CSE221: Data Structures
Programming Assignment 4
Due on Friday 10 December, 23:59

Antoine Vigneron

1 Introduction

In this assignment, you are asked to:

- Implement a directed graph class, which is a slightly modified and simplified version of the textbook's.
- Implement Dijkstra's algorithm to find a shortest path in such a graph

2 Graph class

The directed graph class is similar to the adjacency list structure presented in Lecture 19, and thus Section 13.2.2. from the textbook. The main differences are the following: (See Figure 1.)

- All the edges are directed.
- There is no edge list E.
- An edge object does not record a reference to its position in the two incidence collections of its two endpoints.

The main class `Graph` records a graph, including its vertices and edges, as well as various member functions. It has only two data members: The size n (i.e. the number of vertices) and a list `vertexCollection` of all its vertices

The structure `VertexObject` is a private member of the `Graph` class that records the data related to a vertex v:

- A string `v.id`, which identifies the node to the user. For example, in the graphs representing road networks, Id could be the name of a city, i.e. "Ulsan".
- A list of outgoing edges `v.outgoingEdgeList` that records all the edges whose *origin* is v. So as opposed to the structure in Lecture 19, we do *not* record in these lists the edges whose destination is v.
- An auxiliary integer `aux` which you can use freely when implementing graph algorithms. For instance, if you wanted to implement DFS, it could be used to mark visited vertices. You may find this field useful when implementing Dijkstra's algorithm.

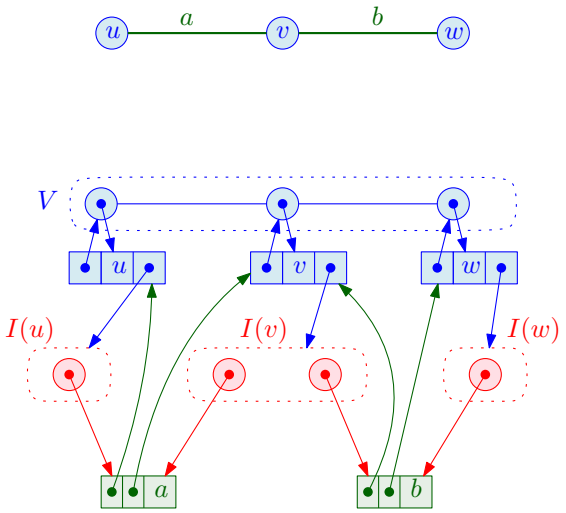

Figure 1: Directed graph data structure.

The `VertexObject`s are represented by blue rectangles in Figure 1.

An instance of the `Vertex` class is essentially a pointer to a `VertexObject`. This way, we can manipulate vertices without copying `VertexObjects`, which would imply copying the lists of outgoing edges. Instances of the `Vertex` class are represented by the pointed blue circles in Figure 1.

The edges are instances of the `Edge` class. It records its origin and destination `Vertex`, as well as a data field, which is a `double` floating-point number called its weight. In particular, in Dijkstra's algorithm, this weight is the length of the edge.

3 Implementation

You are given a header file `graph.h`, which gives the definition of the graph class and the main functions. You should fill in the parts marked "FILL IN WITH YOUR CODE". You should write the definition of the `Dijkstra` function in the `graph.cpp` file. In general, the various parts of the code that you need to fill in are explained in the code itself, and are similar (with the modification presented in the section above) to the member functions presented in class and in the textbook. Here are some more specific instructions.

3.1 Member functions

The function `isAdjacentTo` is slightly different from the version in the textbook. More precisely, u.`isAdjacentTo`(v) is true if and only if the edge (u, v) with origin u and destination v is present in the graph. In particular, whether the edge (v, u) with origin v and destination u is present, does not affect the the result of u.`isAdjacentTo`(v).

`operator==` for `Vertex` checks whether two `Vertex` instances refer to the same `VertexObject` instance. So you may create two `Vertex` instances v_1, v_2 pointing to the same `VertexObject` vobj. In this case, $v_1 == v_2$ returns true.

`insertVertex(const string& id)` inserts a new vertex, with no edges incident to it. Its `id` is given as the argument, and the `aux` field is not initialized.

`insertEdge(double weight, Vertex origin, Vertex dest)` inserts an edge from vertex `origin` to vertex `dest`, with given weight. In particular, the two vertices `origin` and `dest` must already be present in the graph.

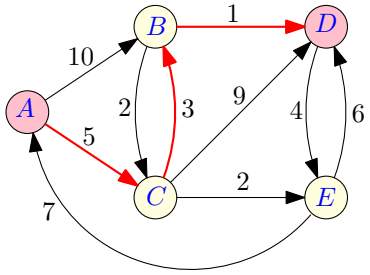

Figure 2: The graph recorded in `instance1.in`. The path goes from A to D.

3.2 Reading an input file

The file `instance1.in` records the graph pictured in Figure 2. It also records the index of the starting point s and destination t of the shortest path we must compute.

- The first line records n, m, s and t, where n is the number of vertices, m is the number of edges, s is the index of the starting point, and t is the index of the destination point.
- The next n lines record the n vertices. Each line contains the vertex index, which is an integer from 0 to n-1, followed by its string `id`. This string will be recorded in the `id` field in the corresponding `VertexObject`.
- The next m lines record the m edges. Each line contains an integer, that gives the index of the origin vertex, then the index of the destination vertex, and then a floating point number which is the weight of the edge.

Two other examples of input files, provided in the instances subdirectory, are `instance2.in` and `instance3.in`. They both represent the graph from Figure 3, which is a (not perfectly accurate) representation of the Korea's highway network. The edges, which represent roads, are undirected, so two copies appear in the file, one in each direction. So for instance there is an edge from Ulsan to Busan with weight 56.4, and an edge from Busan to Ulsan with the same weight 56.4. This weight represents the edge length. The difference between these two files is that `instance2.in` asks for a path from Ulsan to Seoul, and `instance3.in` asks for a path from Gangneung to Gwangju.

The function `G.readFile(filename, s, t)` reads an input file as described above, and constructs the corresponding graph, inside the graph instance G which is initially empty. If G was not empty, it should throw a `runtime_error`. The starting and destination points and are recorded in the argument vertices s and t.

3.3 Dijkstra's algorithm

You need to implement Dijkstra's algorithm in the file `graph.cpp`. A few remarks:

- You should implement your algorithm so that it runs in $O((m+n)\log n)$ time, otherwise you may lose marks for the largest examples.
- You can use standard data structures such as STL lists, vectors, priority queues ...
- The auxiliary field `aux` recorded at each vertex of the Graph data structure may be useful (although some solutions do not use it).

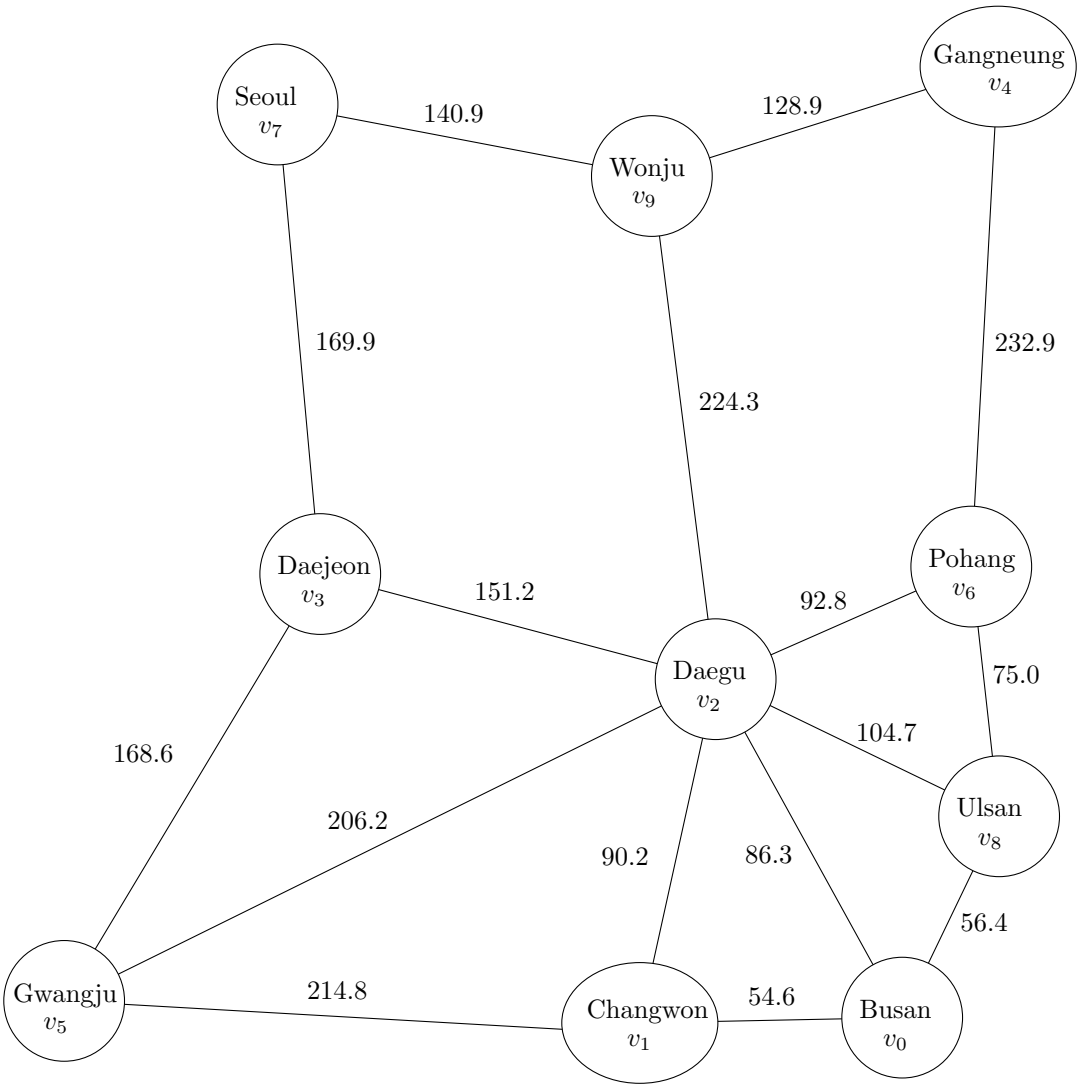

Figure 3: The graph recorded in `instance2.in` and `instance3.in`.

- One issue if you implement the algorithm with an STL priority queue is that this data structure does not support the update of the key of an element recorded in it. One way to bypass this problem is that, instead of updating the key $D[u]$ of a vertex u, you may just insert a new copy of u with the new key $D[u]$. At the same time, you maintain a flag telling you whether u has already been erased from the queue. After this flag has been set, the next time you erase u from the queue, you can just ignore it and move to the next smallest element in the queue.

Your algorithm should return a vector of strings, where the strings are the `id`s of the vertices along this shortest path. So in the example of Figure 2, this vector should be equal to {"A", "C", "B", "D"}.

4 Testing your program

Checking programs for all the functions are provided in the "code" subdirectory. Please read the README.txt file that is provided. It is highly recommended that you run all the tests on the server as your own configuration may differ in certain aspects. (A recent Ubuntu installation should work though.)

Input files recording graph instances are also given in the subdirectory "instances". They are used by the checking programs. Larger instance files will be provided in a few days so that you can check your program more thoroughly.

5 Submission

You should submit your code by simply uploading your `graph.h` and `graph.cpp` files to your **home directory** on the server, as usual.

6 Grading

Your program will be graded automatically. Make sure that the provided test programs compile and run without error on your code. If not, it will make it considerably more difficult to grade your program, and there will be a 10% penalty. If your program does not compile, you may want to comment out the functions that cause it not to compile.

Late submissions up to one day after the deadline will be penalized 15%, and 30% two days after the deadline. After that, your assignment will not be graded (hence you get 0).

The grading program we will use in the end may differ from the ones that have been/will be provided. In any case, some of the testing instances will be different.

7 Other rules

The general rules for the CSE department, given at the beginning of the semester (see course information folder), apply to this assignment. You may contact one of our two TAs for help:

- Seonghyeon Jue (`shjue@unist.ac.kr`)
- Hyeyun Yang (`gm1225@unist.ac.kr`)

Hyeyun Yang will be grading this assignment.

8 Typos and bugs

If you find typos in this handout, or bugs in the code that is being supplied, please send email to the TAs or to me.