`PYTHONPATH=../ python main-dpll.py`
Sudoku	`cpp/SAT/sudoku`	`PYTHONPATH=../ python main-sudoku.py`
Solitaire	`cpp/SAT/solitaire`	`PYTHONPATH=../ python main-solitaire.py`

Table 2: CP Assignments

Assignment	Location	Command
Football Match Scheduling	`cpp/CP/FootballMatchScheduling/FootballMatchScheduling`	`./compila.sh main.cpp ./main.cpp`
Football Match Scheduling Additional restriction: 4 first teams cannot play against each other	`cpp/CP/FootballMatchScheduling Windowing /FootballMatchScheduling Windowing`	`./compila.sh main.cpp ./main.cpp`
Knights Tour Problem	`cpp/CP/KnightsTourProblem /KnightsTourProblem`	`./compila.sh main.cpp ./main.cpp`
Group Sports Scheduling	`cpp/CP/GroupSportsScheduling/GroupSportsScheduing`	`./compila.sh main.cpp ./main.cpp`
Solitaire	`cpp/CP/Solitaire/Solitaire`	`./compila.sh main.cpp ./main.cpp PATH_TO_FILE`

2 Solitaire

2.1 SAT

The code for the SAT version of the Solitaire can be found under `cpp/SAT/solitaire` and run by executing the `PYTHONPATH=../ python main-solitaire.py` command. The respective solutions for each of the inputs are merged into a single file `cpp/SAT/solitaire/solutions/summary`. The generated CNF files are solved by the minisat with a time limit set to 10 minutes.

2.1.1 Description of the Model

Output Format
In the file `summary` the the cards are identified by the pair $x : y$, where x indicates the rank and y points at the suit of the card. The card $0 : 0$ is assumed to be the first card on the stack and is not included in the solution output.

Variables
The variables and clauses are defined in the file `main-solitaire.py`.

The model is defined by the boolean variables $X_{i,j}$, where i represents a card and j one of all the moves (valid and invalid) that the card could take within the course of the game.

Each card is uniquely identified by its *card_index* produced by the mapping function $mapping() := num\_ranks * suit + rank$. The range of j is defined as $\{1, .., number\_of\_cards\}$ Thus, the set of values that the variables $X_{i,j}$ can take is defined by $\{0, .., i * j\}$, whereas $X_{i,j} = 0$ is a preassigned value that is taken by the first card on the deck.

Clauses
The model contains following clauses.

- $\forall j : X_{1,j} \vee .. \vee X_{n-1,j} \vee X_{n,j}$ – there should be at least one card played at each turn.
- $\forall j, \forall i : \neg X_{i,j} \vee \neg X_{i+1,j}, .., \neg X_{i,j} \vee \neg X_{n-1,j}, \neg X_{i,j} \vee \neg X_{n,j}$ – there can only be one card played at a time.
- $\forall i, \forall j : \neg X_{i,1} \vee \neg X_{i,2}, .., \neg X_{i,1} \vee \neg X_{i,n}, \neg X_{i,1} \vee \neg X_{i,n}$ – a single card can only be played once.
- $\forall j, \forall i : |rank(X_{i,j}) - rank(X_{i,j+1})| = 1$ *or* $(min(rank(X_{i,j}), rank(X_{i,j+1})) = 0$ *and* $max(rank(X_{i,j}), rank(X_{i,j+1})) = num\_ranks - 1)$ – the stack may only contain the valid rank moves (as defined in `Card.py#is_valid_neighbor()`) between cards with consecutive positions. There is a special treatment for the top card of each pile as they may only be dealt if there is no conflict with the first card.
- $\forall$ *card_pile* $\in \{Card\_Piles\}, \forall i \in \{card\_pile\}, \forall j : X_{i,j} \vee \neg X_{i+1,j+1}, .., X_{i,j} \vee \neg X_{i+1,n-1}, X_{i,j} \vee \neg X_{i+1,n}$ – only open cards can be dealt: for instance, if the first card of a given card pile has not been put on stack yet, the cards underneath it may not be put on stack.

Results
The results are displayed in Table 3. As can be seen, two benchmarks do not have a solution (solit_4_4_4.txt and solit_4_4_6.txt), all the solutions have been produced within the timeout that was set.

2.2 Constraint Programming

The code for the constraint programming version of the Solitaire can be found under `cpp/CP/Solitaire`. The output vector V represents a mapping from a card to its position on the final stack. The card is represented by the index of the element in the V whereas its position is the value that the element at a given index takes.

As in the SAT implementation, the very first card (number: 0, position in the deck: 0) is not included in the calculated solution, as it is assumed to be placed in the deck. Thus the size of V is equal to $num\_suits * num\_ranks - 1$ and the domain of the variables is restricted by $\{0, .., num\_suits * num\_ranks - 2\}$.

Constraints
There are only three constraints that are initially imposed in the program.

1. $distinct(V)$ - the values (that is position of the respective card) must be distinct
2. $\forall$ *pile in* $\{Piles\}, \forall$ *cardIndex in* $\{Pile\} : V(cardIndex) < V(cardIndex + 1)$ - Each card of a given card pile can only be placed on the stack if the card above it is already on the stack.

3. $\forall\ i\ in\ \{0,..,|V|\}:\ isValidNeighbor(V(i),\ V(i+1))$ - two cards can be only placed on the stack consecutively after each other only if they are valid neighbors ("rank jump" of 1). Similarly to the SAT implementation, the function $isValidNeighbor()$ is defined as $|rank(card\_index) - rank(other\_card\_index)| = 1\ or\ (min(card\_index), rank(other\_card\_index)) = 0\ and\ max(rank(card\_index), rank(other\_card\_index)) = num\_ranks - 1)$.

   As it is also the case in the SAT implementation, there is a special treatment for the top card of each pile as they may only be dealt if there is no conflict with the first card.

Improvements of the Model
However, as can be seen in the original results in Table 4, the performance of the model was very poor – although it correctly solved all the 4_4_*.txt instances, it could not find a solution under the set time limit for any of the 4_7_*.txt files.

I have tried a number of approaches to try to resolve this issue, this includes the following:

1. Experimenting with various branching strategies (commented out in the code):

   `branch(*this, V, INT_VAR_NONE(), INT_VAL_MIN())` - that was the default option

   `branch(*this, V, INT_VAR_SIZE_MIN(), INT_VAL_MIN())`

   `branch(*this, V, INT_VAR_MERIT_MIN(&mer), INT_VAL(&v));`

   `branch(*this, V, tiebreak(INT_VAR_SIZE_MIN(), INT_VAR_RND(r)), INT_VAL_RND(r))`

2. Adding additional constraints to limit the search space: if we break each pile consisting of 3 cards into the three levels, then we can additionally constraint the model to limit the value range for the top, middle, and bottom cards. For instance, the top cards, may not have the very last two positions, because these are taken by the cards below them.

In the end, the combination that works best for some of the 4_7_*.txt instances is some heuristics-based branching. I instruct the Gecode to prefer branching based on the custom merit function, whereas the tie breaks are resolved by preferring the variables with maximum constrains imposed on them. The merit function evaluates the level of the card in its pile. Intuition tells, that the upper the card is on the pile, the more likely considering this card first is a good strategy.

Results
The original unoptimized results are provided in table 4. I have written a bash script that executes the Gecode module for each of the benchmark files. If the process takes more than 5 minutes it is being stopped by the script. The output of the files is written in the file *solutions/solution*.

The results obtained after introducing random branching strategy and additional constraints on the cards depending on their level in the pile are listed in Table 5. As can be seen, the only gain was that one more instance could be solved at the expense of unpredictability due to randomness.

Finally, the results of applying a merit-based branching strategy are listed in **??**. These demonstrate the best performance of the model, which is able to solve more problems.

Table 3: SAT Results

	Input File	Satisfiable	Time (sec.)
1	solit_4_4_0.txt	SATISFIABLE	0.011471
2	solit_4_4_1.txt	SATISFIABLE	0.015458
3	solit_4_4_2.txt	SATISFIABLE	0.015459
4	solit_4_4_3.txt	SATISFIABLE	0.014513
5	solit_4_4_4.txt	UNSATISFIABLE	0.005386
6	solit_4_4_5.txt	SATISFIABLE	0.013613
7	solit_4_4_6.txt	UNSATISFIABLE	0.005463
8	solit_4_4_7.txt	SATISFIABLE	0.010882
9	solit_4_4_8.txt	SATISFIABLE	0.010554
10	solit_4_4_9.txt	SATISFIABLE	0.015062
11	solit_4_7_0.txt	SATISFIABLE	0.05225
12	solit_4_7_1.txt	SATISFIABLE	0.042535
13	solit_4_7_2.txt	SATISFIABLE	0.037909
14	solit_4_7_3.txt	SATISFIABLE	0.031724
15	solit_4_7_4.txt	SATISFIABLE	0.04215
16	solit_4_7_5.txt	SATISFIABLE	0.057432
17	solit_4_7_6.txt	SATISFIABLE	0.040548
18	solit_4_7_7.txt	SATISFIABLE	0.048621
19	solit_4_7_8.txt	SATISFIABLE	0.039247
20	solit_4_7_9.txt	SATISFIABLE	0.050305
21	solit_4_10_0.txt	SATISFIABLE	0.145706
22	solit_4_10_1.txt	SATISFIABLE	0.340935
23	solit_4_10_2.txt	SATISFIABLE	0.489555
24	solit_4_10_3.txt	SATISFIABLE	0.502095
25	solit_4_10_4.txt	SATISFIABLE	0.262237
26	solit_4_10_5.txt	SATISFIABLE	0.160292
27	solit_4_10_6.txt	SATISFIABLE	0.188978
28	solit_4_10_7.txt	SATISFIABLE	0.154226
29	solit_4_10_8.txt	SATISFIABLE	0.146695
30	solit_4_10_9.txt	SATISFIABLE	0.166091
31	solit_4_13_0.txt	SATISFIABLE	4.948247
32	solit_4_13_1.txt	SATISFIABLE	11.510692
33	solit_4_13_2.txt	SATISFIABLE	29.003128
34	solit_4_13_3.txt	SATISFIABLE	10.131731
35	solit_4_13_4.txt	SATISFIABLE	26.169205
36	solit_4_13_5.txt	SATISFIABLE	0.485495
37	solit_4_13_6.txt	SATISFIABLE	64.72407
38	solit_4_13_7.txt	SATISFIABLE	9.0399
39	solit_4_13_8.txt	SATISFIABLE	1.067569
40	solit_4_13_9.txt	SATISFIABLE	1.306856

Table 4: CP Results (unoptimized) – file: *solution_model1*

	Input File	Satisfiable	Time (sec.)
1	solit_4_4_0.txt	SATISFIABLE	0.061
2	solit_4_4_1.txt	SATISFIABLE	0.006
3	solit_4_4_2.txt	SATISFIABLE	0.035
4	solit_4_4_3.txt	SATISFIABLE	0.036
5	solit_4_4_4.txt	UNSATISFIABLE	1.965
6	solit_4_4_5.txt	SATISFIABLE	0.087
7	solit_4_4_6.txt	UNSATISFIABLE	1.646
8	solit_4_4_7.txt	SATISFIABLE	0.038
9	solit_4_4_8.txt	SATISFIABLE	0.005
10	solit_4_4_9.txt	SATISFIABLE	0.036
11	solit_4_7_0.txt	TIMEOUT	
12	solit_4_7_1.txt	TIMEOUT	
13	solit_4_7_2.txt	TIMEOUT	
14	solit_4_7_3.txt	TIMEOUT	
15	solit_4_7_4.txt	TIMEOUT	
16	solit_4_7_5.txt	TIMEOUT	
17	solit_4_7_6.txt	TIMEOUT	
18	solit_4_7_7.txt	TIMEOUT	
19	solit_4_7_8.txt	TIMEOUT	
20	solit_4_7_9.txt	TIMEOUT	
21	solit_4_10_0.txt	TIMEOUT	
22	solit_4_10_1.txt	TIMEOUT	
23	solit_4_10_2.txt	TIMEOUT	
24	solit_4_10_3.txt	TIMEOUT	
25	solit_4_10_4.txt	TIMEOUT	
26	solit_4_10_5.txt	TIMEOUT	
27	solit_4_10_6.txt	TIMEOUT	
28	solit_4_10_7.txt	TIMEOUT	
29	solit_4_10_8.txt	TIMEOUT	
30	solit_4_10_9.txt	TIMEOUT	
31	solit_4_13_0.txt	TIMEOUT	
32	solit_4_13_1.txt	TIMEOUT	
33	solit_4_13_2.txt	TIMEOUT	
34	solit_4_13_3.txt	TIMEOUT	
35	solit_4_13_4.txt	TIMEOUT	
36	solit_4_13_5.txt	TIMEOUT	
37	solit_4_13_6.txt	TIMEOUT	
38	solit_4_13_7.txt	TIMEOUT	
39	solit_4_13_8.txt	TIMEOUT	
40	solit_4_13_9.txt	TIMEOUT	

Table 5: CP Results (random branching) – file: *solution_model2*

		Satisfiable	Time (sec.)
1	solit_4_4_0.txt	SATISFIABLE	0.064
2	solit_4_4_1.txt	SATISFIABLE	0.054
3	solit_4_4_2.txt	SATISFIABLE	0.241
4	solit_4_4_3.txt	SATISFIABLE	0.02
5	solit_4_4_4.txt	UNSATISFIABLE	0.223
6	solit_4_4_5.txt	SATISFIABLE	0.043
7	solit_4_4_6.txt	UNSATISFIABLE	0.184
8	solit_4_4_7.txt	SATISFIABLE	0.038
9	solit_4_4_8.txt	SATISFIABLE	0.095
10	solit_4_4_9.txt	SATISFIABLE	0.055
11	solit_4_7_0.txt	TIMEOUT	
12	solit_4_7_1.txt	SATISFIABLE	0.306
13	solit_4_7_2.txt	TIMEOUT	
14	solit_4_7_3.txt	TIMEOUT	
15	solit_4_7_4.txt	TIMEOUT	
16	solit_4_7_5.txt	TIMEOUT	
17	solit_4_7_6.txt	TIMEOUT	
18	solit_4_7_7.txt	TIMEOUT	
19	solit_4_7_8.txt	TIMEOUT	
20	solit_4_7_9.txt	TIMEOUT	
21	solit_4_10_0.txt	TIMEOUT	
22	solit_4_10_1.txt	TIMEOUT	
23	solit_4_10_2.txt	TIMEOUT	
24	solit_4_10_3.txt	TIMEOUT	
25	solit_4_10_4.txt	TIMEOUT	
26	solit_4_10_5.txt	TIMEOUT	
27	solit_4_10_6.txt	TIMEOUT	
28	solit_4_10_7.txt	TIMEOUT	
29	solit_4_10_8.txt	TIMEOUT	
30	solit_4_10_9.txt	TIMEOUT	
31	solit_4_13_0.txt	TIMEOUT	
32	solit_4_13_1.txt	TIMEOUT	
33	solit_4_13_2.txt	TIMEOUT	
34	solit_4_13_3.txt	TIMEOUT	
35	solit_4_13_4.txt	TIMEOUT	
36	solit_4_13_5.txt	TIMEOUT	
37	solit_4_13_6.txt	TIMEOUT	
38	solit_4_13_7.txt	TIMEOUT	
39	solit_4_13_8.txt	TIMEOUT	
40	solit_4_13_9.txt	TIMEOUT	

Table 6: CP Results (merit-based function) – file: *solution_model3*

	Input File	Satisfiable	Time (sec.)
1	solit_4_4_0.txt	SATISFIABLE	0.005
2	solit_4_4_1.txt	SATISFIABLE	0.006
3	solit_4_4_2.txt	SATISFIABLE	0.006
4	solit_4_4_3.txt	SATISFIABLE	0.007
5	solit_4_4_4.txt	UNSATISFIABLE	0.127
6	solit_4_4_5.txt	SATISFIABLE	0.008
7	solit_4_4_6.txt	UNSATISFIABLE	0.127
8	solit_4_4_7.txt	SATISFIABLE	0.005
9	solit_4_4_8.txt	SATISFIABLE	0.008
10	solit_4_4_9.txt	SATISFIABLE	0.007
11	solit_4_7_0.txt	TIMEOUT	
12	solit_4_7_1.txt	SATISFIABLE	0.017
13	solit_4_7_2.txt	TIMEOUT	
14	solit_4_7_3.txt	TIMEOUT	
15	solit_4_7_4.txt	TIMEOUT	
16	solit_4_7_5.txt	TIMEOUT	
17	solit_4_7_6.txt	SATISFIABLE	0.01
18	solit_4_7_7.txt	TIMEOUT	
19	solit_4_7_8.txt	TIMEOUT	
20	solit_4_7_9.txt	TIMEOUT	
21	solit_4_10_0.txt	TIMEOUT	
22	solit_4_10_1.txt	TIMEOUT	
23	solit_4_10_2.txt	TIMEOUT	
24	solit_4_10_3.txt	TIMEOUT	
25	solit_4_10_4.txt	TIMEOUT	
26	solit_4_10_5.txt	TIMEOUT	
27	solit_4_10_6.txt	TIMEOUT	
28	solit_4_10_7.txt	TIMEOUT	
29	solit_4_10_8.txt	TIMEOUT	
30	solit_4_10_9.txt	TIMEOUT	
31	solit_4_13_0.txt	TIMEOUT	
32	solit_4_13_1.txt	TIMEOUT	
33	solit_4_13_2.txt	TIMEOUT	
34	solit_4_13_3.txt	TIMEOUT	
35	solit_4_13_4.txt	TIMEOUT	
36	solit_4_13_5.txt	TIMEOUT	
37	solit_4_13_6.txt	TIMEOUT	
38	solit_4_13_7.txt	TIMEOUT	
39	solit_4_13_8.txt	TIMEOUT	
40	solit_4_13_9.txt	TIMEOUT	