3rd Lab Class – Part B – Dynamic Programming applied to graphs

Instructions

- Download the zipped file **DA_TP03b_unsolved.zip** from the course's Moodle area and unzip it. It contains the folder **TP3b**, with the *.h* and*.cpp* files, as well as files with datasets of points needed for this lab class.
- Copy the whole folder to the root of the CLion project provided in the first lab class.
- Edit the *CMakeLists.txt* in the root of the project, so as to set up an executable run/debug configuration for **TP3b**, accordingly:
  - Add the source files for the **TP2a** class:
    `file(GLOB TP3b_FILES CONFIGURE_DEPENDS "TP3b/*.cpp")`
  - Add executable target with source files listed in *TP2b_FILES* variable:
    `add_executable(TP3b main.cpp ${TP3b_FILES})`
  - Link the Google Test library to the *TP2b* target:
    `target_link_libraries(TP3b gtest_main gmock_main)`
- Do "*Load CMake Project*" over the file *CMakeLists.txt*. The **TP2b** configuration should now appear and be available from the "*Select Run/Debug Configuration*" drop-down list, on the upper-right corner of the CLion IDE.
- Compile and run the project after selecting the **TP2b** configuration.
- Implement your solutions in the respective *.cpp* file of each exercise.
- IMPORTANT: in case you need to read text files in I/O mode, you should tell CLion where such files are, by redefining the IDE environment variable "Working Directory", through menu Run > Edit Configurations… > Working Directory.

Exercises

IMPORTANT: The template **Graph** class provided already implements some single-source shortest path algorithms, such as Dijkstra's. Since the STL does not support mutable priority queues, you can explore and use the ***MutablePriorityQueue*** class also included, as follows:

- To create a queue: `MutablePriorityQueue<Vertex<T> > q;`
- To insert vertex pointer ***v***: `q.insert(v);`
- To extract the element with minimum value (*dist*): `v = q.extractMin();`
- To notify that thee key (*dist*) of ***v*** was decreased:: `q.decreaseKey(v);`

1. Single-source shortest path using Dynamic Programming: Bellman-Ford Algorithm

a) Implement the following public method in the **Graph** class:

```
void bellmanFordShortestPath(const T &origin)
```

This method implements the Bellman-Ford algorithm to find the shortest paths from ***v*** (vertex which contains element ***origin***) to all other vertices, in a given weighted graph.

b) Implement the following public member function in the class **Graph**:

```
vector<T> getPath(const T &origin, const T &dest)
```

Considering that the ***path*** property of the graph's vertices has been updated by invoking a shortest path algorithm from one vertex *origin* to all others, this function returns a vector with the sequence of the vertices of the path, from the *origin* to ***dest***, inclusively (***dest*** is the attribute ***info*** of the destination vertex of the path). It is assumed that a path calculation function, such as `unweightedShortestPath`, was previously called with the *origin* argument, which is the origin vertex.

2. All-pair shortest paths using Dynamic Programming: Floyd-Warshall Algorithm

a) Implement the following public method in the **Graph** class:

```
void floydWarshallShortestPath()
```

This method implements the Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices in the graph.

Additionally, you will also have to implement the following public method of the **Graph** class:

```
vector<T> getfloydWarshallPath(const T &origin, const T &dest)
```

This method returns a vector with the sequence of elements in the graph in the path from ***orgin*** to ***dest*** (where ***origin*** and ***dest*** are the values of the ***info*** member of the origin and destination vertices, respectively).