Constraint Programming Exam

Nicolas Barnier

Duration: 2h – All documents allowed – Access to the Web is forbidden

The **commented** source code corresponding to your answers must be uploaded **before 10:00**. Follow the submission link in the exam section of the course page on e-campus.

This subject has 2 pages.

Map Scan

To plan an outdoor trek, you are provided with a large topographic map which you cannot take with you, a scanner and a printer, so that you can reproduce enough parts of the map to cover your whole path, described as a sequence of n points (the red dots in figure 1). The scanner works on a square area with side length c and we suppose that the difference between the coordinates of two consecutive points is always less or equal to c: $\forall i \in [1..n-1]$, $|x_i - x_{i+1}| \leq c \wedge |y_i - y_{i+1}| \leq c$.

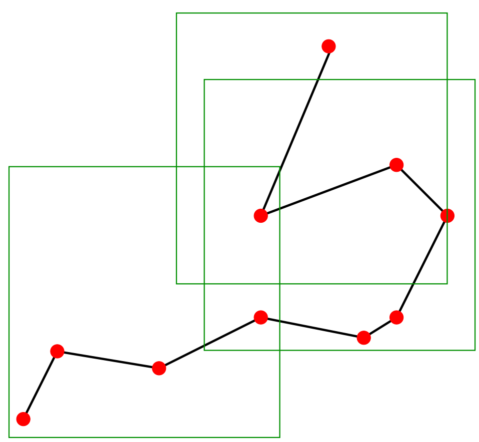

Figure 1: A solution with 3 maps for a problem with 10 points.

To be more handy, **the scanned squares must be located so that each segment of the path is entirely included in a single square** as in figure 1 where, for example, the first square covers the first three segments (note that a segment can be covered by more than one square):

$$\forall i \in [1..n-1], \exists j \text{ s.t. } p_i \in s_j \wedge p_{i+1} \in s_j$$

with p_i a point of the path and s_j a square area.

Data for four problems are provided in files `trek*.txt` where the first line specifies the number of points n and the side length c of the scanner, and the n remaining lines are the coordinates of the points. Read and write functions, as well as a function returning the minimal and maximal coordinates of an array of points are also provided in file `trek.ml`:

```
val read : string -> pb
val fprint_sol : string -> pb -> Fd.t array -> Fd.t array -> unit
val extreme_coordinates : (int * int) array -> int * int * int * int
```

After `fprint_sol` is executed, the corresponding solution file (e.g. `sol.txt`) can then be printed by the `gnuplot` program with the following command:

```
plot [-1:][-1:] "sol.txt" w l
```

1. **[1pt]** What is the **maximal number** of squares required to cover a path of n points in the worst case? Write your answer as a comment in your source code.
2. **[2pt]** We now assume that we always have this maximal number of squares in a solution, some of them possibly unused (depending on the instance and the quality of the solution). Define **function** `solve: pb -> unit` which will solve problem `pb` and reports the result (so as to be able to represent a solution graphically). In function `solve`, define the **decision variables with appropriate domains** required to specify a solution to the map scan problem.
3. **[2pt]** With reifications, define **auxiliary boolean variables** $b_{i,j}$ to represent that p_i lies in square s_j.
4. **[2pt]** Using variables $b_{i,j}$, define new **auxiliary boolean variables** $seg_{i,j}$ to represent that segment $[p_i, p_{i+1}]$ is included in square s_j.
5. **[2pt]** Using variables $seg_{i,j}$, post the **main constraints** of the problem (each segment included in one square at least).
6. **[2pt]** Write a **search goal** that first assigns all the $seg_{i,j}$, then the coordinates of the squares.
7. **[3pt]** To travel light, we wish to use as few squares as possible. Define the **cost of the solution** as the number of used squares, i.e. the number of squares that cover at least one point, using new **auxiliary boolean variables** $used_j$ to represent that at least one point lies inside square s_j (using reification of course...). Adjust the **domains of the decision variables** to allow some of the squares to be placed (just) far enough so that no point belongs to them.
8. **[2pt]** Solve the search goal to find an **optimal solution** and check it graphically (e.g. using `fprint_sol`).
9. **[2pt]** Break the **permutation symmetry** among the squares.
10. **[2pt]** Post **constraints to place all unused squares** at the same location (just far enough to ensure that no point is covered) – using reifications once more...