O COMP 215 Algorithms Θ

Lab #7

This work should be submitted electronically before 11:59pm Friday. Submit a .zip file that contains all the file with your code (preferably a single file lab7.cpp, but you can make it multiple files if you prefer).

This is a first draft of what you will be required to program for Lab #7. This should allow you to start working on the lab early, given that 1 hour will be borrowed from our regular lab time this Wednesday for lecture material.

In this lab, you will be implementing the graph algorithms Depth First Search and Breadth First Search.

First, let's create a class to hold your graph.

1. Create a class `mygraph`. This class will need to contain at least a public integer `nbVertices` holding the number of vertices in the graph, and a public adjacency matrix `adjMatrix`. Note that since the number of nodes is not known upfront, the adjacency matric will have to be dynamically allocated. There is unfortunately not any good way of allocating two-dimensional matrices dynamically, so `adjMatrix` will have to be of type int** (pointer to a pointer to an int). The allocation process is explained in the next point

2. The default constructor for your graph should initialize `nbVertices` to zero and `adjMatrix` to NULL. You should also have a constructor that takes an integer, initializes `nbVertices` to this integer and allocates an `nbVertices` $\times$ `nbVertices` adjacency matrix as follows:

   ```
   adjMatrix = new int*[nbVertices];
   for (i=0; i < nbVertices, i++)
         adjMatrix[i] = new int[nbVertices];
   ```

   After this, you will be able to access the element from the i^{th} row and j^{th} column of the matrix in the usual way, by accessing `adjMatrix`$[i][j]$

3. Deleting all this stuff will also be a bit complicated. Write a destructor for graph that does the following:

   ```
   for (i=0; i < nbVertices, i++)
         delete[] adjMatrix[i];
   delete[] adjMatrix;
   ```

4. Of course, the class will also contain public functions `DFS` and `BFS`, as well as private functions `dfs` and `bfs`, which do as described in the book. You are also allowed to put any other private elements in the class which you feel are required for your implementation of the Depth First Search and Breadth First Search algorithms (possibly an integer `count` and arrays of integers `mark` and `markPrime` as indicated in class, unless you prefer to make those actual global variables).

Some of you may note that instead of dynamically allocating this mess, we could have used vectors instead. Yes, that is true, but I want you to have to experiment with this dynamic allocation of multidimensional arrays at least once.

Then, we need to input the graph from a file.

5. Ask the user for a file name. Open that file, read it and create a graph object corresponding to the description in the file. The first line in the file should be the number of vertices `nbVertices` in the graph, and each following `nbVertices` lines in the file should contain `nbVertices` integers (all these integers should be either 0 or 1, a 0 in position (i, j) means that there is no edge between vertex i and vertex j, a 1 in position (i, j) indicates that there is an edge between vertex i and j.)

Finally, the *pièce de résistance*:

1. Implement the functions `DFS`, `BFS`, `dfs` and `bfs` in any way you see fit. None of the functions are required to return anything, but before exiting, the functions `DFS` should print two lines on screen, the first one indicating the order in which the vertices were added to the DFS graph, and the second indicating the order in which the vertices were popped from the stack; the function `BFS` should print one line on screen indicating the order in which the vertices were added to the BFS queue. (That is, the first vertex of the line should be the vertex "marked" 1, the second vertex of the line should be the vertex "marked" 2, the third vertex on the line should be the vertex "marked" 3, etc.)