CS590 Challenge 3: Graph Theory

Tianye Zhao, Xiaoping Yu, Yunxiu Zhang

August 7, 2019

1 Theoretical Questions

1.1 Define the following concepts

(a) Undirected graph (ungraph)

In an undirected graph $G = (V, E)$, the edges set E, consists of unordered pairs of vertices, that is, an edge is a set (u, v), where $u, v \in V$ and $u \neq v$. In an undirected graph, self-loops are forbidden, and so every edge consists of two distinct vertices.

(b) Directed graph (digraph)

A directed graph (or digraph) G is a pair (V, E), where V is a finite set and E is a binary relation on V, which consists of ordered pairs. Vertices are represented by circles and edges are represented by arrows. Self-loops are possible.

(c) Degree (deg) of a vertex in an ungraph

The degree of a vertex in an undirected graph is the number of edges incident on it.

(d) The in-degree (deg-) and the out-degree (deg+) of a vertex in a digraph

In a directed graph, the out-degree of a vertex is the number of edges leaving it and the in-degree of a vertex is the number of edges entering it.

(e) A simple graph

A directed graph with no self-loops is simple.

(f) A path and simple path in an ungraph and a digraph

A path from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0, u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The path contains the vertices $v_0, v_1, v_2, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.
A path is simple if all vertices in the path are distinct.

(g) A cycle in ungraph and digraph

In a directed graph, a path $\langle v_0, v_1, v_2, \ldots_k \rangle$ forms a cycle if $v_0 = v_k$ and the path contains at least on edge.
In an undirected graph, a path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ forms a cycle if $k > 0$, $v_0 = v_k$, and all edges on the path are distinct; the cycle is simple if $v_0, v_1, v_2, \ldots, v_k$ are distinct.

(h) A (strongly) connected ungraph (digraph) and the (strongly) connected components of a ungraph (digraph)

An undirected graph is connected if every vertex is reachable from all other vertices. The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation.
A directed graph is strongly connected if every two vertices are reachable from each other. The strongly connected components of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation.

(i) A weighted graph

Graphs for which each edge has an associated weight, typically given by a weight function $w : E \to \mathbb{R}$ is called weighted graphs.

(j) A directed acyclic graph (DAG)

A directed graph with no simple cycles is a "directed acyclic graph" and is called a DAG.

1.2 Prove the following

(a)

Proof: Adding the degrees of all the vertices involves counting one for each edge incident on each vertex. How many times does an edge get counted? As it's an ungraph, the edge is not a loop, it is incident on two different vertices and so gets counted twice, once at each vertex, by convention, in the degree of that vertex.
Hence, if $G = (V, E)$ is an ungraph then

$$\sum_{v \in V} \deg v = 2|E|$$

(b)

Proof: Adding the degrees of all the vertices involves counting each edge incident from and to vertex. Obviously, if an edge leaves a vertex, then it enters to a vertex, thus $\sum_{v \in V} \deg^+ v = \sum_{v \in V} \deg^- v$ and $\sum_{v \in V} \deg^+ v + \sum_{v \in V} \deg^- v = \sum_{v \in V} \deg v$.
Hence, using conclusion in (a), if $G = (V, E)$ is an digraph then

$$\sum_{v \in V} \deg^+ v = \sum_{v \in V} \deg^- v = |E|$$

1.3 Prove that any undirected graph has an even number of vertices of odd degree

By conclusion in 1.2.(a), $\sum_{v \in V} \deg v = 2|E|$ is an even number. Since

$$\sum_{v \in V} \deg v = \sum_{v \in V, v \text{ even}} \deg v + \sum_{v \in V, v \text{ odd}} \deg v$$

and the first sum on the right, being a sum of even numbers, is even, so also the second sum

$$\sum_{v \in V, v \text{ odd}} \deg v$$

must be even. Since the sum of an odd number of odd numbers is odd, the number of terms in the sum here, that is, the number of odd vertices, must be even.

1.4 Show that in a simple graph with at least two vertices there must be two vertices that have the same degree

Assume a simple graph G with n vertices, and each vertex with different degree, $n-1, n-2, \ldots, 2, 1, 0$. $\sum_{v \in V} \deg v = (n-1) + (n-2) + \cdots + 2 + 1 + 0 = \frac{n(n-1)}{2}$, which is an odd number. It contradicts with conclusion we drew in 1.2.(a). Our assumption is wrong. Hence, there must be two vertices that have the same degree in a simple graph.

1.5

We know for a complete graph $G = K_n$, each vertex has degree of $n-1$. Use conclusion 1.2.(a). $\sum_{v \in V} \deg v = n(n-1) = 2|E|$, thus $|E| = \frac{n(n-1)}{2}$.

1.6

(a)

vertex	in-degree	out-degree
0	1	2
1	1	2
2	2	1
3	2	1
4	2	4
5	1	2
6	3	0

(b)

vertex	adjacency-list
0	$\rightarrow 2 \rightarrow 4$
1	$\rightarrow 0 \rightarrow 4$
2	$\rightarrow 6$
3	$\rightarrow 1$
4	$\rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$
5	$\rightarrow 3 \rightarrow 6$
6	

(c)

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(d)

$0 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 0$
$0 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 1 \rightarrow 0$
$1 \rightarrow 4 \rightarrow 3$
$1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 1$

(e)

$0 \rightarrow 1 \rightarrow 4 \rightarrow 0$
$0 \rightarrow 2 \rightarrow 4 \rightarrow 0$
$1 \rightarrow 3 \rightarrow 4 \rightarrow 1$
$3 \rightarrow 4 \rightarrow 5 \rightarrow 3$
$4 \rightarrow 5 \rightarrow 6 \rightarrow 4$
$2 \rightarrow 4 \rightarrow 6 \rightarrow 2$
$0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 0$
$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 0$
$0 \rightarrow 2 \rightarrow 6 \rightarrow 4 \rightarrow 0$
$1 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$
$3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 3$
$2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 2$
$0 \rightarrow 1 \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 0$
$0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 0$
$0 \rightarrow 1 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 0$
$0 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 0$
$1 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 1$
$3 \rightarrow 4 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 3$
$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 0$
$0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 0$

$0 \to 1 \to 3 \to 5 \to 6 \to 4 \to 0$
$0 \to 1 \to 4 \to 5 \to 6 \to 2 \to 0$
$0 \to 2 \to 6 \to 5 \to 3 \to 4 \to 0$
$1 \to 3 \to 5 \to 6 \to 2 \to 4 \to 1$
$0 \to 1 \to 3 \to 5 \to 6 \to 2 \to 4 \to 0$
$0 \to 1 \to 4 \to 3 \to 5 \to 6 \to 2 \to 0$
$0 \to 1 \to 3 \to 4 \to 5 \to 6 \to 2 \to 0$
$0 \to 1 \to 3 \to 5 \to 4 \to 6 \to 2 \to 0$
$0 \to 1 \to 3 \to 5 \to 6 \to 4 \to 2 \to 0$
$0 \to 2 \to 6 \to 5 \to 3 \to 1 \to 4 \to 0$
$0 \to 1 \to 3 \to 5 \to 6 \to 2 \to 0$

(f)

No, there is no out-edge for vertex 6.

(g)

$\{0, 1, 3, 4, 5\}$.

1.7

(a)

There are 9 different simple paths.

1. $(3,1) \to (3,2) \to (2,2) \to (2,3) \to (3,3)$
2. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (0,0) \to (0,1) \to (0,2) \to (1,2) \to (1,1) \to (2,1) \to (2,2) \to (2,3) \to (3,3)$
3. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (0,0) \to (0,1) \to (0,2) \to (1,2) \to (1,1) \to (2,1) \to (3,1) \to (3,2) \to (2,2) \to (2,3) \to (3,3)$
4. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (0,0) \to (0,1) \to (0,2) \to (1,2) \to (2,2) \to (2,3) \to (3,3)$
5. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (1,1) \to (0,1) \to (0,2) \to (1,2) \to (1,1) \to (2,1) \to (2,2) \to (2,3) \to (3,3)$
6. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (1,1) \to (0,1) \to (0,2) \to (1,2) \to (1,1) \to (2,1) \to (3,1) \to (3,2) \to (2,2) \to (2,3) \to (3,3)$
7. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (1,1) \to (0,1) \to (0,2) \to (1,2) \to (2,2) \to (2,3) \to (3,3)$
8. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (1,1) \to (2,1) \to (2,2) \to (2,3) \to (3,3)$
9. $(3,1) \to (3,0) \to (2,0) \to (1,0) \to (1,1) \to (2,1) \to (3,1) \to (3,2) \to (2,2) \to (2,3) \to (3,3)$

(b)

There are two strongly connected components.

1. {(2,2), (2,3), (3,2), (3,3)}
2. {(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (3,0), (3,1)}

1.8

Time for computing the out-degree of every vertex is $O(|E|)$, and for in-degree is also $O(|E|)$. Because in both cases it goes through the whole adjacency list whose number is the number of all edges which is $|E|$.

1.9

for $v_i \in V$ in G^T **do**
  $v_i.AdjLst = \emptyset$
end for
for $v_i \in V$ in G **do**
  for u_j in $v_i.AdjLst$ **do**
    $u_j.AdjLst.append(v_i)$ ($u_j \in V$ in G^T)
  end for
end for

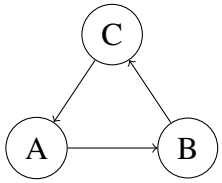
As this algorithm just goes through all edges once, the running time for adjacency list representation is $O(|E|)$.

for $a_{ij} \in G^T.AdjMtrx$ **do**
  $a_{ij} = 0$
end for
for $a_{ij} \in G.AdjMtrx$ **do**
  $b_{ji} = a_{ij}$ ($b_{ji} \in G^T.AdjMtrx$)
end for

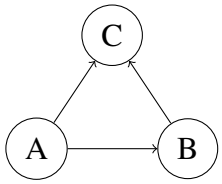
As this algorithm just goes through all elements in matrix whose size is $|V| \times |V|$, the running time for adjacency matrix representation is $O(|V|^2)$.

1.10

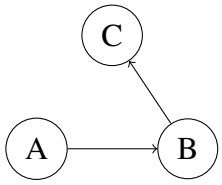
(a)



(b)



(c)



1.11

Given a graph $G = (V, E)$ and a distinguished source vertex s. Store the color of each vertex $u \in V$ in the attribute $u.color$. To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white, gray vertices may have some adjacent white vertices, and all vertices adjacent to black vertices have been discovered.Store the predecessor of u in the attribute $u.\pi$. If u has no predecessor, then $u.\pi =$ NIL. Use attribute $u.d$ to hold the distance from the source s to vertex u and use a queue Q to manage the set of vertices. Breadth-first-search procedure is as following:

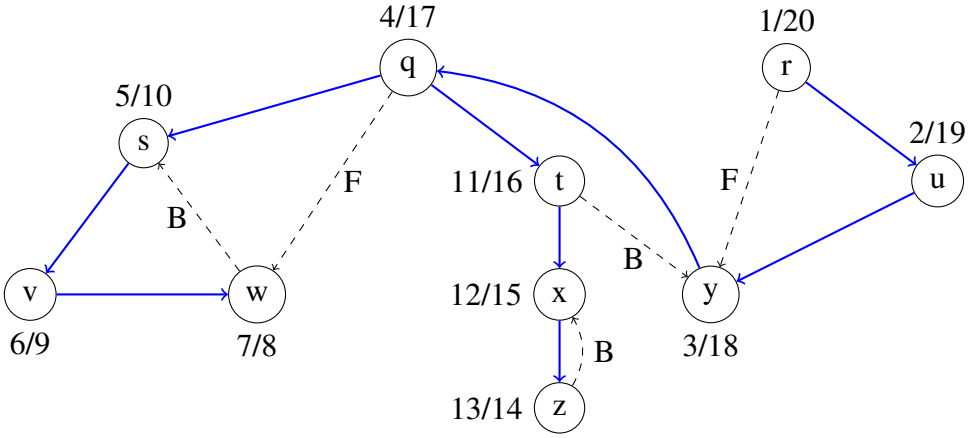
BFS(G, s)
for $u \in G.V - \{s\}$ **do**
  $u.color =$ WHITE
  $u.d = \infty$
  $u.\pi =$ NIL
end for
$s.color =$ GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
$Q.enqueue(s)$
while $Q \neq \emptyset$ **do**
  $u = Q.dequeue()$
  for $v \in G.Adj[u]$ **do**
    if $v.color ==$WHITE **then**
      $v.color =$ GRAY
      $v.d = u.d + 1$
      $v.\pi = u$
      $Q.enqueue(v)$
    end if
  end for
  $u.color =$ BLACK
end while

In depth-first search, we timestamp each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered, and the second timestamp $v.f$ records when the search finishes examining v's adjacency list. The variable $time$ is a global variable for timestamping. Depth-first-search procedure is as following:

DFS(G)
for $u \in G.V$ **do**
  $u.color =$ WHITE
  $u.\pi =$ NIL
end for
$time = 0$
for $u \in G.V$ **do**
  if $u.color ==$ WHITE **then**
    DFS-VISIT(G, u)
  end if
end for

DFS-VISIT(G, u)
$time = time + 1$
$u.d = time$
$u.color =$ GRAY
for $v \in G.Adj[u]$ **do**
  if $v.color ==$ WHITE **then**
    $v.\pi = u$
    DFS-VISIT(G, v)
  end if
end for
$u.color =$ BLACK
$time = time + 1$
$u.f = time$

The results of DFT is as following:



1.12

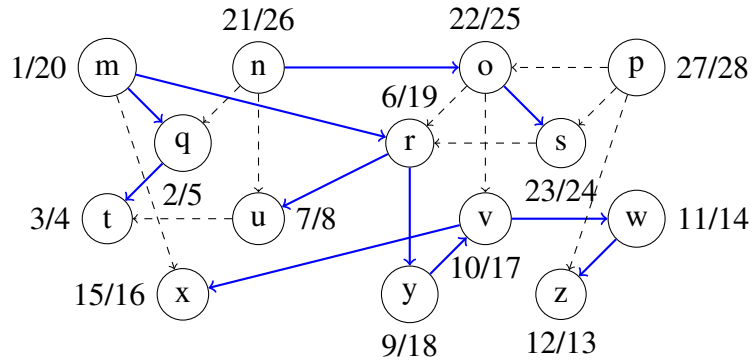
TOPOLOGICAL-SORT(G):

1. call DFS(G) to compute finishing times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list

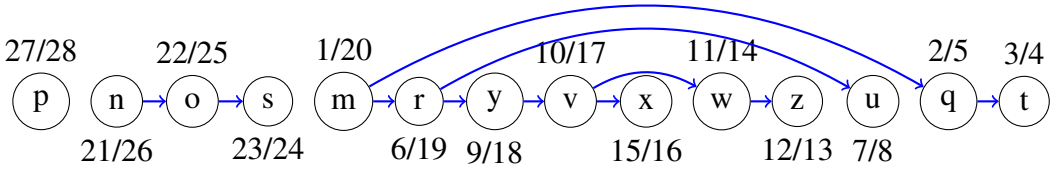
3. return the linked list of vertices

1.13

Apply DFS on the graph:



The topologically sorted graph is shown as:



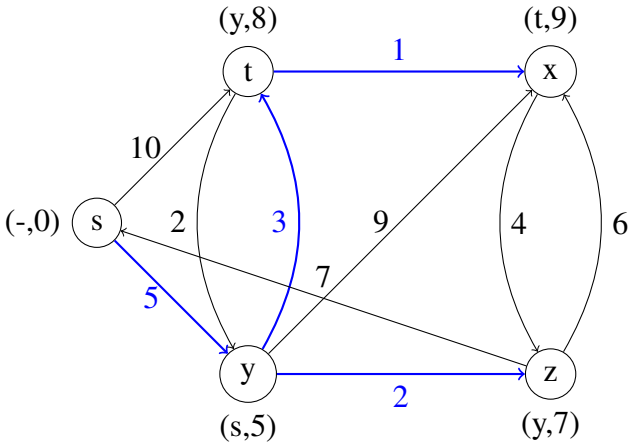
1.14

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. We use a min-priority queue Q of vertices, keyed by their d values, to repeatedly select the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S.

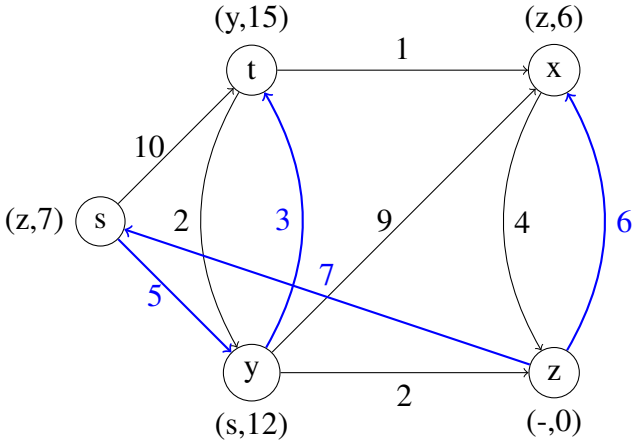
DIJKSTRA(G, w, s)
for $v \in G.V$ **do**
  $v.d = \infty$
  $v.\pi =$ NIL
end for
$s.d = 0$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$ **do**
  $u = Q.extractMin()$
  $S = S \cup \{u\}$
  for $v \in G.Adj[u]$ **do**
    if $v.d > u.d + w(u, v)$ **then**
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
    end if
  end for
end while

Using s as source, let's proceed by progressively assigning to each vertex v in the graph an ordered pair (x, d), where d is the shortest distance from s to v and xv is the last edge on the shortest path,

we can obtain following:



Using vertex z as source, apply same method, we have:



2 Programming Project

2.1 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.

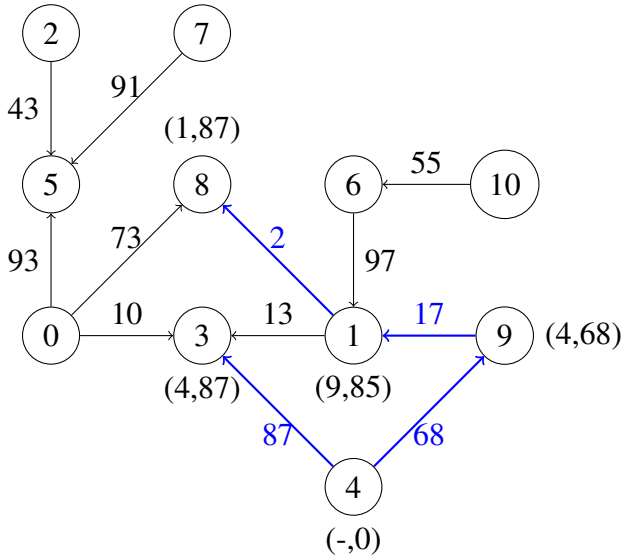
Once the graph has been read, we call $SSAD$ to apply Dijkstra's algorithm for a starting vertex. This algorithm throws a $digraphException$ if there is any error during execution. It catches any $digraphException$ that might be generated and prints an appropriate error message.

2.2 How to run

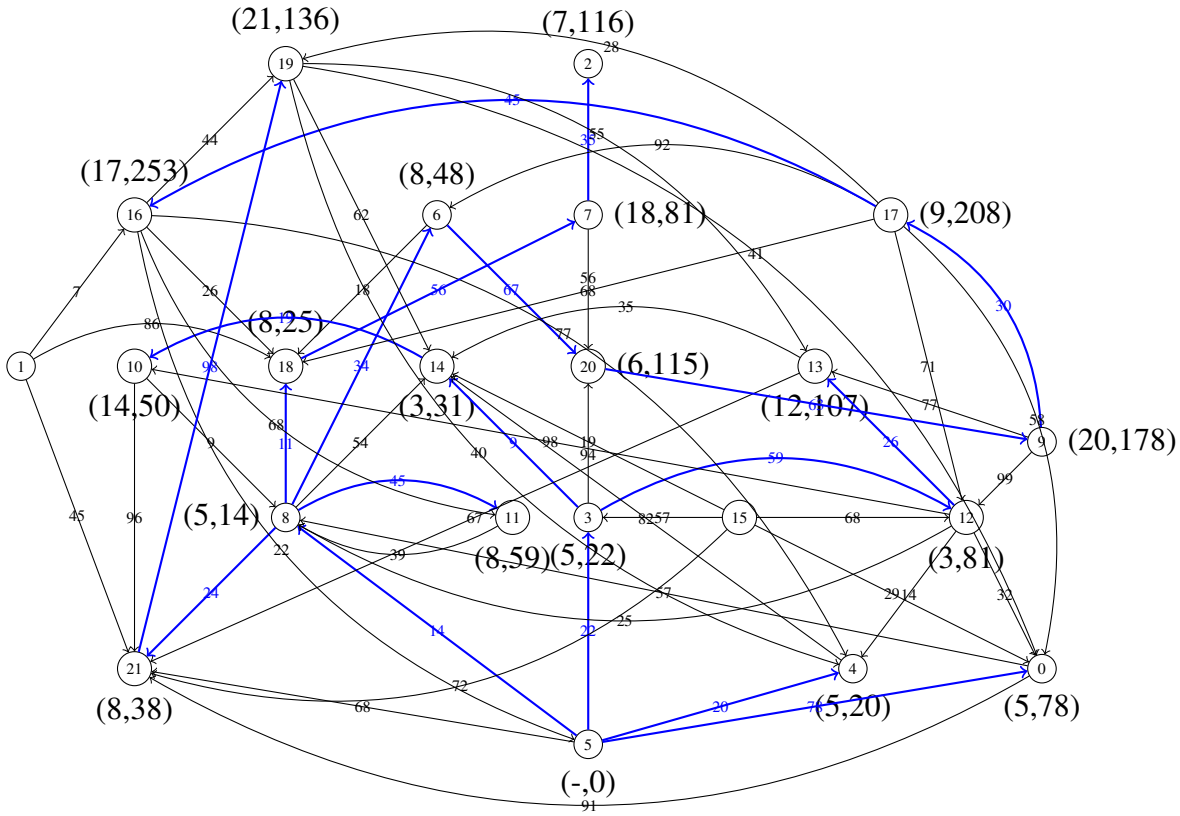
Name your input file as "$File.txt$", then run this program, it will generate text file "$Result.txt$" which contains desired output.

2.3 Input file specification

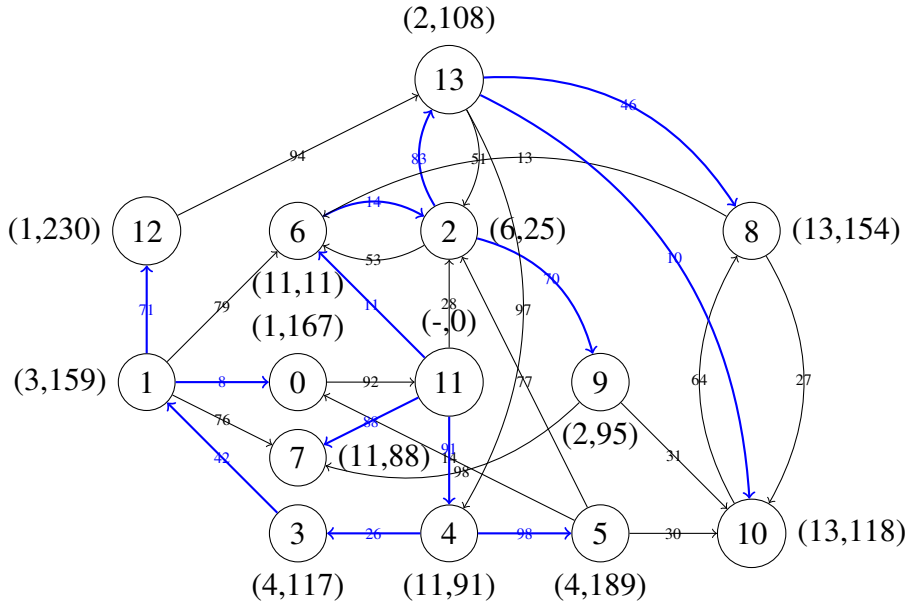
(a) File1.txt



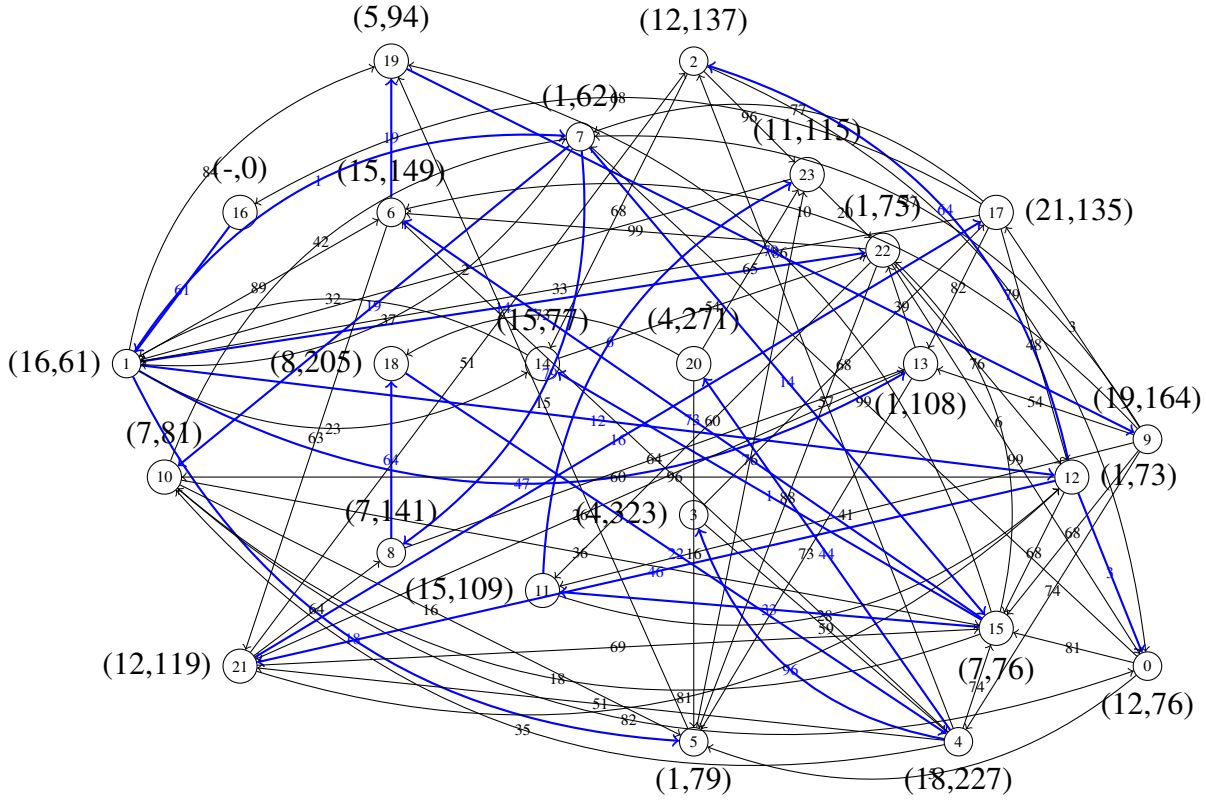
(b) File2.txt



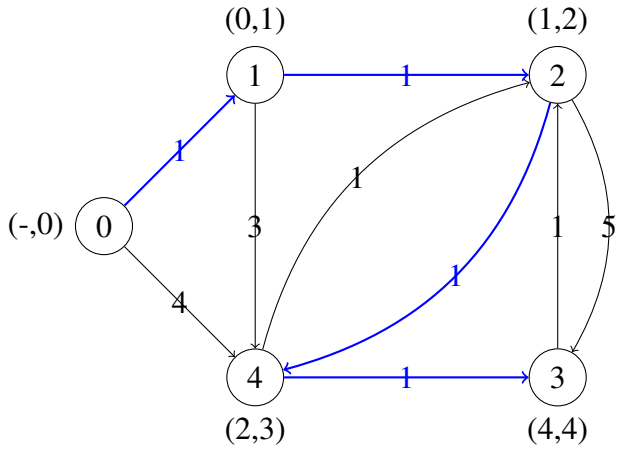
(c) File3.txt



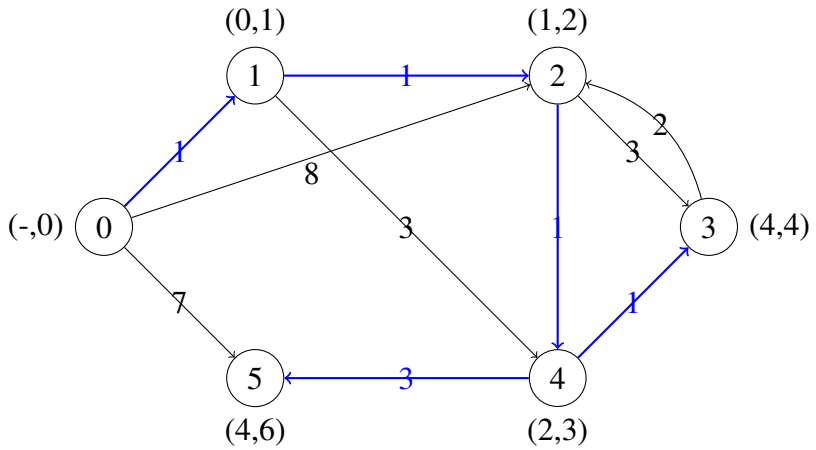
(d) File4.txt



(e) Add1.txt



(f) Add2.txt



(g) Add3.txt