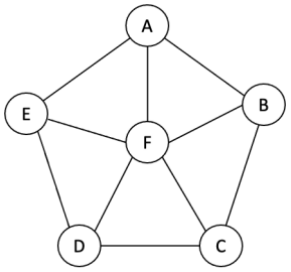
Constraint Programming

2021/2022– Mini-Test #1
Monday, 15 November, 9:00 h, Room 127-II
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}**, and the arcs constraints of difference (**"="**).

a) **(1 pt.)** Show that the problem is not satisfiable.

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

c) **(2 pt.)** Show that there are implicit constraints of equality (e.g., A = C) induced by the original constraints. Would maintaining path-consistency detect such constraints? And maintaining 4-consistency?

d) **(1 pt.)** What implicit equality constraints would remain if variable A would have domain {0,1,2,3}. Would these equalities impose the grounding of any variable (i.e. assigning it a fixed value).

Proposed Solution

a) The problem is not satisfiable. Variable F should take one of the values 1, 2 or 3, and hence ther are only two values left for the remaining variables. For example, for F = 3, the remaining variables must have values 1 or 2. But it is easy to see that if A takes one of the values, say A = 1, then B must be different from A (say B=2), C must be different from B, and hence C = A, D must be different from C , i.e. D =B ≠ A, and E must be different from D, i.e. equal to C and A. But this is not possible, since it should be A ≠ E.

b) There are no unary constraints, so node-consistency does not prune any values from the variables' domains. All variables have at least 2 values in their domains, so maintaining arc-consistency on ≠ constraints, say X ≠ Y, does not prune any values from the variables, since any value from X has at least one support (a different value) in Y.

c) takes the F tasatisfiable, and has two solutions:

- The **>4** constraints prune the domains of variables **a, b**, **d** and **e** to the set **{1,6}**, and **a ≠b**;
- The **=2** constrain variables **a** and **e** to have the same parity (in fact a difference of 0 or 4).
- Hence, **a** must be equal to **e** and, likely, **b** must be equal to **d**.
- Hence the solutions are
  - S1: **a = e = 1, f = 3, b = d = 6, c = 4**; and
  - S2: **a = e = 6, f = 4, b = d = 1, c = 3**.

d) Node-consistency does not prune any domain since there are no unary constraints.

Bounds consistency does not prune any of the domains since the bounds of the domain of all variables have support on the other connected variables. For example, **a = 1** has support on **b = 6** and **c = 3**; **a = 6** has support on **b = 1** and **c = 4**. Hence the domain of **a** is not pruned. And the same applies to variables **b**, **d** and **e**. Regarding variable **c**, its upper bound **6** has support on **b = 4** and **d = 4**, and its lower bound **1** has support in **b = 3** and **d = 3**, and the same applies to **f**.

Arc-consistency removes values **2,3,4** and **5** from the domain of **a**, as they have no support on **b**, and the same applies to **b**. Hence variables **a, b, d** and **e** have their domain reduced to **{1,6}**. And these pruned domains imply, in turn, that variables **c** and f have their domains pruned to **{3,4}**.

e) Path consistency imposes that variables **a** and **e** have a difference of either 0 or 4. With their domains pruned to **{1,6}**, this means that arc-consistency would infer that **a = e** (and **b = d**).

f) Given the vertical symmetry of the problem, a constraint **a ≥ b** should eliminate solution **S1**, that can be inferred from **S2** changing a ↔ b, d ↔ e, and c ↔ f.

2. Global Constraints (5 pts)

Consider a global constraint that given an array of decision variables imposes that they are *shuffled* into another array according to a given index mapping. For example, given a mapping `M`, the decision variables of vector B should be obtained by mapping the variables from vector A according to M, i.e. `B[i] = A[M[i]]`). For example, for `M = {1,2,0}` we should have

`B[0] = A[1], B[1] = A[2] and B[2] = A[0].`

a) **(2pt.)** Implement in Choco this constraint in a function **`shuffle`** with signature

```
public static void shuffle (Model md, IntVar [] S, IntVar [] M, IntVar [] T)
```

Suggestion: Consider the global constraint **element** available in Choco.

b) **(3 pt.)** Assuming you are given two arrays with **n** decision variables representing the starting times (**S**) and durations (**D**) of **n** tasks, use the **`shuffle`** constraint to complete the code below so as to constrain the tasks to be executed with no overlaps nor gaps between them.

```
n = ...;
IntVar [] S = md.intVarArray(n, sLo, sUp); // the starting times
IntVar [] D = md.intVarArray(n, dLo, dUp); // the duration
...
```

Proposed Solution

a) To implement this constraint, we have to consider that the mapping array contains the indices of the tasks so as to have them sorted, i.e. for all i's such `B[i] = A[M[i]]`. Since these constraints require to use decision variables as indices of array A, the element constraint can be used as:

```
function void shuffle (Model md, IntVar [] A, IntVar [] M, int [] B){
     for (int i = 0; i < S.length; i++){
          md.element(B[i], A, M[i], 0).post();
     }
```

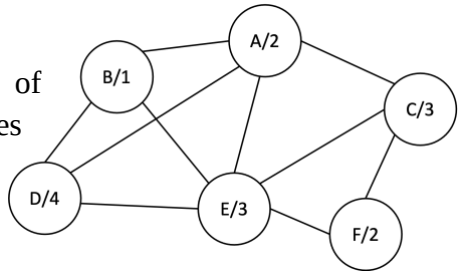
b) Given the array of decision variables representing the tasks, all that is needed is to sort the tasks such that the ending of the mapped tasks is the same as the starting times of the sorted tasks. The mapping should guarantee that all indices are used. Hence,

```
n = ...;
IntVar [] S  = md.intVarArray(n, sLo, sUp); // the starting times
IntVar [] D  = md.intVarArray(n, dLo, dUp); // the durations

IntVar [] Sm = md.intVarArray(n, sLo, sUp); // mapped starts
IntVar [] Dm = md.intVarArray(n, dLo, dUp); // mapped durations
IntVar [] M  = md.intVarArray(n, 0, n-1);   // a mapping
```

```
md.allDifferent(M).post();     // constrain the mapping
shuffle(md, S, M, Sm);         // obtain the mapped starts
shuffle(md, D, M, Dm);         // obtain the mapped durations

for (int i = 0; i < n-1; i++) {
      md.arithm(Sm[i], "+", Dm[i], "=", Sm[i+1]). post();
}
```

3. Modelling with Finite Domain Constraints (9 pts)

Graph Colouring Variant



Given a graph, the standard Graph Colouring problem consists of finding a colouring of the nodes such that any two adjacent nodes have different colours, and:

(SAT) using only colours of a set **{1, 2, ..., n}**; or
(OPT) minimizing the number of colours **n**.

Here we want to model a variant of this problem, assuming that there is an extra colour, **0**, that satisfies any constraint of difference (i.e., **0≠k**, or **k≠0** is satisfied, for any value of **k**, including **k = 0**).

The figure shows a solution with 4 colours (1 to 4) and no zeros. However, if only **2** colours, plus 0s are acceptable a minimal solution would require that at least two nodes A and E take value 0, and this is an optimal solution (there is no solution with colours **{0,1,2}** that uses less than 2 zeros).

The problem you should model is a mixed problem, such as to minimize the number of 0s in a colouring of the graph with colours in the set **{0, 1, ..., n}**.

Assume that the graph is given as the adjacency matrix (a Boolean matrix where a 1 represents a connection between the corresponding nodes). For the graph, the matrix would be (each row represents a node)

G = {{0,1,1,1,1,0}, {1,0,0,1,1,0}, {1,0,0,0,1,1}, {1,1,0,0,1,0}, {1,1,1,1,0,1}, {0,0,1,0,1,0}}

a) **(2 pt.)** Specify a model for this problem in Choco. More precisely, declare the decision variables you chose as well as their domains, together with the model and solver you propose.

b) **(4 pt.)** (SAT) What constraints would you consider to modelling the satisfaction problem, i.e., to find a solution that satisfies the problem (i.e., only uses the colours from 0 to n)? **Remember** : A zero is considered different from any colour, including the 0.

c) **(3 pt.)** (OPT) Complete the optimisation problem, i.e., to model a problem that aims at minimising the number of 0s in the solution.

Proposed Solution:

a) We consider an array C of decision variables, representing the nodes of the graph. The domains of the variables, the allowed colours to label the nodes are the range 0..n.
The choco code may thus be

```
int [][] G = {...}; // given
int n = G.length;
```

```
Model md = new Model();
Solver sv = md.getSolver();
IntVar [] C= md.intVarArray("N", n,  0, n);
```

b) First we have to define a constraint on any nodes x, y that imposes they have different colours, or one of the colours is 0. This can be specified as

```
md.or(md.arithm(x, "≠", y), md.arithm(X, "*", Y, "=", 0))
```

Now, we must post an instance of this constraint on any two adjacent nodes:

```
for (int i = 0; x < n; x++)
  for (int y = x+1; y <= n; y++)
    if (G[i,j]){
      md.or(md.arithm(C[i], '≠', C[j]), md.arithm(C[i], '*', C[j], '=', 0))
```

c) Now we have to "count" the number of 0s. We may use the predefined constraint count(v,A,c) that constrains the number of elements of the intVarArray A that take value v to be equal to intVar c. Hence we should declare a intVar variable c

```
IntVar cnt = md.intVar ("c", n+1,  0, n);
```

Then post the constraint

```
md.count(0, C, cnt).post()
```

and set the minimisation objective by invoking the model method

```
md.setObjective(Model.MINIMIZE, cnt);
```