Challenge 3: Graph Theory

Tianye Zhao

August 8, 2019

1 Programming Project

1.1 Code explanation

In this program, the first major data structure is a map named $vertexMap$ that allows us to find, for any vertex, a pointer to the $Vertex$ object that represents it.
The second major data structure is the $Vertex$ object that stores information about all the vertices.

(a) Vertex

A $Vertex$ object maintains four pieces of information for each vertex.

- $number$: The number corresponding to this vertex is established when the vertex is placed in map and never changes.
- adj: The list of adjacent vertices is established when the graph is read.
- $dist$: The length of the shortest path from the starting vertex to this vertex is computed by Dijkstra's algorithm.
- $prev$: The previous vertex on the shortest path to this vertex.
- $visited$: We use this variable to record whether this this vertex has been visited or not during implementing Dijkstra's algorithm.
- $reset$: This function is used to initialize the data members that are computed by the Dijkstra's algorithm.

(b) Edge

The $Edge$ consists of a pointer to a $Vertex$ and the edge cost.

(c) digraph

In the $digraph$ class interface, $vertexMap$ stores the map. The rest of the class provides member functions that perform initialization, add vertices and edges, save the shortest path.

- Constructor: The default creates an empty map.
- Destructor: It destroys all the dynamically allocated $Vertex$ objects.

- $getVertex$: This method consults the map to get the $Vertex$ entry. If the $Vertex$ does not exist, we create a new $Vertex$ and update the map.
- $addEdge$: This function gets the corresponding $Vertex$ entries and then update an adjacency list.
- $clearAll$: Initialize the members for shortest-path computation using Dijkstra's algorithm.
- $getPath$: This routine returns the shortest path after the computation has been performed. We can use the $prev$ member to trace back the path, it can give the path in order using recursion. The routine performs checking if a path actually exists and then returns inf if the path does not exist. Otherwise, it calls the recursive routine and returns the cost of the path.

(d) Path

This object is placed on the priority queue. It consists of the target vertex and its distance and a comparison function defined on the basis of the distance from start vertex.

(e) Dijkstra's SSAD algorithm

The $SSAD$ function performs shortest-path calculation using Dijkstra's algorithm. We use a method that works with the STL priority queue. This method involves inserting an $Path$ object in the priority queue whenever we lower the distance. To select a new vertex v for visitation, we repeatedly remove the minimum item based on distance from the priority queue until an unvisited vertex emerges.

(f) main

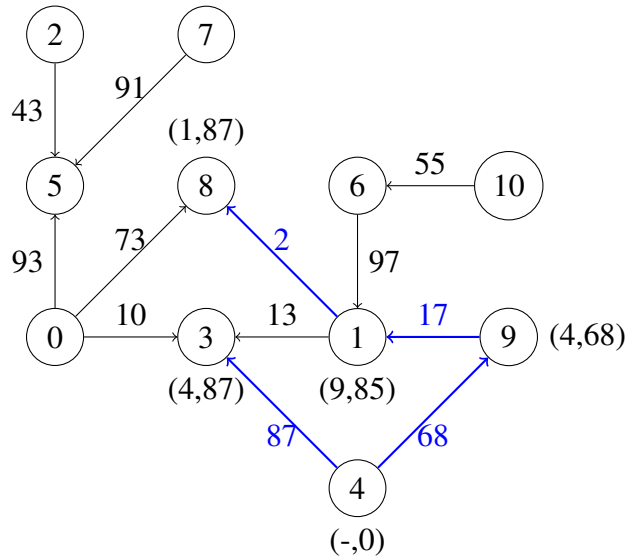
In main function, we provide a simple program that reads a graph in adjacency matrix form from an input file named "$File.txt$", reads in the number of vertices and a start vertex, then runs Dijkstra's SSAD algorithm. To construct the $digraph$ object, we repeatedly read one line of input, assign the line to an $istringstream$ object, parse the line, and call $addEdge$. Using an $istringstream$ allows us to verify that every line has at least the $|V|$ pieces corresponding to an vertex.
Once the graph has been read, we call $SSAD$ to apply Dijkstra's algorithm for a starting vertex. This algorithm throws a $digraphException$ if there is any error during execution. It catches any $digraphException$ that might be generated and prints an appropriate error message.

1.2 How to run

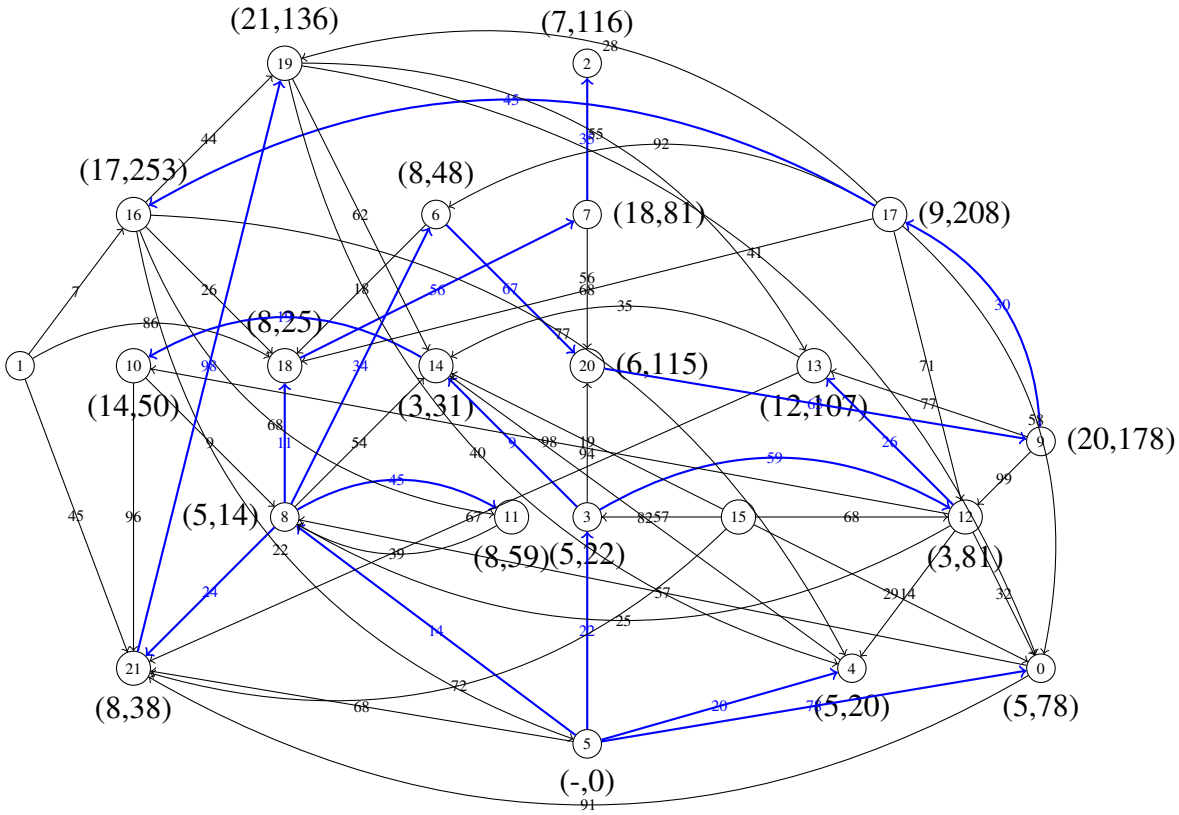
Name your input file as "$File.txt$", then run this program, it will generate text file "$Result.txt$" which contains desired output.

1.3 Input file specification

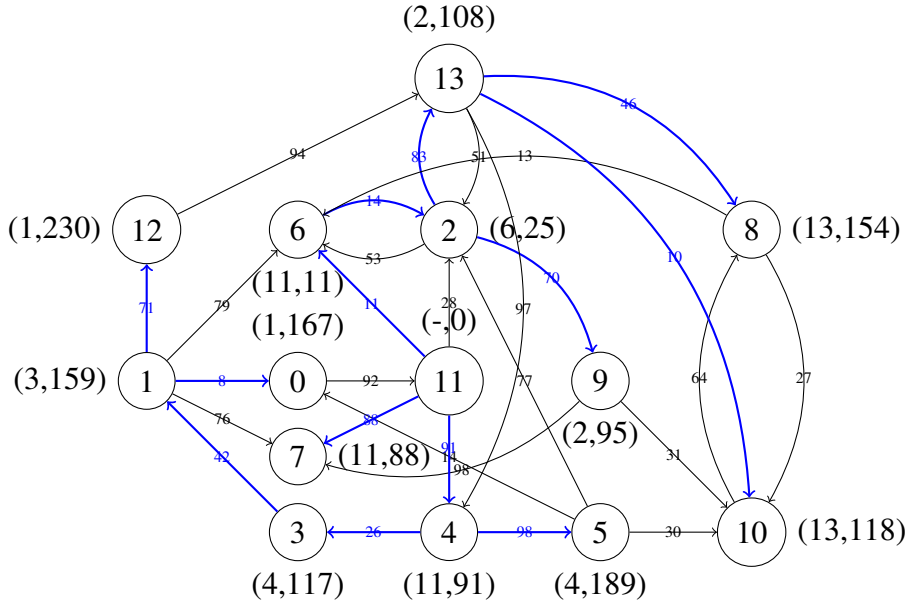
(a) File1.txt



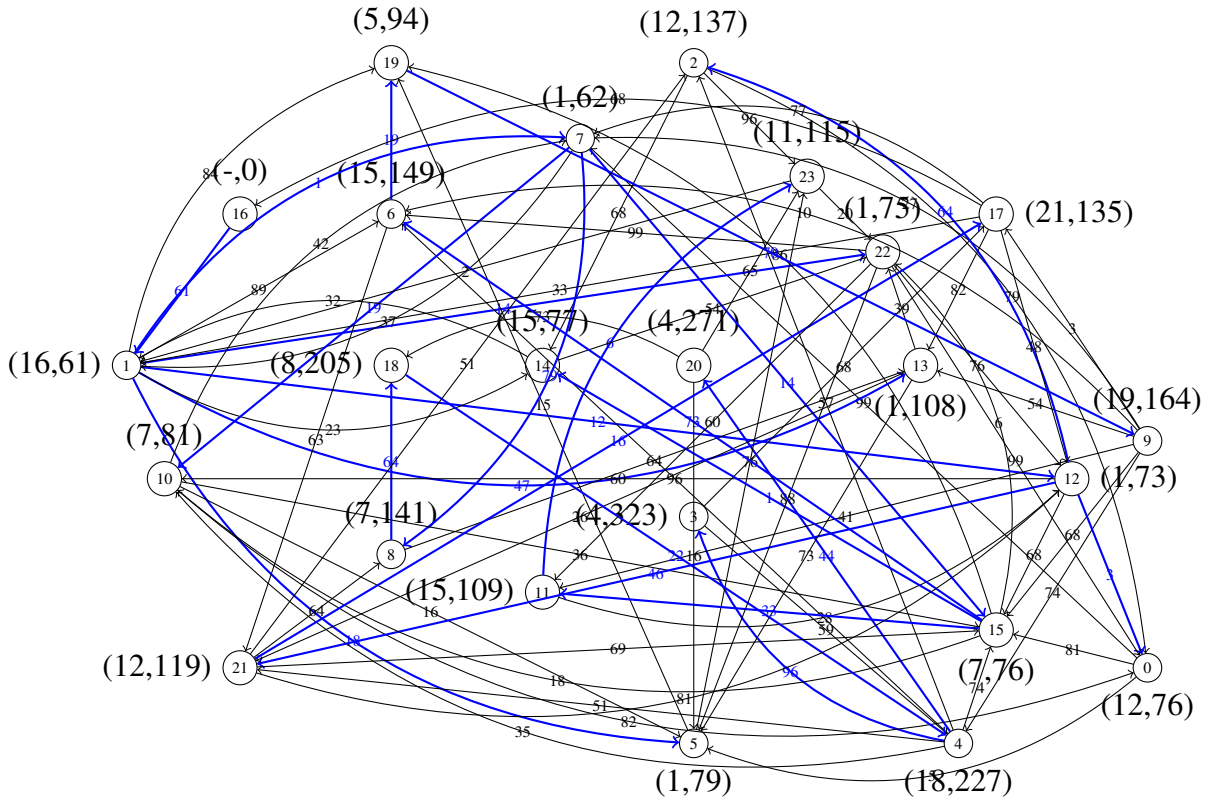
(b) File2.txt



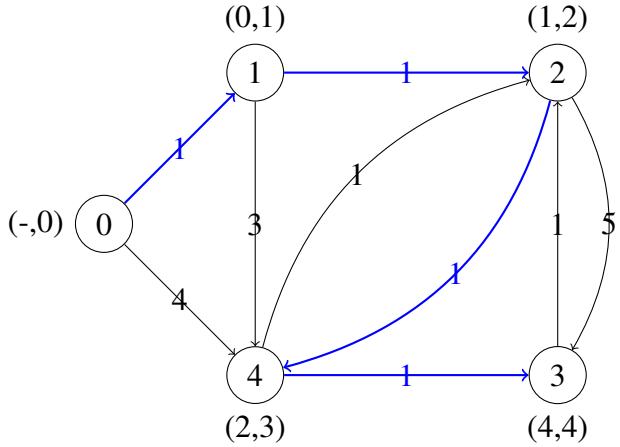
(c) File3.txt



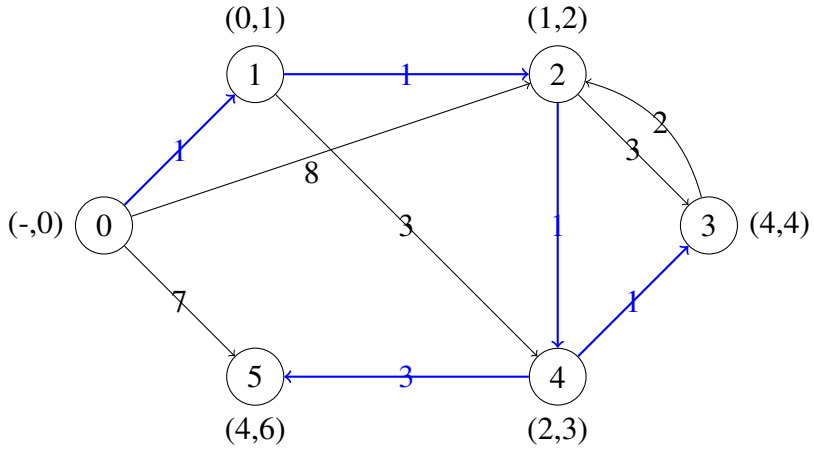
(d) File4.txt



(e) Add1.txt



(f) Add2.txt



(g) Add3.txt