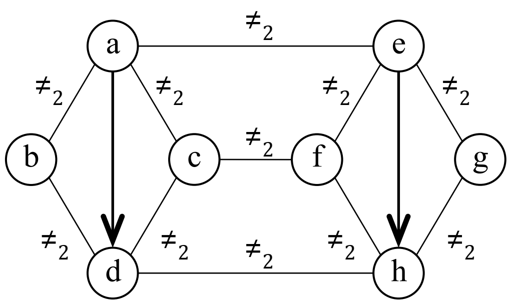
Constraint Programming

2016/2017 – Exam

Friday, 13 January 2017, 09:00 h

Part I – Finite Domains (1.5 h – open book)

1. Propagation (7 pts)



Consider the constraint network on the right, where nodes represent variables with domain {1,2,3,4}, directed arcs mean constraints of inequality (">") and arcs labelled $\neq_k$ denote constraints of *difference-k*, i.e. the values of the corresponding constraints should differ by at least k (i.e. the usual difference constraint ≠ is the special case $\neq_1$).

a) **(3 pt)** Show that the network is satisfiable. What are the possible solutions?

b) **(1 pt)** Show the pruning achieved in the domain of the variables by maintaining arc-consistency.

c) **(2 pt)** Show the pruning that would be achieved in the domain of the variables by maintaining path-consistency?

d) **(1 pts)** Would path-consistency be sufficient to guarantee backtrack free labelling of the variables? Justify.

Proposed Solution

a) Given the inequality constraints between variables **a** and **d**, their domains are narrowed to **{2,3,4}** and **{1,2,3}**, respectively. But **a** cannot be **3** since, on the one hand, variables **b** and **c** would take value **1** and, on the other hand, **d** would take one of the values **1** or **2**, none of which is compatible with the value 1 of variables **b** and **c**. If **a** takes value **2**, then **d** must take value **1** and **b** and **c** should take value **4**. If **a** takes value **4**, then **b** and **c** should take values **1** or **2.** But since **d** cannot be greater than **3,** then both variables **b** and **c** should in fact take value **1,** and **d** take value **3**. Thus the only solutions for variables **a**, **b**, **c** and **d** are **<2,4,4,1>** and **<4,1,1,3>**. Similarly, the only solution tuples for variables **e**, **f**, **g** and **h** are **<2,4,4,1>** and **<4,1,1,3>**. Since the constraint **≠2** on variables **c** and **f** enforce that they must be different, the only two complete solutions are **<2,4,4,1,4,1,1,3>** and **<4,1,1,3,2,4,4,1,>** .

b) Given the inequality constraints, maintaining arc-consistency would eliminate values **1** from variables **a** and **e** and value **4** from those of variables **d** and **h**. Arc-consistency over the remaining **≠2** constraints would not lead to any further pruning.

c) Path-consistency would impose some pruning on variable's domains. The only 2-labels with variables **a** and **d** that can be extended to variables **b** or **c** are **<a/2,d/1>** and **<a/4,d/3>** that can be extended to 3 labels **<a/2,b/4,d/1>** and **<a/4,b/1,d/3>**, leading to the pruning of the domains of a and d to respectively **{2,4}** and **{1,3}**, and similar pruning would occur to the domains of variables **e** and **h.** The remaining variables, **b, c, f** and **g** have their domains pruned to **{1,4}.**

d) Given the above pruning, it is easy to see that the assignment of one of values in *any* of the variables would propagate for one of the 2 complete solutions. For example, after the assignment **c=1** constraint propagation would lead to fixing **a = 2**, **d = 1** and **b = 4**, and also **f = 4**, **e = 2**, **h = 1** and **g = 4**.

2. Global Constraints (6 pts)

Spreading the execution of different tasks such that they do not start all at about the same time is often quite convenient and is the purpose of the global constraint **spread(S)**: given a vector of starting times, each in some of starting times spread constrains the starting times to be as evenly distributed as possible. In more detail, if S refers to n tasks, and all of them start in the range **lo..up** (time units), then the global constraint imposes that the difference in the starting times of any two tasks is at least **floor((up-lo)/(n-1))**. For example, if **5** tasks start between **1** and **22**, then the starting times of any two tasks should differ by at least **floor((22-1)/(5-1) = 5** time units (possible solutions have tasks starting at times **{1,6,11,16,21}** or **{1,6,11,16,22}**,or **{1,7,12,17,22}**. In some applications, tasks must also be sorted, say in increasing order, i.e. **S[i] ≤ S[i+1]**, and this may be imposed by another global constraint **sort(S).**

a) (2 pt) Show that even when both global constraints maintain domain consistency (or generalised arc-consistency), it is possible that their separate execution does not prune the domain of any of the decision variables in **S**, any more than each of them separately, in contrast with their combination in a single global constraint, **spread_sorted(S)**, maintaining domain consistency.

b) (3 pt) Assuming the constraint **spread** is available, implement the **spread_sorted** constraint by means of a function with signature,:

```
function void sorted_spread (Solver<CP> cp, var<CP> int [] S)
```

c) (1 pt) Does your implementation achieve domain-consistency? Justify.

Proposed Solution

a) It is easy to check that, if all variables have initial domain in the range **lo..up**, the **sort** constraint does not achieve any pruning, since a solution where all **S[i] = k** for **k** ∈ **lo..up** is consistent with the constraint. Hence, any value in that range belongs to a solution. On the other hand, the **spread** constraint may prune some of the values from the domain of the variables. For example, if the domain is **1..22** then no task could start at time **3**, since in this case, the other tasks should start no sooner than **8,13,18** and **23** enforcing the last task to start after the time limit. In fact, maintaining domain consistency in the spread constraint would prune the domain of all the decision variables, in this case, to the set **{1,2,6,7,11,12,16,17,21,22}**.

b) The **sorted_spread** constraint could be implemented by imposing inequality constraints with the bias computed from the spread constraint as follows:

```
function void sorted_spread(Solver<CP> cp, var<CP> int [] S){
    cp.post(spread(S));
    int n = S.getSize();
    int lo = min (i in 1..n) S[i].getLow();
    int up = max (i in 1..n) S[i].getUp();
    int k = floor((up-lo)/(n-1));
    forall(i in i..n-1) cp.post(S[i+1] >= S[i]+k);
}
```

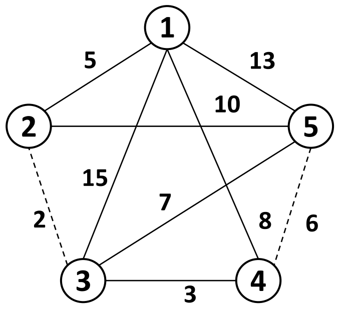
c) This implementation does guarantee domain consistency (provided the spread constraint maintains domain consistency and that **k > 0** (if **k = 0** we would have the same situation as before, no pruning would be achieved).

3. Modeling - Constrained Travelling Salesperson (7 pts)

Given a graph **<V,E>** where **V** is a set of **n** vertices and **E** is a set of undirected weighted arcs between the vertices, represented by a symmetric matrix of distances (positive integers), the goal is to obtain an Hamiltonian Tour of the graph with minimal length such that:

- The sum of the weigths of any two consecutive arcs does not exceed **maxPairs**.
- Some of the arcs are not acceptable. These are represented in a matrix **F**, with two columns and where each row **i** denote the vertices of the arcs $\mathbf{F_{i,1}} \leftrightarrow \mathbf{F_{i,2}}$ (both directions) that should not be used.

For example, the graph below has **5** vertices and matrices **E** , that represents it, and **F** representing forbidden arcs **2↔3** and **4↔5** are shown. The shortest Hamiltonian tour, satisfying `maxPairs = 20` is `1→4→3→5→2→1` (or `1←4←3←5←2←1`), with length `8 + 3 + 7 + 10 + 5 = 33.`



```
int E[1..5,1..5] = [
      [100,  5, 15,  8, 13],
      [  5,100,  2,100, 10],
      [ 15,  2,100,  3,  7],
      [  8,100,  3,100,  6],
      [ 13, 10,  7,  6,100]
     ];
int F[1..2,1..2] = [
      [2,3],
      [4,5]
     ];
```

Specify in Comet a function

```
function ??? tour(int [,] E, int [,] F, int MaxArcs, MaxPairs)
```

to solve this problem, that takes as input parameters the matrices of distances `E` and forbidden arcs `F`, and the maximum accepted values for `MaxPairs`, as discussed above, and outputs a solution in a format that is left open to the modelling you chose.

a) **(2 pts)** Specify the structures and variables you require to solve the problem. Your specification should in particular specify how the encoding used to represent the results.

   Note: In your model, you may use function `Constraint<CP> circuit(var<CP>{int}[] x)`, that creates a circuit constraint which holds if all the variables are assigned values representing an Hamiltonian circuit. More specifically, let `x` be an array `x[l],x[l+1],... ,x[u]` and `R` be `l..u`. Each element in `R` denotes a vertex and `x[i]` denotes the successor of vertex `i`. The constraint succeeds if the graph so formed is an Hamiltonian circuit, i.e. if it visits every vertex exactly once and connects all vertices.

b) **(5 pts)** Implement the constraints required for solving the problem with the variables declared in the previous item (i.e. the block `solve<cp>{…}`, or `minimize<cp> … subject to {…}`.

c) **(1 pts)** Specify a heuristic to solve the problem (i.e. the block `using {…}`). Justify the choice.

Proposed Solution

a) Adopting the circuit constraint, the only variables that need to be considered are the successor variables, **succ**, as used in this global constraint, as well as a variable, tour, that represents the length of the tour that is to be minimised.

The result, **sol**, is simply returned as an array of successor nodes (converted to integers).

```
function int [] tour(int [,] E, int [,] F, MaxPairs)
   int n = E.getSize();
   range Rng = 1..n;
   Solver<CP> cp();
      var<CP>{int} succ[Rng](cp,Rng);
      var<CP>{int} tour(cp, 0..System.getMAXINT());
   minimize<cp> tour subject to {
      ...
   } using {
      ...
   }
   forall(i in Rng) sol[i] = x[i];
   return sol;
}
```

b) The goals and constraints specified would be implemented with this encoding as follows:

- The succ variables represent an Hamiltonian tour:
  ```
  cp.post(circuit(succ));
  ```
- The tour to be minimised:
  ```
  cp.post(tour == sum(i in Rng) E[i,succ(i)];
  ```
- No forbiden arcs are used:
  ```
  forall(i in F.getRange(0))
     cp.post(succ[[F[i,1]] != [F[i,2]);
  ```
- No pair of consecutive arcs exceeds **maxPairs** length:
  ```
  forall(i in F.getRange(0))
     cp.post(maxPairs >= E[i,succ[i]]+ E[succ[i],succ[succ[i]]);
  ```

c) The simpler heuristic is the standard first-fail heuristic implemented on the **succ** decision variables (the other decision variable, **tour**, is implied)

```
labelFF(x);
```

One might try a heuristic that assigns arcs with least length first, although the success of this strategy might depend on the forbidden arcs, and is very much instance dependent.