2nd Practical Class – Backtracking (*Algoritmos de Retrocesso*)

Instructions

- Download the zipped file **CAL-Quickstart.zip** from the course's Moodle area and unzip it (it contains the folder **lib**, as well as the folders for each respective lab classes, named **TP<class number>** (e.g. TP1, TP2, …). The must be edited accordingly as new lab classes (TP* folders) are added.
- In the CLion IDE, open a project, selecting the folder that contains the files mentioned in the previous bullet point.
- Do "*Load CMake Project*" over the file *CMakeLists.txt*
- Run the project (**Run**)

Exercises

1. Labyrinth *(Labirinth.h, Labirinth.cpp)*

The objective of this exercise is to find the exit of a 10 by 10 labyrinth. The initial position is always at (1, 1). The labyrinth is modeled as a 10x10 matrix of integers, where a 0 represents a wall, a 1 represents free space, and a 2 represents the exit.

a. Implement the *findGoal* function (see Labirinth.h and Labirinth.cpp), which finds the way to the exit using backtracking algorithms. This function should call itself recursively until it finds the solution. In each decision point in the labyrinth, the only possible actions are to move left, right, up or down (see Labirinth.h and Labirinth.cpp). Once the exit is found, a message should be printed to the screen. It is suggested that you use a matrix to keep track of the points which have been visited already.
b. What is the time complexity of the algorithm in the worst case?

2. Sudoku *(Sudoku.h, Sudoku.cpp)*

Sudoku [http://en.wikipedia.org/wiki/Sudoku] is a game in which the objective is to fill a 9x9 matrix with numbers from 1 to 9, without repeating numbers in any row, column or 3x3 blocks.

a. Implement the *solve* function which should automatically (and efficiently) solve Sudoku of any degree of difficulty. Your solution should be based on a backtracking algorithm (see skeleton of a backtracking algorithm shown in the lectures). The algorithm should work recursively. That is, it should fill in a cell and call itself to solve the remaining puzzle. Use the following greedy algorithm to choose the cell to fill in: choose the cell with the minimum number of possible values (ideally 1).
b. Implement and test a function capable of determining the multiplicity of solutions (no solution, one solution or more than one solution) of a given Sudoku. Suggestion: adapt the *solve* function.

c. Implement and test a function capable of automatically generating Sudoku. Suggestion: starting with an empty Sudoku, randomly choose a cell and a number; if the cell is not filled in and the chosen number is valid for that cell, fill it in and analyze the multiplicity of solutions; if the Sudoku becomes impossible, clear the cell; if it has one solution, terminate; otherwise, continue the process.

3. Change-making problem (backtracking)

Considering the same description for the change-making problem of exercise 3 of the first practical class, implement the function *changeMakingBacktracking* below using a backtracking strategy instead.

```
bool changeMakingBacktracking(unsigned int C[], unsigned int Stock[],
      unsigned int n, unsigned int T, unsigned int usedCoins[])
```

For example: **C** = [1, 2, 5, 10], **Stock** = [3, 5, 2, 1], *n*=4, *T* = 8
Result: [1, 1, 1, 0]

For example: **C** = [1, 2, 5, 10], **Stock** = [1, 2, 4, 2], *n*=4, T = 38
Result: [1, 1, 3, 2]

4. Activity selection (backtracking)

Considering the same description for the activity selection problem of exercise 6 of the first practical class, implement the function *activitySelectionBacktracking* below using a backtracking strategy instead.

```
vector<Activity> activitySelectionBacktracking(vector<Activity> A)
```

For example: A = { a_1(10, 20), a_2(30, 35), a_3(5, 15), a_4(10, 40), a_5(40, 50) }
Result: {a_1, a_2, a_5} OR {a_3, a_2, a_5}