Modelling for Combinatorial Optimisation (1DL451) and Part 1 of Constraint Programming (1DL442) Uppsala University – Autumn 2022 Report the Project by Team 11 number

Alexander ANDERSSON and Savvas GIORTSIS

14th October 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 Model

The *Interview Assignment Problem* is a problem provided on CSPLib, a library of constraint programming problems. The original title of the problem on the CSPLib website is - 062: Interview Assignment Problem (`https://www.csplib.org/Problems/prob062/`).

The *Interview Assignment Problem* addresses company interviews in a conference. The interviews are held by different companies with certain needs, and if those needs are not met, the conference hosts incur a disappointment cost. The needs regard the number of interviews that a specific company wants to have during the conference. An upper and a lower bound are given.

Students are the actors that are going to those interviews. All students have in anticipation of the conference given preferences to the hosts, deciding which companies they would rather want to interview. A score from 1 to 5 is given, where 1 is a strong preference for wanting an interview and 5 represents no interest whatsoever.

The *Interview Assignment Problem* is to match these students for interviews in these different companies, taking all the parameters above into consideration and providing a solution that satisfies all parties. Students want interviews with their best preferences, and the host wants to minimize any disappointment cost that may incur from the companies.

The model `day2.mzn` is shown below in `Listing 1`.

Listing 1: A MiniZinc model for the Interview Assignment Problem

```
4 include "globals.mzn";
5
6 %Number of companies:
7 int: nCompanies;
8 %Number of Students:
9 int: nStudents;
10
```

```
11 set of int: Students = 1..nStudents;
12 set of int: Companies = 1..nCompanies;
13 set of int: CompaniesAndDummy = 1..nCompanies union {dummy};
14 set of int: Preferences = 1..3;
15
16 % CompanyPreferences[c,s] represents the preference the student s
   has for company c:
17 array[Companies, Students] of int: CompanyPreferences;
18 % Disappointment[c] is the dissapointment cost for company c:
19 array[Companies] of int: Disappointment;
20 % Lower[c] is the minimum number of students company c can accept
   to not charge disappointment cost:
21 array[Companies] of int: Lower;
22 % Upper[c] is the maximum number of students company c can accept:
23 array[Companies] of int: Upper;
24 % dummy represents a dummy company:
25 int: dummy = nCompanies+1;
26
27 % VisitedCompanies[s,p] is the company p to be visited by student s.
28 % p represents one of three interviews assigned to the student
   based on their preferences:
29 array[Students, Preferences] of var CompaniesAndDummy:
   VisitedCompanies;
30 % Symmetry Breaking Constraint, where all assigned preferences p in
   VisitedCompanies[s,p] for each student s are ordered increasingly:
31 constraint symmetry_breaking_constraint(forall(s in
   Students)(increasing([VisitedCompanies[s,p]|p in Preferences])));
32 % 1-way channeling - InterestedStudents[s] is mutually redundant
   with CompanyPreferences. InterestedStudents[s] represents the sum
   of the amount of preferences given by a student s, where each
   preference is less than or equal to 4:
33 array[Students] of 0..nCompanies: InterestedStudents = [sum([1|c in
   Companies where CompanyPreferences[c,s] <= 3])| s in Students];
34 % 1-way channeling - TotalCompanyOccurances[c] is mutually
   redundant with VisitedCompanies. TotalCompanyOccurances[c]
   represents the sum of all assigned interviews for that company c:
35 array[Companies] of var 0..max(Upper): TotalCompanyOccurances =
   global_cardinality(VisitedCompanies,Companies);
36 %array[Companies] of var int: TotalCompanyOccurances =
   [count([VisitedCompanies[s,p]|s in Students, p in Preferences],
   c)|c in Companies];
37 % No single student can have an interview at any company more than
   once, multiple dummy companies are allowed:
38 constraint forall(s in
   Students)(all_different_except([VisitedCompanies[s,i]|i in
   Preferences], {dummy}));
39 % A company c cannot be assigned more interviews than their
   predefined upper limit - Upper[c]:
40 constraint forall(c in Companies)(TotalCompanyOccurances[c] <=
```

```
    Upper[c]) :: bounds_propagation;
41  % A company c cannot be assigned less interviews than their
    predefined lower limit - Lower[c]. If they are assigned less
    interviews than Lower[c], the total occurances of interviews must
    be 0:
42  constraint forall(c in Companies)(if TotalCompanyOccurances[c] <
    Lower[c] then TotalCompanyOccurances[c] = 0 else
    TotalCompanyOccurances[c] >= Lower[c] endif);
43  % For all students that have less than 3 prefferences, where each
    preference is greater than or equal to 4, then there must be
    3-(the sum of preferences less than 4) dummy companies represented
    in the preferences p for that student s in VisitedCompanies[s,p]:
44  constraint forall(s in Students where InterestedStudents[s] <
    3)(count([VisitedCompanies[s,p]|p in Preferences], dummy) = (3 -
    InterestedStudents[s]));
45  % For all students that have more than 3 prefferences, where each
    preference is less than 4, then that student will not have any
    occurances of dummy company represented in the preferences p for
    that student s in VisitedCompanies[s,p]:
46  constraint forall(s in Students where InterestedStudents[s] >=
    3)(count([VisitedCompanies[s,p]|p in Preferences], dummy) = 0);
47  % 1-way channeling - MaxPreferences[s] is mutually redundant with
    CompanyPreferences. MaxPreferences[s] represents the sum of the 3
    best prefrences given by the student s overall, for all companies:
48  array[Students] of 3..15: MaxPreferences =
    [sum([(sort([CompanyPreferences[c,s]| c in Companies]))[i]| i in
    1..3])|s in Students];
49  % 1-way channeling - TotalPreferences[s] is mutually redundant to
    CompanyPreferences. TotalPreferences[s] represents the sum of the
    prefrences p for student s from the 3 assigned interviews in
    VisitedCompanies[s,p]:
50  array[Students] of var 0..15: TotalPreferences =
    [sum([CompanyPreferences[VisitedCompanies[s,p],s]|p in Preferences
    where VisitedCompanies[s,p] < dummy])|s in Students];
51  % 1-way channeling - totalRegret is mutually redundant to
    TotalPreferences and MaxPreferences.totalRegret is represented as
    the sum of all differences between assigned preferences and the 3
    best given preferences for all students:
52  var 0..((15-3)*nStudents): totalRegret = sum([TotalPreferences[s] -
    MaxPreferences[s]|s in Students]) :: value_propagation;
53  % 1-way channeling - dissapointmentCost is mutually redundant to
    Dissapointment and TotalCompanyOccurances. dissapointmentCost is
    the sum of all costs the host must incur if a company is not to
    have any interviews during the conference:
54  var 0..(max(Disappointment) * nCompanies): dissapointmentCost =
    sum([Disappointment[c]| c in Companies where
    TotalCompanyOccurances[c] = 0]) :: value_propagation;
55  % 1-way channeling - prefSum is mutually redundant to
    TotalPreferences. prefSum is the sum of all assigned interview
```

```
  prefereces given to the students:
var int: prefSum = sum(TotalPreferences);

% Solve to minimize dissapointmentCost, totalRegret and prefSum:
solve
% The int search annotation is used to tell the solver to select
  variables with the largest domain and to assign the variables
  their smallest domain value:
:: int_search(VisitedCompanies, largest, indomain_min)
% Restart the search after 1500 nodes:
:: restart_constant(1500)
% The probability of VisitedCompanies being fixed to the previous
  solution is 78:
:: relax_and_reconstruct(array1d(VisitedCompanies), 78)
minimize dissapointmentCost + totalRegret + prefSum;
```

Constraints Enforced by the Choice of Decision Variables. No decision variables enforce or apply constraints automatically in the model. All constraints have been modeled explicitly.

Redundant Decision Variables and Channelling Constraints. Most, if not all, decision variables in the model, except `VisitedCompanies`, are mutually redundant to `VisitedCompanies`. All decision variables are decided or constructed based on the values contained in the `VisitedCompanies` array.

`TotalCompanyOccurances` is a variable array that contains the occurrences for each of the companies in the `VisitedCompanies` array. It helps with readability and simpler constraint definitions. However, since it is derived from `VisitedCompanies` using 1-way channeling, it makes `TotalCompanyOccurances` mutually redundant to `VisitedCompanies`.

`TotalPreferences` is an array that represents the sum of the preferences of the companies to which they were assigned in the `VisitedCompanies` array. Thus, making it mutually redundant with the `VisitedCompanies` array using 1-way channeling.

`totalRegret` is a variable that represents the sum of all regret from all students. Regret is calculated using `TotalPreferences` and `MaxPreferences`, making `totalRegret` mutually redundant to those variables.

`dissapointmentCost` is a variable that represents the sum of all disappointment costs given by the companies if they are not to have any interviews during the conference. Calculating `dissapointmentCost` requires the use of the parameter `Dissapointment` and the variable `TotalCompanyOccurances`, making `dissapointmentCost` mutually redundant to those variables.

`prefSum` is the sum of all `TotalPreferences`, which makes it mutually redundant to `TotalPreferences`.

There are also some redundant non-decision variables present in the model. Namely `InterestedStudents` and `MaxPreferences`.

`InterestedStudents` represents the sum of the number of preferences given by a student. It is populated using the parameter array `CompanyPreferences`, which makes it mutually redundant to it.

The same thing goes for `MaxPreferences` which represents the sum of the top 3 best preferences for each student. The sum is calculated also using the parameter array `CompanyPreferences`, making `MaxPreferences` mutually redundant to it.

Implied Constraints. No implied constraints were used in the model.

Symmetry-Breaking Constraints. The variable array `VisitedCompanies` has vertical reflection and variable symmetries. These symmetries are broken with the use of sorting. Since each column in this array represents students, we sort the rows which represent the assigned companies for the students. Experiments on our local machines show that the symmetry-breaking constraint speeds up the model noticeably. Using a 10-second timeout using `Gecode` on a laptop class processor (Apple Silicon 10 core M1 pro with 16GB ram), an objective value of `406` is observed without the use of the symmetry breaking constraint. With the symmetry breaking constraint, the objective value has reduced down to `395` within the same time frame.

We have taken advantage of the special predicate `symmetry_breaking_constraint` as recommended.

The symmetry-breaking constraint is shown below:

```
31 constraint symmetry_breaking_constraint(forall(s in
     Students)(increasing([VisitedCompanies[s,p]|p in Preferences])));
```

Efficiency. One of the checklist items from the last slides of topic 2, beware of where Θ having variables was violated in multiple places in the model. The reason for violating the advice was to avoid constraining variables of no interest. It may have been possible to create additional variables and thus avoid some of the where conditions, but the space complexity would undoubtedly increase. The checklist item, beware of if Θ with Θ having variables, was violated when ensuring the total company interviews stays within bounds. It would be possible to avoid the if-statement and create two constraints, but we concluded through experiments that it would increase the solving time.

The advice about merging constraints over shared variables from the checklist in the last slides of topic 3 was violated for constraints over the `VisitedCompanies` array simply because the constraints are incompatible. The constraints over `InterestedStudents` are not merged, as it would decrease readability without any noticeable performance improvements. The same goes for the `TotalCompanyOccurances` array.

Correctness. The correctness of the solution cannot be argued at the time of writing this initial report. Issues with outputting the correct variables on the thinlinc machines allowed only for the objective value to be printed. The objective value is an addition of the 3 main variables that we want to minimize: `dissapointmentCost + totalRegret + prefSum`. The problem statement in CSPLib is divided into 4 days, the model written for this report is for day 2. Day 3 hints at an optimal solution for day 2, which is the following:

> "As the best solution yesterday allowed for a maximum of one regret only, this is now imposed as a hard constraint, so the maximal preference regret for each student is one. As all companies could be satisfied in yesterday's optimal solution, we now have to plan for all companies getting between their lower and their upper bound of interviews, we can no longer disappoint them."

Thus the correctness of the approach can be verified by this quote provided in the problem statement.

A further investigation into correctness will be provided once we figure out the situation with the output variables in the thinlinc machines.

Inference Annotations. We have experimented using inference annotations in the model. Using LNS as a search strategy dramatically improved the performance of `Gecode`. Initially, we experimented using a machine with a laptop-class processor (Apple Silicon 10 core M1 pro with 16GB ram). With a 10-second timeout and allowing only a single thread for use by the solver, a score of `395` was achieved using systematic search. When using inference annotations to trigger local search with `Gecode`, the final objective value upon timeout had reached `148`, an objective score `247` less than systematic search, an impressive improvement. On the thinlinc server provided by the university, local search on `Gecode` did not manage to reach an optimal value, even after a 5-minute set timeout (300000 ms). Even though LNS offered a noticeable performance improvement over systematic search on the thinlinc machine, it did not manage to beat the result provided by our local machine.

Backend	Gecode using LNS		Gecode using SS	
instance	`obj`	time	`obj`	time
test2	**158**	t/o	379	t/o

The values of ρ and σ were:

$$\begin{aligned} \rho &= 1500 \\ \sigma &= 78 \end{aligned}$$

Thus we conclude that using LNS for lower-powered machines is beneficial over systematic search. However, this statement only applies to the Interview Assignment Problem and the data set provided in `test2.csv`. We believe it is a reasonable trade-off, as real-world scenarios usually do not consist of small instances of a problem. Achieving a feasible solution fast that is close to optimal outweighs the incrementally small benefit of having an optimal solution, as an optimal solution can take a lot longer to find than a solution that is feasible and close to optimal.

Search Annotation. The Interview Assignment Problem resolves around creating a permutation of a 2-dimensional array, that contains assigned interviews for students. Thus all of the problems with other variables depend on this array, which is why we deemed it most suitable for use in the search annotation. The search annotation `int_search` was used since the `VisitedCompanies` array consists of integers. The search annotation is presented below:

```
59 solve
```

The variable selection strategy used was `largest`. Through experimentation using `smallest` and `first_fail` we found that using `largest` provided better results faster. The `largest` selection strategy allows for a depth-first approach to the search using the largest domain available, which for this instance speeds up the search noticeably.

The domain partitioning strategy `indomain_min` helps focus the search on the smallest available value in the current domain. Using `indomain_min` helps with ignoring symmetric occurrences of a solution, as we choose to go with a value and stick to it rather than testing any of the values in the current domain, which leads to a depth-first approach to the search, which for this problem leads to dramatically faster solving times. The objective value, which we solve for, is also a value that needs to be minimized, which might contribute to why `indomain_min` is faster than any other partitioning strategy.

We thought the same argument could be applied to the other partitioning strategy `indomain_max`, which chooses the largest value in the current domain. However, we observed a significant negative performance impact through local testing, which led us to go with the strategy we chose first, `indomain_min`.

B Evaluation of `day2.mzn`

Backend	CP-SAT		Gecode		Gurobi		PicatSAT		Yuck	
instance	obj	time	obj	time	obj	time	obj	time	obj	time
test2	370	t/o	379	t/o	**142**	**6460**	**142**	91298	242	t/o

Table 1: Results for our model of the second day of the `Interview Assignment` problem, which is a `minimisation` problem. In the 'time' column, if the reported time is less than the time-out (300,000 milliseconds here), then the reported objective value in the '`obj`' column was *proven* optimal; else the time-out is indicated by 't/o' and the reported objective value 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.

It is clear that the best overall performing back-end is `Gurobi`. `Gurobi` is one of the two back-ends that managed to find an optimal, however, it was also the quickest to do so, beating the `PicalSAT` back-end by almost 1.5 minutes.

Conclusions regarding scaling and difficulty cannot be drawn due to there only being one problem instance.

When comparing systematic search and local search results in `Yuck`, we observe worse performance overall, as the objective value did not manage to reach optima before timing out. Yuck always times out, however it may provide better results for much larger instances, but for this instance, it does not manage to. When comparing local search in `Yuck` versus local search in `Gurobi` we observe a further worsening performance in the `Gurobi` back-end, where `Yuck` manages to reach an `obj` value of `242` and `Gurobi` a value of `379`. Thus we conclude that local search for this problem and this instance is not suitable. However, comparing the results we managed to achieve when experimenting using our local machine, seen under `Inference Annotations.`, we can conclude that local search is much more suitable for low-powered ARM-based architectures, as a huge increase in performance was noted.

The results did not contain any contradictory or 'ERR' values.

`A paragraph regarding data pre-processing and conversion will be added in the final report`