Department of Mathematics and Computer Science
University of Southern Denmark, Odense

March 4, 2011
Marco Chiarandini

DM826 - Modeling and Solving Constrained Optimization Problems

Obligatory Assignment 2, Spring 2011

Deadline: 18th March 2011 at noon.

The first two exercises were posed at the Summer School on Constraint Programming organized in Aussois, France by the Association for Constraint Programming (May, 2010).

1 The Aircraft Scheduling problem

The CP model below is from an Aircraft Scheduling problem consisting in scheduling the landing time of planes in an airport.

```
1  import cotfd;
2
3  int NB_PLANES = 30 // number of planes
4  range Planes = 1..NB_PLANES; // plane range
5  range Time = 1..1000; // time range
6  int preferredTime[Planes] = ...; // preferred landing time of each plane
7  int forbiddenPeriod[Planes] = ...; // period during which the runway stays
       unavailable
8  int delayCost[Planes] = ...; // the earliness/tardiness cost
9
10 Solver<CP> cp(); // solver
11 var<CP>{int} time[Planes](cp,Time); // decision variable
12 var<CP>{int} costPlane[Planes](cp,0..100000); // cost for each plane
13 var<CP>{int} cost(cp,0..100000); // total cost
14
15 int startTime = System.getCPUTime();
16 minimize<cp>
17   cost
18 subject to {
19   forall (p in Planes) {
20     forall (p2 in Planes : p != p2) {
21       cp.post(time[p2] >= time[p] + forbiddenPeriod[p] || time[p2]+
             forbiddenPeriod[p2] <= time[p]);
22     }
23     cp.post(costPlane[p] == (abs(preferredTime[p] - time[p]) * delayCost[p]));
24   }
25   cp.post(cost == sum(p in Planes) costPlane[p]);
26 } using {
27   labelFF(time);
28 }
```

1. Write a 5 line description of the model (lines 16-25).

2. Execute the program on the instance provided at `http://www.imada.sdu.dk/~marco/DM826/Resources/Assignment2`.

   The search is very inefficient because the values are considered in increasing order from 0. This can be very far from the optimal solution. Change the search strategy using a **forall** selecting the variables and a **tryall** selecting the values.

   Compare it with the previous search strategy on the number of failures, the number of branchings and the execution time.

2 The 0–1 knapsack problem

Given a set of n objects of weight w_i and usefulness a_i ($1 \leq i \leq n$), select a subset maximizing the usefulness $\sum_{i=1}^{n} x_i a_i$ such that the capacity b of the backpack is not exceeded ($\sum_{i=1}^{n} x_i w_i \leq b$).

2.1 A Branch and Bound approach

The most efficient algorithms for knapsack problems are dynamic programming algorithms. However we will focus here on solving the problem by branch and bound.
Compare the computing time, the number of failures and the number of branchings of the models from the following points.

1. A model is given in the file `knapsack.co`, using the labelFF heuristic. Modify it in a way such that the program searches for the best solution.

2. Substitute the constraints with global constraints

3. Implement an effective search strategies for the selection of the next variable to assign when searching for this problem.

4. **Optimization by iterations** Now you will solve this problem by iterations. This means you will first consider the feasibility problem: does there exist a solution to the knapsack problem with at least a given usefulness u? Then you will use this version of the problem for different u's until you know there is no better solution than the ones you have already found. The simplest strategy is to first compute an upper bound ub on the usefulness of the optimal solution. The search begins with $u = ub$ and is restarted with a decreased u while no solution is found. An upper bound function is provided in `knapsack.co`.

   In Comet: To implement optimization by iteration in COMET consider the following procedure:

   - Add an `Integer u` to your program and modify the code so that the CP engine will search for a solution having at least the cost of the value stored in `u`. Do it by adding a constraint that binds the sum of the `usefulness` of the selected items to a variable `v` and label this variable to the value of `u` at the beginning of the **using** block. Do not put the constraint `totalUsefulness == u` in the solve block so that we can restart the solver (this means restarting the search without any change to the constraints) without creating a new solver. This is more efficient.

- Use the `onCompletion()` event to decrease the value of u by one. This event is triggered when the search has completed and no solution has been found. Use the `onFeasibleSolution(Solution)` event to stop the search as soon as a solution is found.

```
whenever cp.getSearchController()@onCompletion() {
    u := u - 1;
    cp.reStart();
}
whenever cp.getSearchController()@onFeasibleSolution(Solution s) {
    cout << "Solution : " << totalUsefulness.getSnapshot(s) << endl;
    cp.exit();
}
```

5. **Optimization by iterations and dichotomic search** if the upper bound is far from the optimal solution, the CP solver will spend much time searching for solutions with very high usefulness that do not exist.

   So we want to decrease u faster while preserving completeness. We can do that by modifying u dichotomically. We compute a lower bound lb and an upper bound ub of the usefulness of the optimal solution. Then we search for a solution having a usefulness $c \geq \lceil \frac{lb+ub}{2} \rceil$. If such a solution exists, then we update lb to the usefulness of the solution and loop. Otherwise u is a strict upper bound on the cost of the optimal solution; we update $ub := \lfloor \frac{lb+ub}{2} - 1 \rfloor$ and loop. We stop when $lb = ub$. At any point in time we have that $s^* \in [lb; ub]$ (s^* is the optimal solution of the problem).

   In Comet:

   - Create two `Integer ub` and `lb`. Initialize `ub` using the upper bound algorithm and `lb` using the given lower bound algorithm.
   - Use the events `onCompletion()` and `onFeasibleSolution(Solution)` to update the value of `ub` and `lb` accordingly.

3 Logical constraints in CP and IP

Consider the following CSP:

$$\begin{aligned}\mathcal{P} = \quad & \langle X \equiv (x_1, x_2, x_3), \mathcal{DE} \equiv \{D(x_1) = \{0,1\}, D(x_2) = \{0,1\}, D(x_3) = [-2..2]\}, \\ & \mathcal{C} \equiv \{(x_1 \vee x_2) \Rightarrow (x_3 \geq 0)\}\rangle\end{aligned}$$

1. *In CP.* Which is the arity of the constraint? How can arc consistency be enforced? Is it arc consistent? Which level of arc consistency does it satisfy?

2. *In IP.* Rewrite the constraint in a form that can be handled by IP solvers.