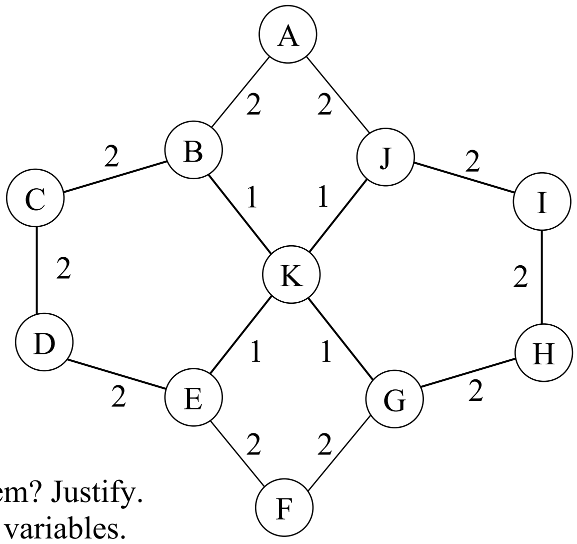
Constraint Programming

2016/2017– Mini-Test #1

Friday, 3 November, 18:00 h in Room 111-VII

Duration: 1.5 h (open book)

1. Finite domain Constraints - Propagation (6 pts)



Consider the constraint network on the right, where nodes represent variables, all with domain {**1,2,3**}. Arcs labelled k, mean that the value of the connected variables must differ by at least k (i.e. arc with label 1 means ≠).

a) **(2 pt)** What pruning is achieved if node-consistency is maintained? And arc-consistency?

b) **(2 pt)** What pruning is achieved if path-consistency is maintained?

c) **(2 pt)** How many distinct solutions exist for the problem? Justify.
Hint: Analyse the induced constraints on the problem variables.

Proposed Solution

a) Since there are no unary constraints, node-consistency does no pruning. As to arc-consistency, all constraints of type “2” impose that value 2 is pruned from the domain of the variables. So all variables but variable K have their domain pruned to {1,3}. For variable K that is only involved in ≠ constraints with variables B, E, G and K, since all of them have 2 values in their domain, one of them will always support each of the values in the domain of K, which keeps its initial domain {1,2,3}.

b) Path-consistency enforces K to take value 2. In fact, since B and D are both connected to C the only way in which their values is extended to C is to guarantee that they B and D are equal (i.e. either 1 or 3, so that C can take the other value. Likewise C and E must be equal. Then extending label {B,C} to E (equal to C) imposes a constraint of type “2” between E and B, so one takes value 1 and the other value 3. Then the only possible extension of label {E,B} will impose K = 2. This is compatible with the domains of both G and J.

c) The above reasoning can be extended to show that

```
K = 2
A = C = E = G = I in {1,3}
B = D = F = H = J in {1,3}
A ≠ B
```

Hence the only two solutions are:

```
K = 2
A = C = E = G = I = 1
B = D = F = H = J = 3
```

or

```
K = 2
A = C = E = G = I = 3
B = D = F = H = J = 1
```

2. Modelling with Finite Domain Constraints (8 pts)

A group of **nf** friends is organizing a number of **nw** weekly lunches on a restaurant, which has a round table that fit all of them (i.e. it seats **nf** people) . Because the goal is to socialise, the organiser is trying to find an allocation of seats for the participants, so that each of them has different neighbours every week.

a) **(4 pt)** Specify a model for this problem in Comet, namely declaring the decision variables you use in the model (with their domains), as well as the constraints that should be posted to impose the restrictions of the problem. Assume that the friends are identified by the integers **1..nf**.

b) **(2 pt)** Adapt the above problem, for the case where there are some "friends" that do not like each other so they should never be placed in adjacent seats. Assume that these incompatibilities are given by a **k × 2** integer array **nn**, where each row denotes the pair of ids of **k** incompatible friends (e.g. nn = [[1,4],[3,5], [3,7]] indicates that the group members with ids 1 and 4 should never be neighbours in the table, and the member with id 3 is incompatible with those with ids 5 and 7).

c) **(2 pt)** Given the models you adopted, are there symmetric solutions, i.e. solutions that can be obtained from others by a simple mapping? If so can you add extra constraints that prevent searching for these symmetric solutions.

Proposed Solution:

a) We may use a nw × nf matrix of decision variables where each row represents the seats taken by the different members of the group. The domain of each variable is the set of available seats, i.e. the range 1..nf.
Two friends are neighbours in some week if their positions differ either by 1 or by nf-1 (the table is round, so seat **1** is next to seat **np**). So for the whole number of weeks, the number of times their positions are next to each other should not exceed **1**.
Moreover the positions of the friends in every week must must be different.
Hence the following Comet specification models the problem:

```
int nw = …;
int nf = …;

Solver<CP> cp();
  var<CP>{int} pos[1..nw, 1..nf](cp,1..nf);
solve<cp> {
   forall(i in 1..nw)
      cp.post(alldifferent(all(j in 1..nf) pos[i, j]));
   forall(i in 1..nf, j in 1..nf: j > i)
      cp.post(1 >= sum(w in 1..nw)
          (abs(pos[w,i]-pos[w,j])== 1 ||
           abs(pos[w,i]-pos[w,j])== np-1));
} using {
...;
}
```

b) Given the matriz nn of incompatible neighbours, all that is needed is to add the corresponding incompatible constraints for every week, i.e. the positions of the elements referred should not differ by 1 nor by nf-1.

```
...
int nc = 3;
int nn[1..nc,1..2] = ...; // e.g. nn = [[1,4],[3,5], [3,7]];
...
solve<cp> {
...
   forall(w in 1..nw, i in 1..nc)
      cp.post( abs(pos[w,nn[i,1]]-pos[w,nn[i,2]]) != 1 &&
               abs(pos[w,nn[i,1]]-pos[w,nn[i,2]]) != np-1);
...
```

c) The model above allows several symmetries, namely

1. The weeks can be interchanged in the model. This can be prevented by imposing an increasing order of the position of friend with id 2.

   ```
   forall(w in 2..nw) cp.post(pos[w-1,2] < pos[w,2]);
   ```

2. The positions of all the friends may be shifted by k positions in each of the weeks. This can be prevented by imposing that the friend with id=1 always occupies seat 1.

   ```
   forall(w in 1..nw) cp.post(pos[w,1] == 1;
   ```

3. The positions of the friends can be “reversed”, i.e. changing the left and right neighbour. To impose that only one “direction” is possible, and since friend with id 1 takes seat 1, we may impose that the neighbour in one of his sides has a greater id than the other neighbour (in all the weeks), as follows by imposing that the friend with id=1 always occupies seat 1.

   ```
   forall(w in 1..nw) cp.post(pos[w,2]> pos[w,nf] );
   ```

3. Global Constraints (6 pts)

Consider a global constraint `max_copies(x, k)` that given integer `k` and an array `x` of decision variables defined on a solver `cp`, with range `1..n`, constrains the number of variables `x` that have some value `v` to be at most `k`. For example, for `k = 1` this constraint corresponds to the alldifferent constraint. For `k = 2`, no more than 2 variables may have the same value (i.e. it cannot be `x[1] = x[2] = x[3]`). Consider further that the implementation of the global constraint achieves domain consistency (or generalized arc-consistency).

Consider now that in some application, not only should the above constraint be imposed on some array x, but also the array must be sorted in increasing order, and this is imposed by the standard set of constraints

```
forall(i in 2..n) cp.post(x[i-1] <= x[i]).
```

a) **(2 pt)** Justify that if the array x has size 6 and its variables have domain `1..4` (i.e. they are specified as `var<CP>int x[1..6](cp,1..4)`), the separate specification of the `max_copies(x,2)` and the less or equal constraints (<=) do not lead to any pruning of the domain of the variables.

b) **(2 pt)** Show that such pruning would be possible if a global constraint `sorted_max_copies(x, k)` were implemented so as to achieve domain consistency (or generalised arc-consistency). In particular what pruning would be achieved in the situation of item a).

c) **(2 pt)** Assuming that global constraint `max_copies` is available, but that `sorted_max_copies` is not, would you suggest any redundant constraints that could help pruning the search space, namely when all of the variables of array `x` have a range `1..n` as their initial domain? What pruning would be achieved in the situation of item a)?

Proposed Solution:

a) In an array with 6 variables with 4 distinct possible values any variable can take any of the 4 values, since a solution in which value 1 is assigned to 2 variables, value 2 to other 2 variables and values 3 and 4 to the remaining variables satisfies the `max_copies(x, 2)` constraint. Since any permutation of the variables is also a solution, each variable in the array may take any value in its domain `1..4`, and the global constraint `max_copies(x, 2)` could not impose any pruning.

   Moreover, the sorting (`<=`) constraints also do not impose any pruning, since an assignment of any of the values in the domain `1..4` to all the variables satisfies the constraints, so no value is pruned from the domain of the 6 variables.

b) Since at most 2 variables may take the same value, value 3 cannot be taken by variable `x[2]`, since, together with the sorting constraints, this would impose that the 5 variables `x[2]` to `x[6]` would take values 3 or 4, and only 4 of them could take these 2 values. Hence variable `x[2]` would have its domain pruned to `1..2`. By a similar reasoning, the same would happen to variable `x[1]`.

   On the other hand, variable `x[4]` cannot not take value 4, otherwise, given the sorting constraints, the 3 variables `x[4]` to `x[6]` should all be assigned value 4, exceeding the number of copies. Neither can variable `x[3]` have value 1, otherwise the 3 variables `x[1]` to `x[3]` should all be assigned value 1, exceeding the number of copies. Generalising this reasoning, the domain of the different elements of array `x` should be pruned to

   `x[1]` and `x[2]`: `1..2;`
   `x[3]` and `x[4]`: `2..3;`
   `x[5]` and `x[6]`: `3..4;`

c) Following the reasoning of the previous item, one may impose that:
   a. Value 1 can be removed from the domain of variables `x[k+1] .. x[n]`.
   b. Value 2 can be removed from the domain of variables `x[2k+1].. x[n]`
   c. Value 3 can be removed from the domain of variables `x[3k+1].. x[n]`
   d. …

   Such pruning would be achieved by posting constraints

```
            forall (i in 1..k, j in 1..k: 2*i+k <= n)
                  cp.post(x[2*i]+j >= i;
```

   Similarly, the higher values of the initial variables could be removed by constraints to impose that
   a. Value `k` can be removed from the domain of variables `x[n- k+1] .. x[ n ]`.
   b. Value `k-1` can be removed from the domain of variables `x[n-2k+1] .. x[n- k]`
   c. Value `k-2` can be removed from the domain of variables `x[n-3k+1] .. x[n-2k]`…
   d. …

   Such pruning would be achieved by posting constraints

```
            forall (i in 1..k, j in 1..k: n-i*+j+1 >= 1)
                  cp.post(x[n-(i*k)+j] <= n-i*k;
```