Modelling for Combinatorial Optimisation (1DL451) and Part 1 of Constraint Programming (1DL442) Uppsala University – Autumn 2022 Report for Assignment 1 by Team 1

Andreas JONASSON and Marion DÜBENDORFER

8th September 2022

All experiments were run under Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with a 70 GB RAM and an 8 MB L3 cache (a ThinLinc computer of the IT department).

Problem 1
Solving Technologies

Backend	Solving Technology
Gecode	Constraint Programming (CP)[1]
CP-SAT	Constraint Programming (CP) and Boolean Satisfiability (SAT)[2]
Gurobi	Linear Programming (LP) and Mixed Integer Programming (MIP)[3]
Yuck	Constraint-Based Local Search (CBLS)[4]
PicatSAT	Boolean Satisfiability (SAT)[5]

Problem 2
The Nine Children Problem

A Model

Our model, with the prescribed comments, has the imposed name `nineChildren.mzn`, is uploaded, and is given in Listing 1.

Constraints Enforced by the Choice of Decision Variables. The constraint that no person is younger than 0 years or older than 150 years is enforced by the tight domains of our decision variables.

Efficiency. In violation of checklist item number 10, we make use of the `pow` functionality inside a constraint. We make use of this functionality because, to our knowledge, there is no linear reformulation in order to get the square of a number.

Listing 1: A MiniZinc model for the The Nine Children problem

```
include "globals.mzn";

% number of children:
int: n = 9;
% array of integers representing the children's ages:
array[1..n] of var 0..150: Children;
% integer representing the parent's age:
var 1..150: parent;

% find solution:
solve satisfy;
% find minimal solution
%solve minimize parent;

% ensure children are not born at the same time:
constraint all_different(Children);
% omit symmetric solutions:
constraint forall(i in 1..n-1)(Children[i+1] > Children[i]);
% ensure uniform age gap between children:
constraint forall(i in 1..n-2)(Children[i+1] - Children[i] =
  Children[i+2] - Children[i+1]);
% ensure squared age of parent equals sum of squared ages of
  children:
constraint pow(parent,2) = sum([pow(Children[i],2) | i in 1..n]);
```

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
	`Parent`	time	`Parent`	time	`Parent`	time	`Parent`	time	`Parent`	time
	48	790	48	**570**	48	2420	48	7170	??	t/o

Table 1: Results for our model of The Nine Children Problem, which is a constraint satisfaction problem. The 'time' column reports the time needed to find a satisfying solution, if one was found before the time-out (120,000 milliseconds here), else the time-out is indicated by 't/o'. The 'parent' column reports the age of the parent, if a solution was found, else '??'.

Parent	Children
48	2, 5, 8, 11, 14, 17, 20, 23, 26
96	4, 10, 16, 22, 28, 34, 40, 46, 52
144	6, 15, 24, 33, 42, 51, 60, 69, 78

Table 2: The ages for the parent and children for all found solutions.

B Evaluation

Table 1 gives the results for `n=9` in our model. The time-out was 120,000 milliseconds. We observe that Gecode wins overall, as it has the shortest runtime with `n=9`.

C Solutions

As seen in Table 2, multiple solutions to the problem exist. There are several ways to find out if there is more than one solution. When solving, we can use the `-a` flag to print all solutions. We can also solve to minimize the parent's age first, and then solve again to maximize the parent's age. If the resulting solutions differ, we know that there are multiple solutions.

In order to determine a solution with the youngest parent, instead of solving to satisfy, we could solve to minimize the parent's age.

Problem 3
The Halmos and his Wife Problem

A Model

Our model, with the prescribed comments, has the imposed name `wifeHalmos.mzn`, is uploaded, and is given in Listing 2.

Constraints Enforced by the Choice of Decision Variables. None of the constraints of the problem are automatically enforced by our choice of decision variables, so we have explicitly modelled all the constraints of the problem.

Efficiency. The model features no violations of any pieces of advice in the checklists of the final slides of Topic 2.

Listing 2: A MiniZinc model for the The Halmos and his Wife problem

```
include "globals.mzn";
% Number of couples:
int: n = 5;
% Generate a column and row for each person.
% Handshakes[i,j] = 1 if person i shook person j's hand.
% Couples are represented consecutively in each array; Halmos has index 1, and his wife has index 2.
array[1..2*n, 1..2*n] of var 0..1: Handshakes;
% The number of hands Halmos's wife shook
var 0..(2*n)-2: wifeHands;

solve satisfy;

% Halmos's wife's handshake count is equal to the sum of the second row in Handshakes
constraint wifeHands = sum(Handshakes[2,..]);
% No person shakes their own hand, therefore, diagonals contain only 0:
constraint forall(i in 1..2*n)(Handshakes[i,i] = 0);
% Ensure symmetry of Handshakes (i.e. if person j shook person i's hand, person i also shook person j's hand:
constraint forall(i,j in 1..2*n)(Handshakes[i,j] = Handshakes[j,i]);
% No person shakes their partner's hand:
constraint forall(i in 1..n)(Handshakes[2*i, 2*i-1] = 0);
% Each person shakes a different number of hands (except for Halmos):
constraint all_different([sum(Handshakes[i,..]) | i in 2..2*n]);

% Part C: Check if there is any other solution than 4
% constraint sum(Handshakes[2,..]) != 4;
```

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
	shakes	time	shakes	time	shakes	time	shakes	time	shakes	time
	4	2860	4	**490**	4	580	4	630	4	6850

Table 3: Results for our model of the Halmos and his Wife problem, which is a constraint satisfaction problem. The 'time' column reports the time needed to find a satisfying solution. All backends solved the problem before the time-out (120,000 milliseconds here) and all found a solution where Halmos's wife shakes hand with four people. Boldface indicates the best time.

B Evaluation

Table 3 gives the results for `n=10` (with `n` the number of attending people) in our model. The timeout was 120,000 milliseconds. We can observe that Gecode has the shortest runtime with 490ms, followed by Gurobi with 580ms.

C Uniqueness

First, we know that 4 is a possible solution to the problem according to Table 3. Therefore, to check for different solutions, we define a constraint which prevents 4 from being a solution (see Line 19 of Listing 2). If the model is run with this additional constraint, the problem becomes unsatisfiable. This implies that 4 is the only possible number of handshakes for Halmos' wife.

Nonetheless, there is another aspect to the range of solutions. Even though the number 4 is a unique solution with respect to how many hands Halmos' wife shook, there can still exist multiple solution *instances*. More precisely, various permutations of `n x n` (and in this case `n=10`) matrices that satisfy the problem can exist, and it is possible that different backends find different matrices as first solution.

Problem 4
The Largest Permutation Problem

A Model

Our model, with the prescribed comments, has the imposed name `largestPerm.mzn`, is uploaded, and is given in Listing 3.

Constraints Enforced by the Choice of Decision Variables. None of the constraints of the problem are automatically enforced by our choice of decision variables, so we have explicitly modelled all the constraints of the problem.

Efficiency. The model features no violations of any pieces of advice in the checklists of the final slides of Topics 2 (and 3).

B Evaluation

Table 4 gives the results for various values of parameter n in our model. The time-out was 60,000 milliseconds.

Listing 3: A MiniZinc model for the The Largest Permutation problem

```
include "globals.mzn";

% The length of the permutation:
int: n;

% Array containing the permutation:
array[1..n] of var 1..n: Permutation;

% Each integer in the range 1..n appears exactly once on the
  permutation:
constraint all_different(Permutation);

% Maximize the sum of all pairs of successive values:
solve maximize sum([ Permutation[i] * Permutation[i+1] | i in
  1..n-1]);
```

It is noteworthy that no backend is able to prove solutions optimal for $n \geq 15$. That being said, even though Yuck times out for each n, it is able to find the highest objective values overall compared to all of the other backends. Therefore, it can be considered to win overall, even though it times out for small n already.

In terms of scalability, with respect to proving optimality of a solution, CP-SAT, Gecode, and Gurobi perform similarly, timing out with $n \geq 15$, while PicatSAT times out for $n \geq 10$ and Yuck times out for $n \geq 5$. With respect to maximising the objective value, Yuck scales best for $n \geq 15$.

For $n \leq 10$ the runtimes indicate that the difficulty of instances increase monotonically with their size. Additionally, it can be observed that for each backend, the maximal objective values monotonically increase with n.

Boolean satisfiability (SAT) does not seem to be a suitable method for this problem, as PicatSAT can prove optimality only for $n = 5$ and finds the lowest objective value for all other n compared to the other backends. The combination of boolean satisfiability (SAT) and constraint programming (CP) seems to perform better, as CP-SAT outperforms Gecode, Gurobi and PicatSAT with respect to finding the maximal objective value. In general, local search should be considered an unsuitable method for this problem, as it can not prove maxima by construction. This is reflected in the fact that Yuck times out for all n. However, Yuck does outperform all the other backends when it comes to finding the maximal objective value, and can therefore still be considered suitable.

The proven optima are consistent in all backends, and therefore there are no contradicting results. There were no occurrences of `'ERR'` generated by the experiment script.

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
n	obj	time	obj	time	obj	time	obj	time	obj	time
5	**46**	960	**46**	**580**	**46**	590	**46**	740	**46**	t/o
10	**366**	33530	**366**	**2780**	**366**	36840	354	t/o	**366**	t/o
15	1201	t/o	1191	t/o	1198	t/o	931	t/o	**1211**	t/o
20	2776	t/o	2721	t/o	2795	t/o	2243	t/o	**2831**	t/o
25	5356	t/o	5188	t/o	5360	t/o	4063	t/o	**5476**	t/o
30	9186	t/o	8863	t/o	9257	t/o	7623	t/o	**9396**	t/o
35	14516	t/o	13980	t/o	14414	t/o	11172	t/o	**14841**	t/o
40	21596	t/o	20813	t/o	21598	t/o	15395	t/o	**22061**	t/o
45	30676	t/o	29588	t/o	30360	t/o	25948	t/o	**31306**	t/o
50	41985	t/o	40563	t/o	41968	t/o	33873	t/o	**42826**	t/o
55	55790	t/o	53988	t/o	55726	t/o	43446	t/o	**56869**	t/o
60	72365	t/o	70113	t/o	72407	t/o	57440	t/o	**73689**	t/o
65	91940	t/o	89188	t/o	91758	t/o	69447	t/o	**93534**	t/o
70	114762	t/o	111463	t/o	114310	t/o	86067	t/o	**116651**	t/o
75	141080	t/o	137188	t/o	140982	t/o	109975	t/o	**143290**	t/o
80	171093	t/o	166613	t/o	170640	t/o	130450	t/o	**173719**	t/o
85	205161	t/o	199955	t/o	204680	t/o	164954	t/o	**208162**	t/o
90	243400	t/o	237563	t/o	242970	t/o	192134	t/o	**246881**	t/o
95	286215	t/o	279588	t/o	285760	t/o	206838	t/o	**290117**	t/o
100	333777	t/o	326134	t/o	333300	t/o	264847	t/o	**338134**	t/o

Table 4: Results for our model of The Largest Permutation problem, which is a maximisation problem. The column called `n` denotes the length of the permutation. In the `time` column, if the reported time is less than the time-out (60,000 milliseconds here), then the reported area in the 'obj' column was proven optimal; else the time-out is indicated by 't/o' and the reported area is the best found, but not proven optimal solution before timing out. Boldface indicated the best performance (time or objective value) on each row.

Problem 5

The Tile Packing Problem

A Model

Our model, with the prescribed comments, has the imposed name `tilePacking.mzn`, is uploaded, and is given in Listing 4.

Constraints Enforced by the Choice of Decision Variables. None of the constraints of the problem are automatically enforced by our choice of decision variables, so we have explicitly modelled all the constraints of the problem.

Efficiency The model features no violations of any pieces of advice in the checklists of the final slides of Topics 2 and 3.

Listing 4: A MiniZinc model for the The Tile Packing problem

```
include "globals.mzn";

% Number of tiles:
int: n;
% The side length of each tile:
array[1..n] of int: side = 1..n;
% The maximal side lengths the resulting rectangle can obtain
% is the sum of all the lengths of the tiles:
int: maxDim = sum(1..n);
% The width of the bounding rectangle
var 1..maxDim: w;
% The height of the bounding rectangle
var 1..maxDim: h;

% Array for the x-coordinates of the tiles:
array[1..n] of var 1..maxDim: x;
% Array for the y-coordinates of the tiles:
array[1..n] of var 1..maxDim: y;

% Ensure that no tiles are overlapping:
constraint diffn(x,y,side,side);

% For each tile of width i, it holds that the x-coordinate of the
  tile
% plus the width of the tile is within the width of the resulting
  rectangle:
constraint w = max([x[i] + i | i in 1..n]);
% The same holds for the y-coordinate of each tile
% and the height of the resulting rectangle:
constraint h = max([y[i] + i | i in 1..n]);

% For part C:
% Place the largest tile in the lower-left quadrant of the bounding
  rectangle,
% in order to avoid symmetric solutions
constraint x[n] * 2 < w;
constraint y[n] * 2 < h;

% minimize width and height of the bounding rectangle
solve minimize w*h;
```

B Evaluation

Table 5 gives the results for various values of the parameter n in our model. The time-out was 60,000 milliseconds. We observe that CP-SAT wins overall, as it achieves the best result among the backends for all n. Gurobi also performs well, as it is only beat by CP-SAT for $n = 15$. These two scale the best, as they achieve comparatively good results and start timing out for greater n than the other backends. For instances $n \leq 8$, all backends find the optimal solution and for $n \in \{5, 6\}$ Gecode is the fastest.

For CP-SAT, we have that $n = 4$ is more difficult to solve than $n = 5$. The same applies for Gecode, where also $n = 6$ is easier to solve than $n = 5$. For all other n, and for all other backends, the difficulty of instances increases monotonically with the size of n. With the exception of Gecode for $n = 13$, the found area also monotonically increases with the size of n. In the Gecode case, the solving for $n \geq 9$ always times out, so the found solutions are not guaranteed to be optimal, and we see from e.g. CP-SAT that there are better slutions where the area increases monotonically.

Local search seems to be a poor method for this problem, as Yuck for $n \geq 11$ can not find as good solutions as Gurobi and CP-SAT. No results are contradictory, as the proven optima are the same for all backends. No occurrences of 'ERR' were generated by the experiment script.

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
n	w, h, obj	time	w, h, obj	time	w, h, obj	time	w, h, obj	time	w, h, obj	time
4	6, 8, **48**	1460	8, 6, **48**	570	6, 8, **48**	**530**	6, 8, **48**	730	6, 8, **48**	t/o
5	6, 13, **78**	510	13, 6, **78**	**500**	6, 13, **78**	570	6, 13, **78**	1060	13, 6, **78**	t/o
6	10, 12, **120**	540	12, 10, **120**	**480**	10, 12, **120**	630	12, 10, **120**	4560	10, 12, **120**	t/o
7	12, 15, **180**	**620**	15, 12, **180**	1810	12, 15, **180**	1040	12, 15, **180**	12510	15, 12, **180**	t/o
8	15, 16, **240**	**700**	16, 15, **240**	19710	16, 15, **240**	1110	16, 15, **240**	13720	16, 15, **240**	t/o
9	16, 21, **336**	**2089**	19, 18, 342	t/o	21, 16, **336**	2150	16, 21, **336**	34690	19, 18, 342	t/o
10	16, 28, **448**	4380	28, 16, **448**	t/o	16, 28, **448**	**2310**	28, 16, **448**	t/o	16, 28, **448**	t/o
11	20, 28, **560**	13010	32, 18, 576	t/o	28, 20, **560**	**7160**	21, 27, 567	t/o	29, 20, 580	t/o
12	24, 30, **720**	t/o	34, 57, 1938	t/o	24, 30, **720**	**27020**	30, 24, **720**	58420	34, 22, 748	t/o
13	23, 39, **897**	t/o	47, 20, 940	t/o	39, 23, **897**	t/o	24, 39, 936	t/o	27, 35, 945	t/o
14	24, 46, **1104**	t/o	51, 69, 3519	t/o	46, 24, **1104**	t/o	20, 75, 1500	t/o	39, 31, 1209	t/o
15	24, 56, **1344**	t/o	55, 82, 4510	t/o	41, 33, 1353	t/o	39, 35, 1365	t/o	29, 51, 1479	t/o

Table 5: Results for our model of The Tile Packing Problem, which is a minimisation problem. In the 'time' column, if the reported time is less than the time-out (60,000 milliseconds here), then the reported area in the 'obj' column was *proven* optimal; else the time-out is indicated by 't/o' and the reported area is the best found but not proven optimal before timing out. Boldface indicated the best performance (time or objective value) on each row.

C Symmetry-Breaking Constraint

A reflection or rotation of any solution results in a symmetrical solution. By placing the largest tile in the lower-left quadrant, we break these symmetries. The symmetry-breaking constraints can be seen in lines 31-32 in Listing 4. The model with this constraint was evaluated in the same way as the original model, and the results are given in Table 6. The effect of the constraint varies among backends and values of n. For CP-SAT, the same areas are found, but the time needed is longer or shorter without a clear pattern. Gecode finds the same areas while taking less time for the instances without a time-out. For Gurobi, the time is longer for all instances except $n \in \{7, 8, 9\}$ and worse areas are found for $n \in \{14, 15\}$. PicatSAT finds worse areas for $n \in \{13, 15\}$ and better for $n = 14$ while being faster for $n \in \{7, 8\}$, otherwise slower. Yuck finds better solutions for $n \in \{9, 12, 13\}$ and worse for $n \in \{14, 15\}$. In summary, the added constraint improves the performance of Gecode, while the other backends see a mix of gains and losses.

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
n	w, h, obj	time	w, h, obj	time	w, h, obj	time	w, h, obj	time	w, h, obj	time
4	6, 8, **48**	760	8, 6, **48**	**440**	8, 6, **48**	540	8, 6, **48**	800	8, 6, **48**	t/o
5	6, 13, **78**	490	13, 6, **78**	**450**	6, 13, **78**	620	6, 13, **78**	1150	13, 6, **78**	t/o
6	10, 12, **120**	570	12, 10, **120**	**450**	10, 12, **120**	650	10, 12, **120**	4490	10, 12, **120**	t/o
7	12, 15, **180**	580	15, 12, **180**	**480**	15, 12, **180**	880	12, 15, **180**	6520	15, 12, **180**	t/o
8	15, 16, **240**	710	16, 15, **240**	**610**	16, 15, **240**	1040	15, 16, **240**	16559	16, 15, **240**	t/o
9	16, 21, **336**	**1330**	19, 18, 342	t/o	21, 16, **336**	1570	21, 16, **336**	35490	21, 16, **336**	t/o
10	16, 28, **448**	4470	28, 16, **448**	t/o	16, 28, **448**	**2490**	16, 28, **448**	36980	16, 28, **448**	t/o
11	20, 28, **560**	12330	32, 18, 576	t/o	20, 28, **560**	**7780**	27, 21, 567	t/o	20, 29, 580	t/o
12	24, 30, **720**	t/o	34, 57, 1938	t/o	24, 30, **720**	**40600**	24, 30, **720**	t/o	24, 32, 768	t/o
13	23, 39, **897**	t/o	47, 20, 940	t/o	39, 23, **897**	t/o	25, 40, 1000	t/o	33, 29, 957	t/o
14	24, 46, **1104**	t/o	51, 69, 3519	t/o	31, 36, 1116	t/o	51, 23, 1173	t/o	38, 31, 1178	t/o
15	24, 56, **1344**	t/o	55, 82, 4510	t/o	24, 57, 1368	t/o	111, 31, 3441	t/o	27, 53, 1431	t/o

Table 6: Results for our model of The Tile Packing Problem, with a symmerty-breaking constraint. In the 'time' column, if the reported time is less than the time-out (60,000 milliseconds here), then the reported area in the 'obj' column was *proven* optimal; else the time-out is indicated by 't/o' and the reported area is the best found but not proven optimal before timing out. Boldface indicated the best performance (time or objective value) on each row.

Feedback to the Teachers

First of all, the fact that this assignment is pass/fail helps to get a look and feel on how to solve the following assignments without being too concerned about the grade of this particular assignment. In addition, the overall difficulty of the problems was appropriate, with some being a bit more tricky than others. For the following years, the first assignment might benefit from a more strict and/or precise task formulation. For example, an indication for whether the questions in a task should be answered by implementing it in code or whether describing the solution in text is enough would make some tasks more clear, and helps getting an intuition for the following assignments.

References

[1] Gecode Team. Gecode: A generic constraint development environment, 2020. The Gecode solver and its MiniZinc backend are available at `https://www.gecode.org`.

[2] Google Optimization Tools. Cp-sat solver. `https://developers.google.com/optimization/cp/cp_solver`, 2022. The CP-SAT solver is available at `https://github.com/google/or-tools`.

[3] Gurobi Optimization, LLC. Gurobi optimizer. `https://www.gurobi.com/products/gurobi-optimizer/`, 2022.

[4] informarte. Yuck. `https://github.com/informarte/yuck`, 2022.

[5] Neng-Fa Zhou. Picat. `http://picat-lang.org/`, 2022.