Catholic University of Louvain

Project 4 : Modeling

LINGI2365 - Constraint Programming

Auteurs :
Vanwelde Romain (3143-10-00)
Crochelet Martin (2236-10-00)

Groupe 7

Superviseurs :
Pr. Yves Deville
François Aubry

6 avril 2014

Table des matières

1 Louvain-La-Neuve Golfer Problem

1.1 Explain which symmetries can arise for this problem.

This problem is highly symmetrical : indeed we can easily identify four different type of symmetry :

- Two different teams can be switched for the same week
- For two players, we can switch their planning entirely
- Inside a group, the players can be switched anyway we want
- Two different weeks can be switched entirely

Moreover, from the point of view of the constraints, stating that player A can't play twice with player B is strictly equivalent to saying that player B can't play twice with player A. This has a huge impact on the performances of the models.

1.2 Describe two possible models for this problem.

1.2.1 explain your models in detail

- The first model we consider is actually the most powerful of them both : it uses a variable that stores for each player and each week, the group in which the golfer plays. This formulation benefits from various advantages, simplicity being one of the most valuable. Indeed, we can summarize the constraints applied to the CSP into two types :
  - the first type is simply the necessity that all groups are full. This is important because it is not naturally forced by our choice of variable : indeed a simple (despite false) solution to the problem is that no golfer at all plays during the weeks.
  - the second type is a constraint that derives from the definition of the problem : no golfer can play with another more than once. We simply verify that there is not two weeks where two players belong to the same group. The finesse here is that we only set half the constraints : indeed, as explained hereabove, the other half is just redundant and would consume CPU time to be verified.
- the second model we considered is using a 3D matrix in order to keep track of the position of the golfers. We use a variable that stores, for each week, group, slot, the golfer that takes it. This choice has proven more difficult to use to express our constraints since we add a new dimension, the slot that complexifies the search. The main idea behind this choice was that we could control the symmetry generated by the slots but this has proven itself quite difficult. Moreover, the memory space taken by this model is smaller (for a number of weeks smaller than 8) the one taken by the previous model. The constraints we use here are :
  - the fact that a player can only appear once per week in the planning (that is not enforces by our choice of variable). This means that all assignments of our variable for a same week must be different.
  - The fact that a player can play with another at most once (again, on half of the domain).

1.2.2 explain how you can modify your models to take symmetries into account

We use the symmetries of the problem by breaking them and by doing that pruning the search space from the very beginning. (SBSA) Indeed, in both case, we arbitrarily choose the assignment for the first week and the assignment of the first golfer for the whole planning. By doing that, we remove already a great number of possible states : we remove the symmetrical states of the first week as well as the ones for the position of the first player. Moreover, this selective assignment does not remove any solution thanks to the symmetry of the problem.

1.3 Implement both models (considering symmetries) in Comet.

cf. code.

1.4 Explain which variable / value ordering heuristics you use with each model.

In both model, we use the simple heuristic of the least remaining values. However, the order in which we consider the dimensions is quite important :

— for the first model, we consider first the golfers that have most of their planning complete and then the weeks that are almost already planned. By doing so it that order, we rapidly fail to assign the golfers to a group and can prune the search tree more extensively.

— for the second model, we have not determined any particular order since any order we tried decreased the performances of a non negligible factor.

1.5 What is the theoretical maximum number of week in a schedule ?

The simplest way to think about that is to think by absurd : Imagine that we could plan the golfers for 11 weeks, that would mean that one player "p" would need to play with $3 \cdot 11$ different players. This is impossible because we have only 31 different players at our disposal. Leading the same reflexion for 10 weeks easily show that it is feasible.

1.6 Indicate the maximum number of weeks you could identify in a reasonable time limit using both models.

With the first algorithm (as stated hereafter) we easily find the solution for 9 weeks after 372 ms and have to kill the program after 5 min of research for the 10 weeks planning.

1.7 Indicate for each model, for each number of weeks the time needed to find a solution, the number of failures and the number of choices. Explain the results, do they correspond to what you would have expected ?

Algorithm	Week	Time taken (ms)	Choices	Failures
1	3	23	62	224
1	4	46	93	336
1	5	74	124	448
1	6	135	155	560
1	7	177	186	672
1	8	260	228	872
1	9	372	259	984
1	10	killed after 5 min	/	/
2	3	1903	96	1488
2	4	2885	128	1984
2	5	4086	160	2480
2	6	killed after 5 min	/	/
2	7	/	/	/
2	8	/	/	/
2	9	/	/	/
2	10	/	/	/

2 The Time Tabling Problem

2.1 Explain which symmetries arise in this problem.

The first symmetry we find is in relation with timeslots. Indeed, we can easily invert all the course given at specific timeslot with all the courses of another timeslot.

A second symmetry could be the attribution of rooms that are exactly the same (same features, and same capacity). Since we assume that this kind of symmetry doesn't happen often, and that it's something quite tricky to express in our model, we didn't implement it in our model.

2.2 Design an efficient model for this problem.

```
...

// create data variables
int roomsize[rroom];                        // roomsize
int atends[rstudent,revent];                // attends
int roomequipment[rroom,rfeature];          // room equipment
int eventrequirement[revent,rfeature];      // course needed equipment

int nstudentattends[revent] = 0;            // student # enroled in the course
```

```
10
11
12 ...
13
14
15 // model variables
16 tuple triple {int a1;int a2;int a3;}
17
18 set{triple} Triples();
19 forall (i in 1..nroom, j in timeslot)
20    Triples.insert(triple(i,j,(i-1)*ntime + j));
21
22 Table<CP>roomslot(all(e in Triples) e.a1, all(e in Triples) e.a2, all(e in
       Triples) e.a3);
23
24
25 var<CP>{int} lectureslot[revent](cp, timeslot);
26 var<CP>{int} lectureroom[revent](cp, rroom);
27 var<CP>{int} roomslotid[revent](cp, 1..nroom*ntime);
```

time_tabling_model.co

2.3 Explain your model for this problem and explain how it handles symmetries.

In the model, the data variables contains information extract from the inputfile. We added one data variable (`nstudentattends`) which will store the number of students which assists to different courses.

The model is composed of 3 model variables, which are `lectureslot`, `lectureroom` and `roomslotid`. `lectureslot` gives, for each event, all the possible timeslots for this event. `lectureroom` gives, for each event, all the possible rooms for this event. Finally, `roomslotid` gives an ID for each possible combination of room/timeslot.

```
1  solve<cp> {
2     ...
3  } using {
4     label(lectureslot[1],1);
5
6     forall(e in revent) by (lectureroom[e].getSize())
7        tryall<cp>(r in rroom : lectureroom[e].memberOf(r)) by (r)
8           label(lectureroom[e],r);
9
10    forall(e in revent) by (lectureslot[e].getSize())
11       tryall<cp>(t in timeslot : lectureslot[e].memberOf(t)) by (t)
12          label(lectureslot[e],t);
13 }
```

time_tabling_symmetry.co

The symmetry concerning the time is handeled in the solver, we can arbitrary fix the first timeslot to any course. Then, we continue to handle it by trying to bound `lectureslot` by growing slot ID. An event will then be assign to a slot which already

contains some events, or the first which has no events yet, but it will never visit events with higher ID (if the problem is solvable).

2.4 Implement your model in Comet.

Here are the constraints implemented in Comet. All others important part are already in the previous sections.

```
solve<cp> {
   // A student can attend only one lecture at any time slot
   forall (s in rstudent){
      cp.post(alldifferent(all (e in revent : atends[s,e] == 1) lectureslot
          [e]),onBounds);
   }

   // compute roomslot's id
   forall (e in revent){
      cp.post((lectureroom[e]-1) * ntime + lectureslot[e] == roomslotid[e])
          ;
   }

   // A room can hold only one lecture at any time slot
   cp.post(alldifferent(all (e in revent) roomslotid[e]),onDomains);


   // Table constraint
   forall (e in revent){
      cp.post(table(lectureroom[e],lectureslot[e],roomslotid[e],roomslot));
   }

   // A lecture can take place only in a room having the required features
   forall(e in revent) {
      forall (f in rfeature) {
         cp.post(eventrequirement[e,f] <= roomequipment[lectureroom[e],f]);
      }
   }

   // A lecture can take place only in a room big enough to hold all the
       students that need to attend
   forall (e in revent){
      cp.post(nstudentattends[e]<=roomsize[lectureroom[e]]);
   }

} using {
   ...
}
```

time_tabling_constraint.co

2.5 To solve this problem you will need a search procedure that is more efficient than a simple label.

We first implemented our model by binding variables which have few rooms left, then by binding variables which have few timeslots left. We decided to do it in this way

(and not first the timeslots then the rooms) after some tests which were showing that the second solution were far away lower than the first one. This is probably because finding an appropriate room is more difficult in this problem than finding a good timeslot. The code is already given in previous section, and the results of the tests are in the next one.

2.6 Test your model on each of the instances provided on the iCampus site. Indicate for each instance the time needed to solve it, the number of failures and the number of choices.

File ID	Time	#Failures	#Choices
#1	3685	1	800
#2	3535	0	800
#3	2671	368	1041
#4	307397	259291	140520
#5	Out Of Time	Out Of Time	Out Of Time
#6	4044	3	700
#7	3934	0	700
#8	3692	0	800
#9	5222	0	880
#10	3548	0	800
#11	4375	0	800
#12	3671	0	800
#13	4017	0	800
#14	4530	0	700
#15	4085	0	700
#16	Out Of Time	Out Of Time	Out Of Time
#17	3718	0	700
#18	3126	0	800
#19	4186	0	800
#20	4000	1	700