LINGI2365 : Constraint Programming

Assignment 3 :

Search

F.Aubry & Y. Deville

March 2014

1 Objective and practical details

The goal of this assignment is for you to understand and implement different search methods in Comet.

You should respect the following constraints

- Deadline : **Friday 21 March 2014 2pm**.
- This assignment is **mandatory**.
- This assignment must be completed **by groups of two** (the same groups as in the previous assignments).

Modalities

- A **hardcopy of your report** must be turned in (box in front of the INGI secretary) before 2pm on the 21 of March.
- Submit your files, **code and report pdf**, on iCampus in a zip (or tar.gz) with the name **tp1_gXY** where **XY** is your group number.
- In case of problems, write to `f.aubry@uclouvain.be`.

2 Problems

2.1 Read this first

We ask you to perform more extensive experiments in this assignment. Some of these will take some time to execute. **Start early on** and write a small script executing your instances for you. You will not want to watch them execute one by one!

Also, it is important that you complete the subproblems for the Knapsack Problem in order (2.3.1, then 2.3.2, then 2.3.3) since you will need the code from the previous subproblem to solve the current one.

In points 2.3.2 and 2.3.3 you are asked to use functions provided in the file `knapsackLBUB.co`. You can use those functions in your code by including this file. If you prefer to copy the functions into your own code, do not forget to add the `import cotls;` and `import cotln;` statements.

2.2 The Brussels airport problem (7 pts)

At Brussels airport the landing times of planes need to be decided. Each plane has a preferred landing time p_i. There is also a cost c_i associated with each plane. If a plane lands sooner or later than its prefered landing time, a penalty cost is incurred. The penality cost is given by $c_i \cdot (p_i - t_i)^2$ where t_i is the landing time of plane i.

After a plane has landed, no other plane may land for a given period b_i (depending on the plane that has landed). The goal is to minimize the sum of the costs incurred by early and late landings.

For each plane you are given:

- its preferred landing time (p_i)
- number of time units during which no other plane may land following its landing (b_i)
- a cost per early or late time unit (c_i)

A model for this problem is given in iCampus, `airport.co`.

You are asked to:

1. Explain the given model (1).
2. Design 2 different variable and/or value ordering heuristics for this problem. A simple `labelFF(cp)` does not count.
   - describe them and explain what is the reasoning behind them in your report (1.5)
   - implement them in Comet by adding the model (1.5)
3. Which criteria are meaningful for comparing different search strategies? Briefly explain those criteria in your report. (1)
4. Based on your criteria, compare your heuristics with the `labelFF` heuristic by testing them on the instance on iCampus. (1)

5. Consider the following strategy. Starting with $d = 0$ we compute the optimal solution with the additional constraint

$$-d \leq delay[i] \leq d$$

and we increase d by 1 until we reach some point where the value of the objective function does not change from d to $d+1$. Given an example with three planes where this strategy is wrong (1).

The instance file is structured as follows: first line = number of planes. Then one line for each plane with three integers p_i, b_i and c_i.

2.3 The Knapsack Problem (13 pts)

In this part of the assignment we will explore different methods to optimize the knapsack problem. You'll find benchmark instances for this problem on iCampus. In the knapsack problem, given a set N of n objects of weight w_i and usefulness a_i ($1 \leq i \leq n$), we want to find a subset of the objects such that the capacity b of the knapsack is not exceeded and such that the total usefulness is maximized. The problem can be formulated as follows:

$$\begin{aligned} &\textit{maximize} && \sum_{i \in N} x_i a_i \\ &\textit{subject to} && \sum_{i \in N} x_i w_i \leq b \\ & && x_i \in \{0,1\} \quad \forall i \in N \end{aligned}$$

2.3.1 A Branch & Bound approach (4.5 pts)

1. Model the knapsack problem as Constraint **Optimization** Problem in Comet. (0.5)
2. Describe your model in the report. (0.5)
3. Design 3 different heuristics (specific to the knapsack problem) for variable selection.
   - describe and explain them in your report (1)
   - implement them in Comet (1)
4. Test your heuristics and the `labelFF` heuristic on the `knapsack-A` instances. (0.5)
5. Present and discuss the results in your report. (1)

The `knapsack-X-zy.txt` files are structured as follows: first line = number of objects. One line per object : object id, object weight, object usefulness. Last line = capacity of knapsack.

Choose one of your search strategies, you will use it **for the remainder of this assignment**.

2.3.2 Optimization over iterations (4.5 pts)

In this point you are asked to optimize this problem over several iterations. The idea is to fix the value of the objective function (in our case the total usefulness) beforehand to *ub*, and to determine then if a feasible solution to the problem having *ub* as objective value exists. We are going to initialize *ub* with an upper bound on the total usefulness. The value of *ub* will be decreased over the iterations.

1. Model the knapsack problem as a Constraint **Satisfaction** Problem. (0.5)
2. In order to implement the optimization over iterations you will have to:
   (a) Add a decision variable `totalUsefulness` corresponding to the usefulness of a solution to your model (if you don't already have such a variable).
   (b) Add an `Integer ub` (Integer not int!) to your program
   (c) Use the function `getUB(int n, int b, int[] weight, int[] usefulness)` (provided in `knapsackLBUB.co`) to initialize `ub` to the upper bound value.
   (d) Add a constraint forcing `totalUsefulness` to take the value of `ub` at the beginning of the `using` block. We do not put this constraint inside the `solve` block because we want to be able to restart the search after changing `ub` without having to recreate a new solver and having to repost all the constraints.
   (e) Make use of events. The event `cp.getSearchController()@onCompletion()` is triggered when the search is complete and no solution has been found.
   The event `cp.getSearchController()@onFeasibleSolution(Solution s)` is triggered when a solution has been found. You can see how events are used in the file `EventsDemo.co` on iCampus. In order to find an optimal solution to the knapsack problem you need to use these events to:
      i. modify the value of `ub`
      ii. restart the search (using `cp.reStart()`)
      iii. end the search (using `cp.exit()`)
3. Which of these points (i., ii., iii.) do you need to execute on which events? Explain in the report and implement it in Comet. (1.5)

4. How do you modify the value of `ub` to be sure to find the optimal solution? (0.5)

5. Can you explain why we initialize `ub` with an upper bound instead of any other value? (0.5)

6. Experiment this program on the instances `knapsack-A, -B`. (0.5)

7. Present and discuss the results in your report. (1)

2.3.3 Optimization via divide and conquer (4pts)

Using optimization over iterations the search for an optimal solution may take a long time if the optimal solution is far from the upper bound. Before we checked for every potential value of `totalUsefulness` (starting at the upper bound) if a feasible solution existed, and only stopped if we found one. Now we want to divide and conquer! That means that we want to divide the interval of potential values `totalUsefulness` could take in the optimal solution and eliminate uninteresting intervals as soon as possible.

1. In order to implement the optimization via divide and conquer you will have to:

   (a) Add an `Integer lb` to your previous program

   (b) Use the (non-deterministic!) function `getLB(int n, int b, int[] weight, int[] usefulness)` (provided in `knapsackLBUB.co`) to initialize `lb` to the lower bound value. Note that you can use the "-d" flag to force Comet into deterministic mode.

   (c) Modify your program s.t. `totalUsefulness` is forced to lie in the interval $[\lceil \frac{\texttt{lb}+\texttt{ub}}{2} \rceil, \texttt{ub}]$. Note that you can use the `ceil(v)` and `floor(v)` functions to round a `float v` up or down.

   (d) Use the events `@onCompletion()` and `@onFeasibleSolution(Solution s)` to update your `lb` and `ub` variables. You will need to:

      i. update `lb` to the value of the current solution
      ii. update `ub` to $\lceil \frac{\texttt{lb}+\texttt{ub}}{2} - 1 \rceil$
      iii. restart the search
      iv. stop the search when a specific condition is fulfilled

2. Which of these points (i., ii., iii., iv.) do you need to execute on which events? Explain in the report and implement it in Comet. (2)

3. Experiment this version on the instances `knapsack-A,-B,-C`. (1)

4. Present and discuss the results in your report. (1)