Modelling for Combinatorial Optimisation (1DL451) and Part 1 of Constraint Programming (1DL442) Uppsala University – Autumn 2022 Report for Assignment 2: Spacecraft Assembly Problem by Team 1

Andreas JONASSON and Marion DÜBENDORFER

23rd 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).

A Pre-computed Values

In order to render our model more efficient and make it more readable, we pre-compute the derived parameters `DueWeek` as well as `Type`. In addition, in order to tighten the domains of the variables `totalStorageCost` and `totalSetupCost`, we derive the maximal storage cost and setup cost. The derived values can be found in Listing 1.

Listing 1: Derived parameters for SAP

```
% The maximal possible setup cost, used to tighten the domain of totalSetupCost:
int: maxSetupCost = weeks * max(SetupCost);
% The maximal possible storage cost, used to tighten the domain of totalStorageCost:
int: maxStorageCost = (weeks*(weeks-1))*storageCost div 2;
% DueWeek[s] is the week that spacecraft s is due:
array[1..spacecrafts] of int: DueWeek = [ w | t in 1..types, w in 1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];
% Type[s] is the type of spacecraft s:
array[1..spacecrafts] of int: Type = [ t | t in 1..types, w in 1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];
```

B Models

B.1 Viewpoint 1

Our model for viewpoint 1, with the prescribed comments, has the imposed name `SAP1.mzn`, is uploaded, and is given in Listing 2.

Listing 2: A MiniZinc model for viewpoint 1 of the Spacecraft Assembly problem

```
include "globals.mzn";

%----Parameters----
int: weeks; % #weeks for the planning
int: types; % #types of spacecraft
% Order[t,w] = #spacecrafts of type t to assemble by end of week w:
array[1..types,1..weeks] of int: Order;
int: storageCost; % cost of storing one spacecraft during one week
% SetupCost[t1,t2] = cost of adapting factory from type t1 to t2:
array[1..types,1..types] of int: SetupCost;

%----Derived parameters----
int: spacecrafts = sum(Order); % total #spacecrafts to assemble
% The maximal possible setup cost, used to tighten the domain of
  totalSetupCost:
int: maxSetupCost = weeks * max(SetupCost);
% The maximal possible storage cost, used to tighten the domain of
  totalStorageCost:
int: maxStorageCost = (weeks*(weeks-1))*storageCost div 2;
% DueWeek[s] is the week that spacecraft s is due:
array[1..spacecrafts] of int: DueWeek = [ w | t in 1..types, w in
  1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];
% Type[s] is the type of spacecraft s:
array[1..spacecrafts] of int: Type = [ t | t in 1..types, w in
  1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];

%----Decision variables----

% Spacecraft[w] denotes which spacecraft (if any) to build in week
  w:
% 0 is used as dummy value. Therefore, Spacecraft[w] = 0 denotes
  that no spacecraft is built in week w.
array[1..weeks] of var 0..spacecrafts: Spacecraft;
% FactorySetup[w] denotes which type of spacecraft the factory is
  set up to build in week w.
% FactorySetup is non-mutually redundant with the array of
  variables Spacecraft:
array[1..weeks] of var 1..types: FactorySetup;
% If a spacecraft is assembled before its due week, a storageCost
  for every week it is not due incurs.
% Weeks where no spacecraft is assembled (i.e. Spacecraft[w] = 0)
  are ignored:
var 0..maxStorageCost: totalStorageCost =
  sum([(DueWeek[Spacecraft[w]] - w) * storageCost | w in 1..weeks
  where Spacecraft[w] != 0]);
% If the types of spacecrafts built in two consecutive weeks
  differ, a setup cost incurs:
```

```
37 var 0..maxSetupCost: totalSetupCost =
     sum([SetupCost[FactorySetup[w], FactorySetup[w+1]] | w in
     1..weeks-1 ]);
38
39 %----Constraints----
40 % Ensure that each spacecraft is built either before it is due or
     in the week it is due:
41 constraint forall(w in 1..weeks where Spacecraft[w] !=
     0)(DueWeek[Spacecraft[w]] >= w);
42 % Ensure that each spacecraft is built (and not more than once):
43 constraint global_cardinality(Spacecraft, 1..spacecrafts, [ 1 | s
     in 1..spacecrafts]);
44 % Ensure that the type of the spacecraft built in week w (if any)
     matches with the type according to the factory setup.
45 % This is a one-way channeling constraint from the array of
     variables Spacecraft to its redundant array of variables
     FactorySetup:
46 constraint forall(w in 1..weeks where Spacecraft[w] !=
     0)(Type[Spacecraft[w]] = FactorySetup[w]);
47
48 %----Objective----
49 % Minimize the total cost, consisting of totalStorageCost and
     totalSetupCost:
50 solve minimize totalStorageCost + totalSetupCost;
51
52 %----Output----
53 output [show(totalStorageCost + totalSetupCost)];
```

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.

Redundant Decision Variables and Channelling Constraints. The variable `FactorySetup` is non-mutually redundant with `Spacecraft`. This is useful because it is used to model the total setup cost. We do not know if this approach results in faster solving than an approach without redundant variables, as we never wrote a model for the latter using viewpoint 1.

Implied Constraints. We did not detect any implied constraints that we would consider useful. The model performed fairly well without implied constraints, which is why we decided not to pursue them due to time constraints.

Efficiency. In violation of checklist item 6, we use the expression `where Spacecraft[w] != 0`, with `Spacecraft` being an array of decision variables. Due to time constraints, we were not able to find a violation-free reformulation using this viewpoint. Given that with our model of viewpoint 2 we provide a model without this particular checklist violation, we consider this trade-off to be acceptable.

B.2 Evaluation Viewpoint 1

Table 1 gives the results for all given problem instances for our model using the first viewpoint. The time-out was 300,000 milliseconds. We observe that the backends do well on different instances. For the first five instances (`sap_005_02` to `sap_010_06`), all backends except Yuck find the optimal solutions, with Gecode being the fastest or near fastest, except for `sap_010_05`, where CP-SAT is the fastest. For the remaining instances, all backends time out. CP-SAT performs well for instances `sap_015_10` to `sap_100_15`, except for `sap_030_05` where Gurobi wins. For `sap_150_15` only Gecode and PicatSAT find a satisfying solution, with Gecode finding a more optimal one. The overall winning backends are thus CP-SAT and Gecode, as they both find optimal solutions for the smaller instances, CP-SAT finds better solutions for the bigger instances and Gecode finds the best solution for `sap_150_15`. Gecode and PicatSAT scale the best in terms of finding satisfying solutions for the greatest instances, while CP-SAT finds better solutions except for `sap_150_15` and thereby also scales well. For this model, Yuck scales poorly as it fails to find a solution for all but the first four instances. PicatSAT finds solutions for all but the last instance, but it is comparatively slow for the smallest five and finds worse solutions than e.g CP-SAT for the remaining ones.

The difficulty of an instance depends on the combination of number of weeks, types of spacecrafts, setup costs, storage cost and the order. Thus comparing for example `sap_005_02` and `sap_006_02`, the latter instance has one more week, while the former has an extra spacecraft in its order. The storage and setup costs also differ. In order to tell exactly what makes an instance difficult, one could create instances that only differ in one parameter and compare the results. However, in general, the larger instances are more difficult to solve, as the backends for the six largest instances all time out.

For this model local search seems unsuitable, as Yuck does not find any satisfying solutions for any but the first four instances and only finds an optimal solution for the first two. The remaining backends, using systematic search, perform much better.

No results are contradictory: all proven optima are the same. No occurrences of 'ERR' were generated by the experiment script.

B.3 Viewpoint 2

Our model for viewpoint 2, with the prescribed comments, has the imposed name `SAP2.mzn`, is uploaded, and is given in Listing 3.

Listing 3: A MiniZinc model for viewpoint 2 of the Spacecraft Assembly problem

```
include "globals.mzn";

%----Parameters----
int: weeks; % #weeks for the planning
int: types; % #types of spacecraft
% Order[t,w] = #spacecrafts of type t to assemble by end of week w:
array[1..types,1..weeks] of int: Order;
int: storageCost; % cost of storing one spacecraft during one week
% SetupCost[t1,t2] = cost of adapting factory from type t1 to t2:
array[1..types,1..types] of int: SetupCost;

%----Derived parameters----
int: spacecrafts = sum(Order); % total #spacecrafts to assemble
```

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
instance	obj	time	obj	time	obj	time	obj	time	obj	time
sap_005_02	**171**	875	**171**	647	**171**	**635**	**171**	689	**171**	t/o
sap_006_02	**280**	600	**280**	**501**	**280**	655	**280**	1544	**280**	t/o
sap_008_04	**431**	823	**431**	**575**	**431**	1116	**431**	7444	444	t/o
sap_010_05	**675**	**1506**	**675**	4118	**675**	2707	**675**	51631	1045	t/o
sap_010_06	**917**	1071	**917**	**606**	**917**	1162	**917**	16350	–	t/o
sap_015_10	**1486**	t/o	1849	t/o	1621	t/o	2255	t/o	–	t/o
sap_030_05	1531	t/o	1703	t/o	**1129**	t/o	1189	t/o	–	t/o
sap_030_10	**1986**	t/o	2790	t/o	1990	t/o	4403	t/o	–	t/o
sap_100_15	**2017**	t/o	11084	t/o	2500	t/o	5681	t/o	–	t/o
sap_150_15	–	t/o	**30894**	t/o	–	t/o	33367	t/o	–	t/o
sap_200_15	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o

Table 1: Results for our model of viewpoint 1 of the SAP, which is a minimisation problem, using the first viewpoint. In the 'time' column, if the reported time is less than the time-out (300,000 milliseconds here), then the reported total cost in the `obj` column was *proven* optimal; else the time-out is indicated by 't/o' and the reported total cost is either the best one found but *not* proven optimal before timing out, or '–' indicating that no feasible solution was found before timing out. Boldface indicates the best performance (time or objective value) on each row.

```
% the maximal possible setup cost, used to tighten the domains for totalSetupCost:
int: maxSetupCost = weeks * max(SetupCost);
% the maximal possible storage cost, used to tighten the domains of totalStorageCost:
int: maxStorageCost = (weeks*(weeks-1))*storageCost div 2;

% DueWeek[s] is the week that spacecraft s is due:
array[1..spacecrafts] of int: DueWeek = [ w | t in 1..types, w in 1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];
% Type[s] is the type of spacecraft s:
array[1..spacecrafts] of int: Type = [ t | t in 1..types, w in 1..weeks, amount in 1..Order[t, w] where Order[t, w] >= 0];

%----Decision variables----
% Week[s] denotes in which week spacecraft s is built:
array[1..spacecrafts] of var 1..weeks: Week;
% FactorySetup[w] denotes which type of spacecraft the factory is set up to build in week w.
% FactorySetup is non-mutually redundant with the array of variables Week:
array[1..weeks] of var 1..types: FactorySetup;

% If a spacecraft is assembled before its due week, it will incur a storage cost (of amount storageCost) for every week that it is not
```

```
    due:
34  var 0..maxStorageCost: totalStorageCost = sum([(DueWeek[s] -
    Week[s]) * storageCost | s in 1..spacecrafts]);
35  % If the types of spacecrafts built in two consecutive weeks
    differ, a setup cost incurs:
36  var 0..maxSetupCost: totalSetupCost =
    sum([SetupCost[FactorySetup[w], FactorySetup[w+1]]| w in
    1..weeks-1]);
37
38  %----Constraints----
39  % Each spacecraft is built exactly once:
40  constraint all_different(Week);
41  % Ensure that for each spacecraft s, s is assembled in a week
    before or in the due week:
42  constraint forall(s in 1..spacecrafts)(Week[s] <= DueWeek[s]);
43  % Ensure that for each spacecraft s, Type[s] matches with the type
    that the factory is set up to build for the week s is built.
44  % This is a one-way channelling constraint from the array of
    variables Week to its redundant array of variables FactorySetup:
45  constraint forall(s in 1..spacecrafts)(FactorySetup[Week[s]] =
    Type[s]);
46
47  %----Objective----
48  solve minimize totalStorageCost + totalSetupCost;
49
50  %----Output----
51  output [show(totalStorageCost + totalSetupCost)];
```

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.

Redundant Decision Variables and Channelling Constraints. Similar to viewpoint 1, the variable `FactorySetup` is non-mutually redundant with `Week`. This is useful because it is used to model the total setup cost. In addition, this approach yields faster solving for some instances and some backends (see Section D).

Implied Constraints. As is the case with viewpoint 1, we did not detect any implied constraints that we would consider useful. The model performed fairly well without implied constraints, which is why we decided not to pursue them due to time constraints.

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

B.4 Evaluation Viewpoint 2

Table 2 gives the results for all given problem instances for our model using the second viewpoint. The time-out was 300,000 milliseconds.

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
instance	obj	time	obj	time	obj	time	obj	time	obj	time
sap_005_02	**171**	844	**171**	642	**171**	**580**	**171**	731	**171**	t/o
sap_006_02	**280**	592	**280**	**544**	**280**	582	**280**	1147	**280**	t/o
sap_008_04	**431**	997	**431**	**504**	**431**	855	**431**	18299	**431**	t/o
sap_010_05	**675**	2041	**675**	**662**	**675**	3261	**675**	52910	**675**	t/o
sap_010_06	**917**	1050	**917**	**506**	**917**	982	**917**	13857	**917**	t/o
sap_015_10	1811	t/o	**1530**	t/o	1627	t/o	2456	t/o	1538	t/o
sap_030_05	1848	t/o	1298	t/o	**1119**	**26044**	**1119**	t/o	1149	t/o
sap_030_10	2485	t/o	1910	t/o	**1754**	t/o	4503	t/o	1774	t/o
sap_100_15	10449	t/o	5365	t/o	**2213**	t/o	5408	t/o	3227	t/o
sap_150_15	31387	t/o	–	t/o	–	t/o	30613	t/o	**20159**	t/o
sap_200_15	54577	t/o	–	t/o	–	t/o	53258	t/o	**33642**	t/o

Table 2: Results for our model of viewpoint 2 of the SAP, which is a minimisation problem, using the second viewpoint. In the 'time' column, if the reported time is less than the time-out (300,000 milliseconds here), then the reported total cost in the `obj` column was *proven* optimal; else the time-out is indicated by 't/o' and the reported total cost is either the best one found but *not* proven optimal before timing out, or '–' indicating that no feasible solution was found before timing out. Boldface indicates the best performance (time or objective value) on each row.

We observe that the backends do well on different instances. For the first five instances (`sap_005_02` to `sap_010_06`), all backends find the optimal solution, with Gecode being the fastest for most instances. For the next four instances (`sap_015_10` to `sap_100_15`), Gurobi finds the best solutions, with the exception of `sap_015_10`, where Gecode wins. For instances `sap_150_15` and `sap_200_15`), Yuck finds the best solutions. The overall winning backends are thus Gecode, Gurobi and Yuck, depending on which instances we observe. Yuck scales best, as it finds comparatively good solutions for the instances where Gurobi and Gecode fail to find one. CP-SAT and PicatSAT also find solutions for these instances, but less optimal ones. For the smaller instances Gecode and Gurobi perform better.

The difficulty of an instance depends on the combination of number of weeks, types of spacecrafts, setup costs, storage cost and the order. Thus comparing for example `sap_005_02` and `sap_006_02`, the latter instance has one more week, while the former has an extra spacecraft in its order. The storage and setup costs also differ. In order to tell exactly what makes an instance difficult, one could create instances that only differ in one parameter and compare the results. However, in general, the larger instances are more difficult to solve, as the backends for the six largest instances tend to time out.

Local search seems suitable for this problem using this model, as Yuck finds the best solutions among the backends for all inputs except the four between `sap_015_10` and `sap_100_15`. While systematic search can stop before the timeout and it outperforms Yuck for four instances, these backends also time out for greater instances and is greatly outperformed by Yuck for the last two. No results are contradictory: all proven optima are the same. No occurrences of 'ERR' were generated by the experiment script.

C Ease of Modelling and Understandability

With respect to ease of modelling, during the first two phases of the incremental modelling process (i.e. modelling the assembly schedule and then the storage costs), the difficulty of implementing viewpoint 1 and 2 was comparable. For both viewpoints, we were able to build correct models in reasonable time on the basis of the derived parameters. Finally, when modelling the setup costs for viewpoint 1, we had trouble finding a suitable way to handle dummy values when constructing variables and constraints in order to calculate the setup cost. This was not the case with viewpoint 2, since there was no need for dummy values and thus we were able to build a correct model (although not yet a very efficient one) quite easily. Only after some more consideration (and some inspiration from the `Warehouse Location Problem` case study in Topic 6) did we get the idea to use redundant variables and letting the minimisation handle the dummy values (i.e., model the weeks in which no spacecraft is assembled with type values such that the setup cost is minimised). We then used the same approach with redundant variables to make the model for viewpoint 2 more efficient.

Similarly, with respect to understandability, viewpoint 2 seems more intuitive and therefore more understandable than viewpoint 1. The absence of the dummy value and the formulation of the decision variables increases the readability of the constraints. The only thing that might be more intuitive with viewpoint 1 is the design of the redundant variables - in viewpoint 1, the redundant array `FactorySetup` has the same length `1..weeks` as the array `Spacecraft`, and both arrays denote *what* is built in week `w`. This is not the case in viewpoint 2.

D Model Improvements

In our first implementation of the second viewpoint, we used the `argsort` predicate to sort the types of the spacecrafts in chronological order, in order to calculate the total setup cost. In the final model, this was replaced by the redundant `FactorySetup` array and a channelling constraint. The results of the model using the `argsort` predicate for a selection of the instances are given in Table 3. Compared to our final model, the `argsort` implementation is faster for `sap_005_02`. It is significantly slower for `sap_010_05` and `sap_030_05`, with the exception of PicatSAT which performs better with the `argsort` implementation. For `sap_015_10` and `sap_030_05`, Yuck performs better with the `argsort` implementation.

In conclusion, the models perform differently for different instances, but we chose our final model over the one using the `argsort` predicate since it was in general faster, and had better performance for e.g. `sap_030_05`.

E Recommendation

Our recommendation for the factory manager is the model using viewpoint 2, with either Gurobi or Yuck as backend, depending on the size of the instances they need to solve and how quickly they need to do be solved. If the instances are similar to the instances between `sap_005_02` and `sap_100_15` in Table 2, Gurobi in general finds the best or close to the best solutions, comparing the results for all backends in both viewpoints. It is also the only backend that proves optimality for `sap_030_05` in 300,000 milliseconds. If the problem instances are of larger size, Yuck is more likely to find a solution, as it found solutions for all instances and having the most optimal solutions for the two greatest instances in Table 2. If the results are not time-sensitive, a longer timeout could be used and both backends should produce better solutions for greater instances. Gurobi may be able to prove optimality, if that is of importance.

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
instance	obj	time	obj	time	obj	time	obj	time	obj	time
sap_005_02	**171**	601	**171**	**498**	**171**	543	**171**	649	**171**	t/o
sap_010_05	**675**	12589	**675**	14374	**675**	**5709**	**675**	20955	**675**	t/o
sap_015_10	1779	t/o	2258	t/o	1496	t/o	1516	t/o	**1486**	t/o
sap_030_05	1768	t/o	2222	t/o	1129	t/o	**1119**	**76697**	**1119**	t/o

Table 3: Results for our model of the SAP, which is a minimisation problem, using the second viewpoint with the `argsort` predicate implementation, for a selection of the problem instances. In the 'time' column, if the reported time is less than the time-out (300,000 milliseconds here), then the reported total cost in the `obj` column was *proven* optimal; else the time-out is indicated by 't/o' and the reported total cost is either the best one found but *not* proven optimal before timing out, or '–' indicating that no feasible solution was found before timing out. Boldface indicates the best performance (time or objective value) on each row.

Viewpoint 2 also has the advantage that to us it was more intuitive, especially for modelling the setup cost. Since it is easier to understand, it will likely also be easier to maintain.

F Real-World Example of SAP

A real-world example of the spacecraft assembly problem can be any vehicle that is assembled in a factory and might have similar types or designs. For instance, a factory that assembles Boeing airplanes and supplies them to airline companies might be such a case.

Different airlines might place orders with overlapping due dates, so storage room to store assembled airplanes might be necessary. In addition, various models of Boeing's commercial airplanes, such as the 737, 747, or 787, are being built nowadays. Therefore, the factory would need to be able to switch its setup between those types.

Finally, depending on the model, it might be cheaper to produce a model early and store if for a long time rather than switching between factory setups multiple times. The problem then becomes an optimisation problem similar to the SAP.

Feedback to the Teachers

In this assignment, it was very interesting to model a problem that has potential real-world applications. The problem itself was quite difficult, especially because two different viewpoints had to be implemented and it took us quite some time to build a working model for one of them. However, the code skeleton with the suggested pre-computed parameters helped a lot. It was also interesting to see how the performance of a model changes when certain predicates or approaches as a whole are replaced with different ones.