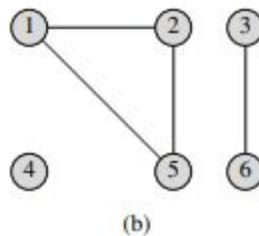


## Answers of the Theoretical Questions

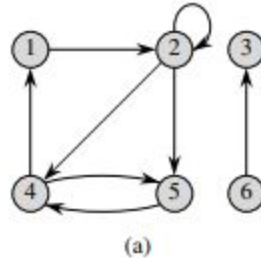
1.

- a. In an undirected graph  $G = (V, E)$ , the edge set  $E$  consists of unordered pairs of vertices, rather than ordered pairs. That is, an edge is a set  $\{u, v\}$ , where  $u, v \in V$  and  $u \neq v$ . By convention, we use the notation  $(u, v)$  for an edge, rather than the set notation  $\{u, v\}$ , and  $(u, v)$  and  $(v, u)$  are considered to be the same edge. In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct vertices. Figure B.2(b) is a pictorial representation of an undirected graph on the vertex set  $\{1, 2, 3, 4, 5, 6\}$ . An undirected graph  $G = (V, E)$ , where  $V = \{1, 2, 3, 4, 5, 6\}$  and  $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$ . The vertex 4 is isolated.



- b. A directed graph (or digraph)  $G$  is a pair  $(V, E)$ , where  $V$  is a finite set and  $E$  is a binary relation on  $V$ . The set  $V$  is called the vertex set of  $G$ , and its elements are called vertices (singular: vertex). The set  $E$  is called the edge set of  $G$ , and its elements are called edges. Figure B.2(a) is a pictorial representation of a directed graph on the vertex set  $\{1, 2, 3, 4, 5, 6\}$ . Vertices are represented by circles in the figure, and edges are represented by arrows. Note that self-loops—edges from a vertex to itself—are possible. A directed graph  $G = (V, E)$ , where  $V = \{1, 2, 3, 4, 5, 6\}$  and  $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$ . The edge  $(2, 2)$  is a self-loop.

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**



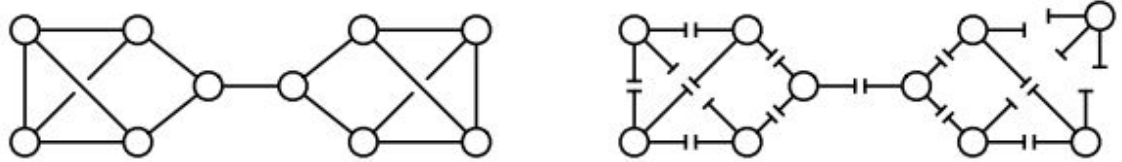
- c. The degree of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure b above has degree 2. A vertex whose degree is 0, such as vertex 4 in Figure b above, is isolated.
- d. In a directed graph, the out-degree of a vertex is the number of edges leaving it, and the in-degree of a vertex is the number of edges entering it. The degree of a vertex in a directed graph is its indegree plus its out-degree. Vertex 2 in Figure B.2(a) has in-degree 2, out-degree 3, and degree 5
- e. In a directed graph, a path  $v_0, v_1, \dots, v_k$  forms a cycle if  $v_0 = v_k$  and the path contains at least one edge. The cycle is simple if, in addition,  $v_1, v_2, \dots, v_k$  are distinct. A self-loop is a cycle of length 1. Two paths  $v_0, v_1, v_2, \dots, v_{k-1}, v_0$  and  $v_0, v_1, v_2, \dots, v_{k-1}, v_0$  form the same cycle if there exists an integer  $j$  such that  $v_i = v_{(i+j) \bmod k}$  for  $i = 0, 1, \dots, k-1$ . In Figure B.2(a), the path 1, 2, 4, 1 forms the same cycle as the paths 2, 4, 1, 2 and 4, 1, 2, 4. This cycle is simple
- f. A path of length  $k$  from a vertex  $u$  to a vertex  $u$  in a graph  $G = (V, E)$  is a sequence  $v_0, v_1, v_2, \dots, v_k$  of vertices such that  $u = v_0, u = v_k$ , and  $(v_{i-1}, v_i) \in E$  for  $i = 1, 2, \dots, k$ . The length of the path is the number of edges in the path. The path contains the vertices  $v_0, v_1, \dots, v_k$  and the edges  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ . (There is always a 0-length path from  $u$  to  $u$ .) If there is a path  $p$  from  $u$  to  $u$ , we say that  $u$  is reachable from  $u$  via  $p$ , which we sometimes write as  $u \sim u$  if  $G$  is directed. A path is simple if all vertices in the path are distinct. In Figure a, the path 1, 2, 5, 4 is a simple path of length 3. The path 2, 5, 4, 5 is not simple.
- g. In a directed graph, a path  $v_0, v_1, \dots, v_k$  forms a cycle if  $v_0 = v_k$  and the path contains at least one edge. The cycle is simple if, in addition,  $v_1, v_2, \dots, v_k$  are distinct. A self-loop is a cycle of length 1. Two paths  $v_0, v_1, v_2, \dots, v_{k-1}, v_0$  and  $v_0, v_1, v_2, \dots, v_{k-1}, v_0$  form the same cycle if there exists an integer  $j$  such that  $v_i = v_{(i+j) \bmod k}$  for  $i = 0, 1, \dots, k-1$ . In Figure B.2(a), the path 1, 2, 4, 1 forms the same cycle as the paths 2, 4, 1, 2 and 4, 1, 2, 4. This cycle is simple

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

$v_0, v_1, v_2, \dots, v_{k-1}, v_0$  form the same cycle if there exists an integer  $j$  such that  $v_i = v_{(i+j) \bmod k}$  for  $i = 0, 1, \dots, k-1$ . In Figure B.2(a), the path 1, 2, 4, 1 forms the same cycle as the paths 2, 4, 1, 2 and 4, 1, 2, 4. This cycle is simple, but the cycle 1, 2, 4, 5, 4, 1 is not. The cycle 2, 2 formed by the edge (2, 2) is a self-loop. A directed graph with no self-loops is simple. In an undirected graph, a path  $v_0, v_1, \dots, v_k$  forms a (simple) cycle if  $k \geq 3$ ,  $v_0 = v_k$ , and  $v_1, v_2, \dots, v_k$  are distinct. For example, in Figure B.2(b), the path 1, 2, 5, 1 is a cycle. A graph with no cycles is acyclic.

- h. 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. A directed graph is strongly connected if it has only one strongly connected component. The graph in Figure a has three strongly connected components:  $\{1, 2, 4, 5\}$ ,  $\{3\}$ , and  $\{6\}$ . All pairs of vertices in  $\{1, 2, 4, 5\}$  are mutually reachable. The vertices  $\{3, 6\}$  do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.
  - i. Weighted graphs for which each edge has an associated weight, typically given by a weight function  $w : E \rightarrow \mathbb{R}$ . For example, let  $G = (V, E)$  be a weighted graph with weight function  $w$ . The weight  $w(u, v)$  of the edge  $(u, v) \in E$  is simply stored with vertex  $v$  in  $u$ 's adjacency list.
  - j. A graph is formed by a collection of vertices and edges, where the vertices are structureless objects that are connected in pairs by edges. In the case of a directed graph, each edge has an orientation, from one vertex to another vertex. A path in a directed graph can be described by a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence; a path forms a cycle if the starting vertex of its first edge equals the ending vertex of its last edge. A directed acyclic graph is a directed graph that has no cycles.
- 2.
- a. Let  $G$  be an arbitrary graph. Split each edge of  $G$  into two ‘half-edges’, each with one endpoint.

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**



Any vertex  $v$  is incident to  $\deg(v)$  half-edges. Thus, the number of half-edges is  $\sum_{v \in V} \deg(v)$ . Every edge was split into exactly two half-edges. Thus, the number of half-edges is also  $2|E|$ .

- b. For any directed graph  $G = (E, V)$ ,  $|E| = \sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v)$ .  
 Discussion When considering directed graphs we differentiate between the number of edges going into a vertex versus the number of edges coming out from the vertex. These numbers are given by the in degree and out degree. Notice that each edge contributes one to the in degree of some vertex and one to the out degree of some vertex. This is essentially the proof of This Theorem.
3. The sum of all the degrees is equal to twice the number of edges. Since the sum of the degrees is even and the sum of the degrees of vertices with even degree is even, the sum of the degrees of vertices with odd degree must be even. If the sum of the degrees of vertices with odd degree is even, there must be an even number of those vertices.
4. First, suppose that  $G$  is a connected finite simple graph with  $n$  vertices. Then every vertex in  $G$  has degree between 1 and  $n - 1$  (the degree of a given vertex cannot be zero since  $G$  is connected, and is at most  $n-1$  since  $G$  is simple). Since there are  $n$  vertices in  $G$  with degree between 1 and  $n - 1$ , the pigeon hole principle lets us conclude that there is some integer  $k$  between 1 and  $n - 1$  such that two or more vertices have degree  $k$ . Now, suppose  $G$  is an arbitrary finite simple graph (not necessarily connected). If  $G$  has any connected component consisting of two or more vertices, the above argument shows that that component contains two vertices with the same degree, and therefore  $G$  does as well. On the other hand, if  $G$  has no connected components with more than one vertex, then every vertex in  $G$  has degree zero, and so there are multiple vertices in  $G$  with the same degree.
5. A complete graph is an undirected graph in which every pair of vertices is adjacent. The number of edges in  $K_N$  is  $N(N - 1) / 2$ .  
 A complete graph means that every vertex is connected with every other vertex. If you take one vertex of your graph, you therefore have  $n-1$  outgoing edges from that particular vertex. Now, you have  $n$  vertices in total, so you might be tempted to say that there are  $n(n-1)$  edges in total,  $n-1$  for every vertex in your graph. But this method counts every edge twice, because every edge going out from one vertex is an edge

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

going into another vertex. Hence, you have to divide your result by 2. This leaves you with  $n(n-1)/2$

6.

a.

Vertex	Indegree	Outdegree
0	1	2
1	1	2
2	2	1
3	2	1
4	2	4
5	1	2
6	3	0

b.

0:	2	4		
1:	0	4		
2:	6			
3:	1			
4:	2	3	5	6
5:	3	6		
6:				

c.

Vertex	0	1	2	3	4	5	6
0	0	0	1	0	1	0	0
1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	1

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

3	0	1	0	0	0	0	0
4	0	0	1	1	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0

- d. 1-4-3-1  
1-4-5-3-1  
1-0-4-3-1  
1-0-4-5-3-1
- e. 0-4-1-0  
0-4-2-0  
2-4-6-2  
5-4-6-5  
3-4-5-3  
1-3-4-1  
1-0-5-3-1  
0-2-6-4-0  
1-4-5-3-1  
2-4-5-6-2  
1-0-2-4-3-1  
2-4-3-5-6-2  
1-0-4-5-3-1  
1-0-2-6-4-1  
1-4-6-5-3-1  
1-4-6-2-0-1  
1-0-2-6-5-4-1
- f. The Graph G is not complete strongly connected components. Only the 0-4-5-3-1-0 is strongly connected but the other half is not strongly connected
- g. 0-4-5-3-1-0  
1-4-3-1  
1-4-5-3-1  
1-0-4-3-1
- 7.
- a. 4  
(3,1)(3,2)(2,2)(2,3)(3,3)  
(3,1)(3,0)(2,0)(1,0)(0,0)(0,1)(0,2)(1,2)(2,2)(2,3)(3,3)

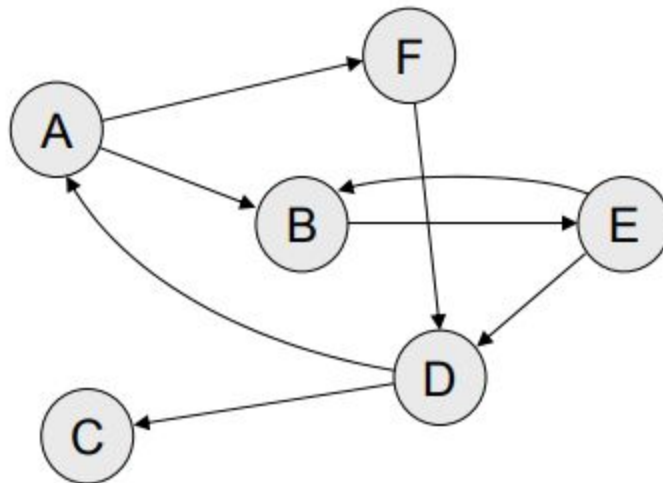
**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

(3,1)(3,0)(2,0)(1,0)(1,1)(2,1)(2,2)(2,3)(3,3)  
 (3,1)(3,0)(2,0)(1,0)(1,1)(0,1)(0,2)(1,2)(2,2)(2,3)(3,3)

b. 6

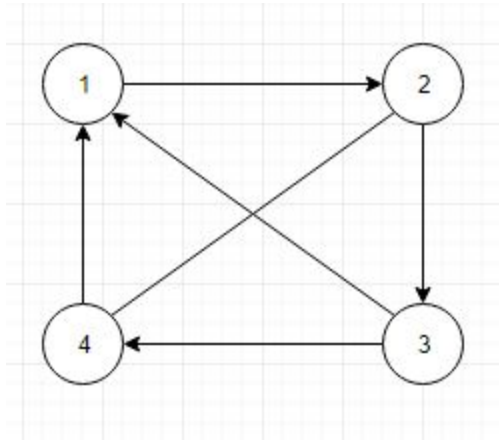
(3,3)(3,2)(2,2)(2,3)(3,3)  
 (2,1)(2,0)(1,0)(1,1)(2,1)  
 (0,2)(1,2)(1,1)(0,1)(0,2)  
 (3,0)(2,0)(1,0)(0,0)(0,1)(0,2)(1,2)(1,1)(2,1)(3,1)(3,0)  
 (2,0)(1,0)(0,0)(0,1)(0,2)(1,2)(1,1)(2,1)(2,0)  
 (3,0)(2,0)(1,0)(1,1)(2,1)(3,1)(3,0)

8. Since it seems as though the list for the neighbors of each vertex  $v$  is just an undecorated list, to find the length of each would take time  $O(\text{out-degree}(v))$ . So, the total cost will be  $\sum_{v \in V} O(\text{out-degree}(v)) = O(|E| + |V|)$ . Note that the  $|V|$  showing up in the asymptotics is necessary, because it still takes a constant amount of time to know that a list is empty. This time could be reduced to  $O(|V|)$  if for each list in the adjacency list representation, we just also stored its length. To compute the in degree of each vertex, we will have to scan through all of the adjacency lists and keep counters for how many times each vertex has appeared. As in the previous case, the time to scan through all of the adjacency lists takes time  $O(|E| + |V|)$ .
9. For the adjacency matrix representation, to compute the graph transpose, we just take the matrix transpose. This means looking along every entry above the diagonal, and swapping it with the entry that occurs below the diagonal. This takes time  $O(|V|^2)$ . For the adjacency list representation, we will maintain an initially empty adjacency list representation of the transpose. Then, we scan through every list in the original graph. If we are in the list corresponding to vertex  $v$  and see  $u$  as an entry in the list, then we add an entry of  $v$  to the list in the transpose graph corresponding to vertex  $u$ . Since this only requires a scan through all of the lists, it only takes time  $O(|E| + |V|)$ .
10.
  - a. Digraphs with at least one cycle

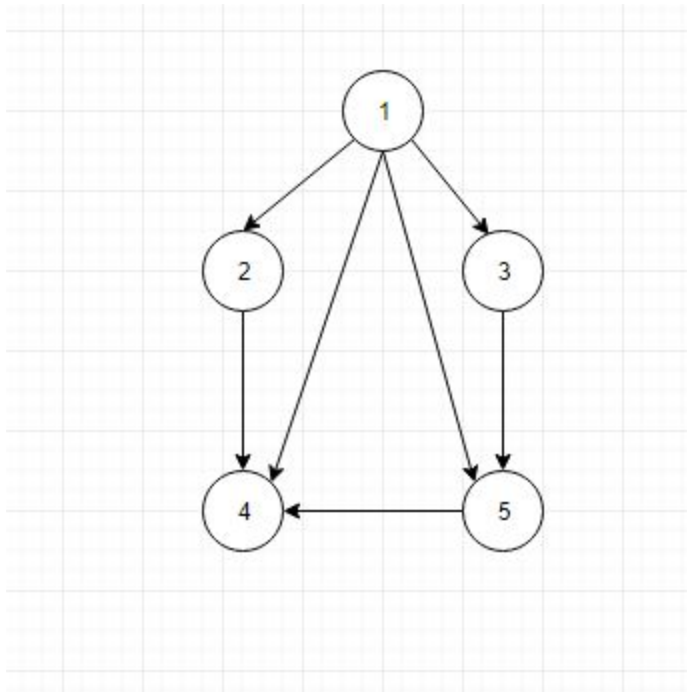


**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

b.



c.



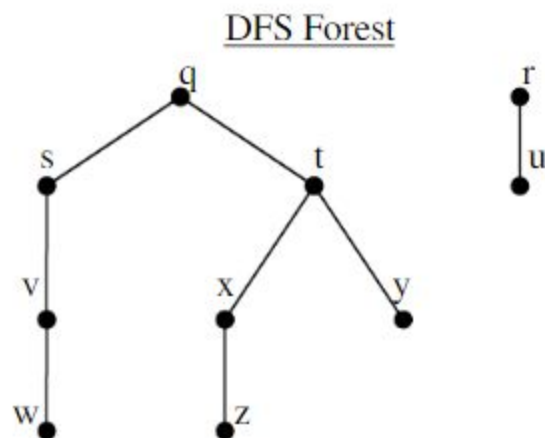
11.

Vertex	adj	Discover	Finish
q	S t w	1	16
r	U v	17	20



**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

s	v	2	7
t	X y	8	15
u	y	18	19
v	w	3	6
w	s	4	5
x	z	9	12
y	q	13	14
z	x	10	11



Edge	Classification
(q,s)	tree
(q,t)	tree
(q,w)	forward
(r,u)	tree
(r,y)	cross
(s,v)	tree

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

(t,x)	tree
(t,y)	tree
(u,y)	cross
(v,w)	tree
(w,s)	back
(x,z)	tree
(y,q)	back
(z,x)	back

12. topological sort using DFS's pseudocode is given below:

First step is to create a graph by using addEdge(a,b), then use topologicalSort(). TopologicalSort has 3 stages as first stage create a stack and a boolean array named as visited[]; then stage two is to Mark all the rest of the vertices as not visited i.e. initialize visited[] with 'false' value. As stage three apply the recursive helper function topologicalSortUtil() to store Topological Sort starting from all vertices one by one.

As Step three is def topologicalSortUtil(int v, bool visited[], stack<int> & Stack): but it has 3 stages as well stage one-Mark the current node as visited

Stage 2-Recur for all the vertices adjacent to this vertex.

Stage 3-Push current vertex to stack which stores result.

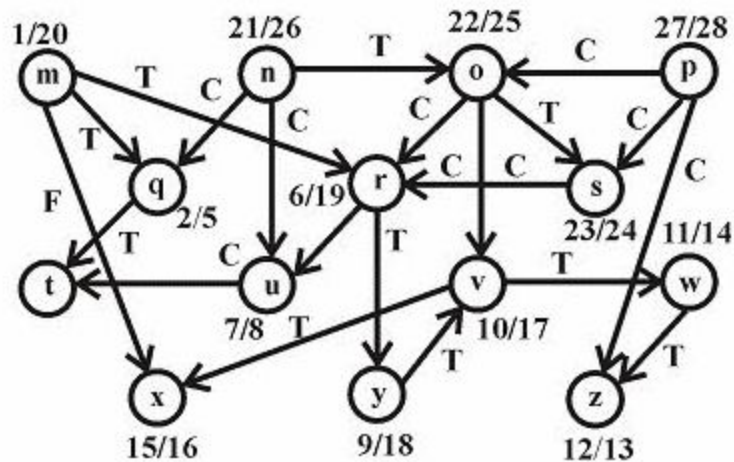
The final main step is to return from the utility function, print the contents of stack.

13.

label	d	f
m	1	20
q	2	5
t	3	4
r	6	19
u	7	8
y	9	18
v	10	17

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

w	11	14
z	12	13
x	15	16
n	21	26
o	22	25
s	23	24
p	27	28



And so, by reading off the entries in decreasing order of finish time, we have the sequence p, n, o, s, m, r, y, v, x, w, z, u, q, t.

14. Dijkstra's algorithm is the algorithm used to find the single-source shortest path from a source vertex to all other vertices in the graph.

From vertex S as resource

- The shortest path from vertex S to vertex T is 3
- The shortest path from vertex S to vertex X is vertex S to vertex T is 3 + vertex T to X is 6 so the total is 9
- The shortest path from vertex S to vertex Y is 5
- The shortest path from vertex S to vertex Z is vertex S to vertex Y is 5 + vertex Y to X is 6 so the total is 11

From vertex Z as resource

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

- The shortest path from vertex Z to vertex Y is vertex Z to vertex S is 3 + vertex S to Y is 5 so the total is 8
- The shortest path from vertex Z to vertex X is 7
- The shortest path from vertex Z to vertex T is vertex Z to vertex S is 3 + vertex S to T is 3 so the total is 6
- The shortest path from vertex Z to vertex S is 3

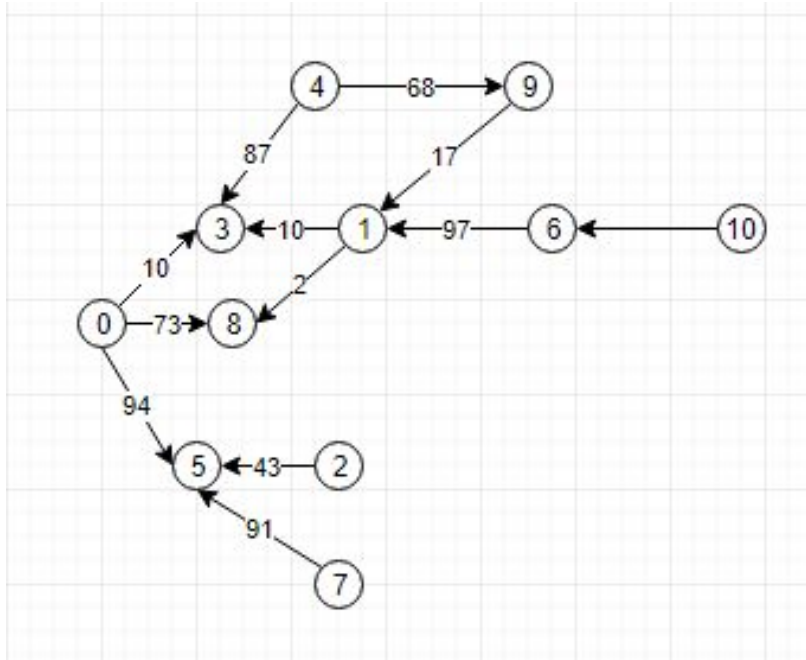
## Summary of the Implemented project

The Project has been completely implemented using C++ programming Language and VisualBasic IDE. As a beginning the console will throw some details and ask the question to the user about “which file do you want to run?” user can select (File1,File2,File3,File4) one of the four files. Also user doesn't have the problem with Capital or small letters. As soon as user entered the name program will start to analyse the specific file inside the “data” folder. The specified data folder is inside the project file which is holding 4 files and 4 result files(.txt) along with one sample.txt file. If the user selects file1.txt the results will be storing inside the Result1.txt. Also it will print in console also for more clarification. After opening the selected file program will retrieve the information about the total vertex and start vertex the create a 1 Dimensional vector to store the Input details of the Matrix. Then the program will create another 2 Dimensional vector and store each 1 D vector inside it. Then program will analyse the details about the vector and display the details as a first step based on the requirement.

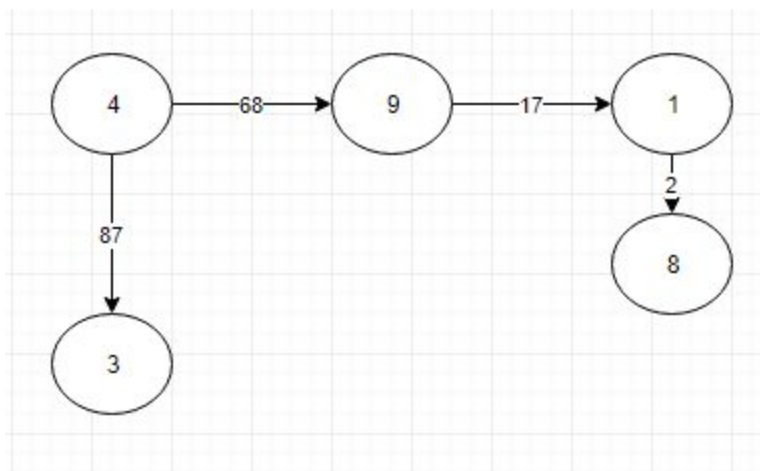
During the analyzing the data line by line File1.txt doesn't have much problem but the rest of the files having unwanted space. so the Line2 of file1 is holding the information of the start vertex but the rest input files Line3 is holding the start vertex information. So based on that i have to manage those things to avoid unwanted problems. As a second part program will pass call another class called digraph which is holding the dijkstra algorithm. The function will create an instance of the 3 arrays for distance, parent and boolean array of the vertex with the same size of total vector also those arrays has been set to default value and the int distance[] array has been set to INT\_MAX/INFINITE value and the minimum distance of the same start vertex has been set to 0. As a next step the program will find the shortest path for all vertices. While doing that another function will find out the minimum distance of all vertex and the same vertex will set to true in the boolean array. Right after that it will update distance[] not in vertex, there is an edge from u to v, and total weight of path from src to v through u is smaller than the current value of distance[i] Then the program will print the value and the path.

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

File1.txt

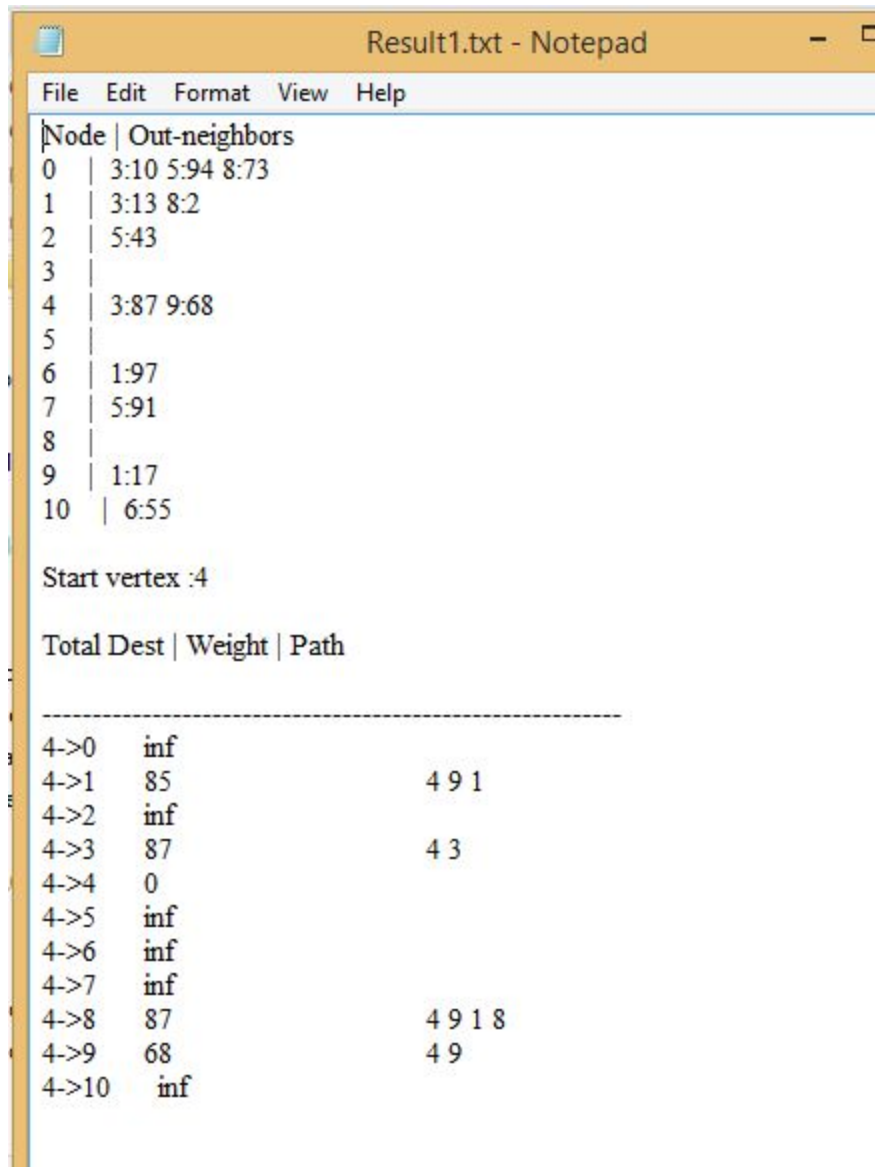


Graph of the file1.txt



The image above is the shortest path of the previous graph without INF vertices found by the Dijkstra's SSAD algorithm of File1.txt input

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**



Result1.txt - Notepad

Node	Out-neighbors
0	3:10 5:94 8:73
1	3:13 8:2
2	5:43
3	
4	3:87 9:68
5	
6	1:97
7	5:91
8	
9	1:17
10	6:55

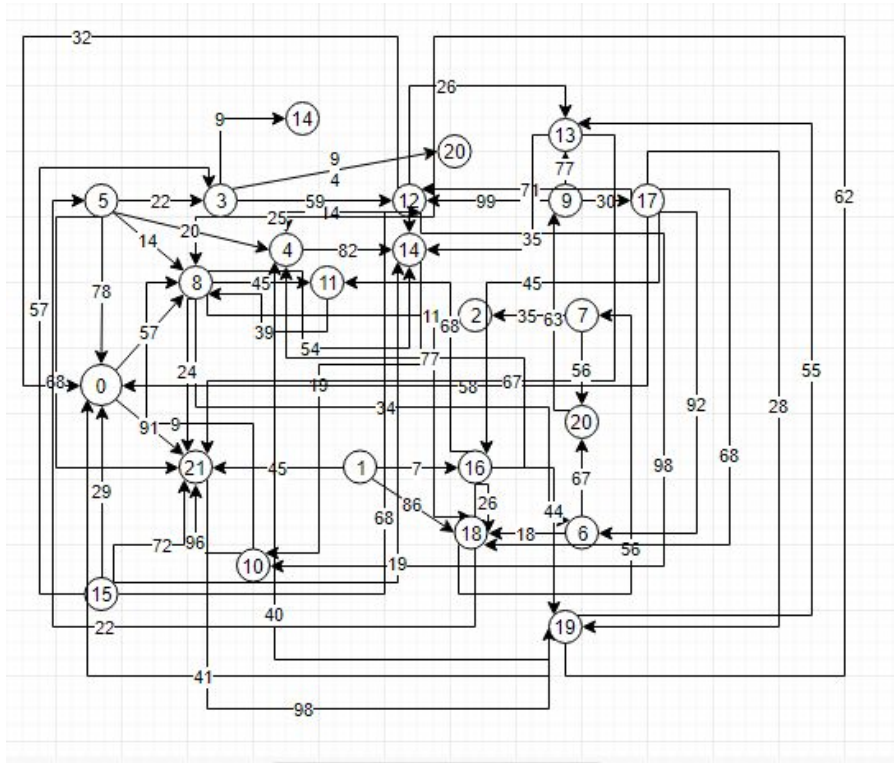
Start vertex :4

Total Dest	Weight	Path
4->0	inf	
4->1	85	4 9 1
4->2	inf	
4->3	87	4 3
4->4	0	
4->5	inf	
4->6	inf	
4->7	inf	
4->8	87	4 9 1 8
4->9	68	4 9
4->10	inf	

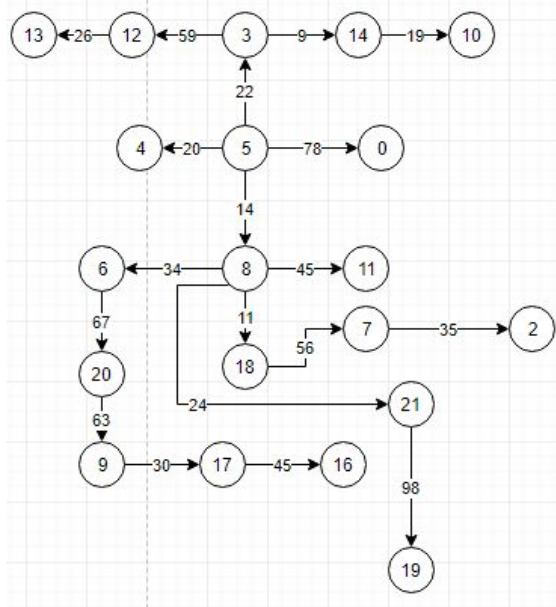
This image representing the shortest path found by the Dijkstra's SSAD algorithm & Screenshot of the Output(Result1.txt) for the file1.txt

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

File2.txt



Above image is indicating the Graph of the file2.txt



The image above is the shortest path of the previous graph without INF vertices found by the Dijkstra's SSAD algorithm of File2.txt input

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

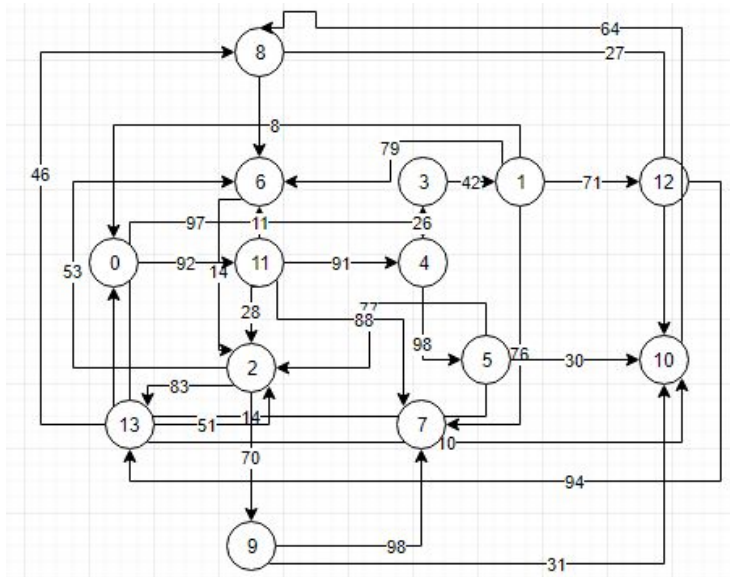
File Edit Format View Help		Start vertex :5		
Node	Out-neighbors	Total	Dest	Weight   Path
Start vertex: 5				
0	8:57 21:91	5->0	78	5 0
1	16:7 18:86 21:45	5->1	inf	
2		5->2	116	5 8 18 7 2
3	12:59 14:9 20:94	5->3	22	5 3
4	14:82	5->4	20	5 4
5	0:78 3:22 4:20 8:14 21:68	5->5	0	
6	18:18 20:67	5->6	48	5 8 6
7	2:35 20:56	5->7	81	5 8 18 7
8	6:34 11:45 14:54 18:11 21:24	5->8	14	5 8
9	12:99 13:77 17:30	5->9	178	5 8 6 20 9
10	8:9 21:96	5->10	50	5 3 14 10
11	8:39	5->11	59	5 8 11
12	0:32 4:14 8:25 10:98 13:26	5->12	81	5 3 12
13	14:35 21:67	5->13	107	5 3 12 13
14	10:19	5->14	31	5 3 14
15	0:29 3:57 12:68 14:19 21:72	5->15	inf	
16	4:77 5:22 11:68 18:26 19:44	5->16	253	5 8 6 20 9 17 16
17	0:58 6:92 12:71 16:45 18:68 19:28	5->17	208	5 8 6 20 9 17
18	7:56	5->18	25	5 8 18
19	0:41 4:40 13:55 14:62	5->19	136	5 8 21 19
20	9:63	5->20	115	5 8 6 20
21	19:98	5->21	38	5 8 21
Start vertex :5				

This image above representing the shortest path found by the Dijkstra's SSAD algorithm & Screenshot of the Output(Result2.txt) for the file2.txt

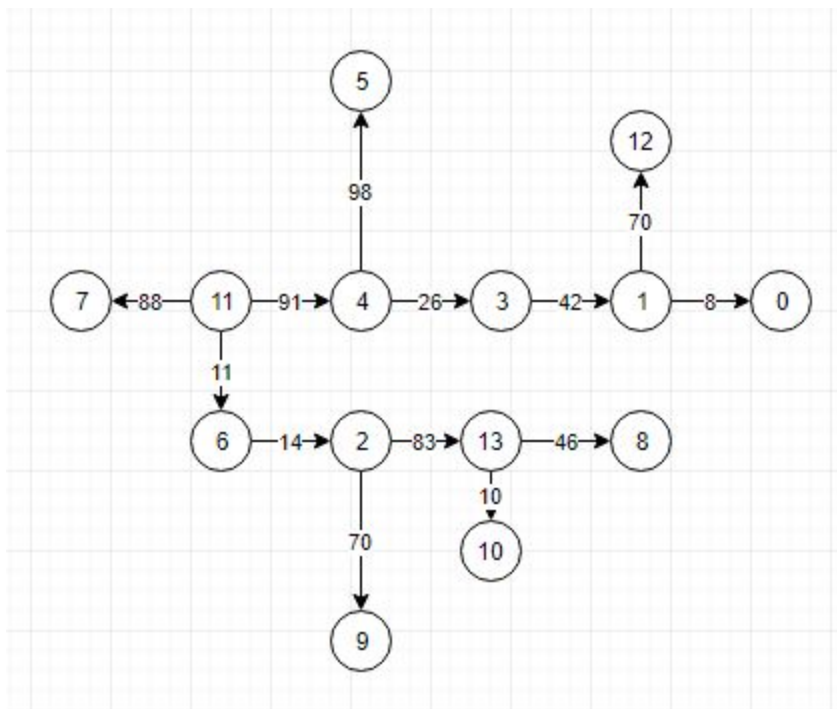


**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

File3.txt



This image above representing the graph of the File3.txt



The image above is the shortest path of the previous graph without INF vertices found by the Dijkstra's SSAD algorithm of File3.txt input

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

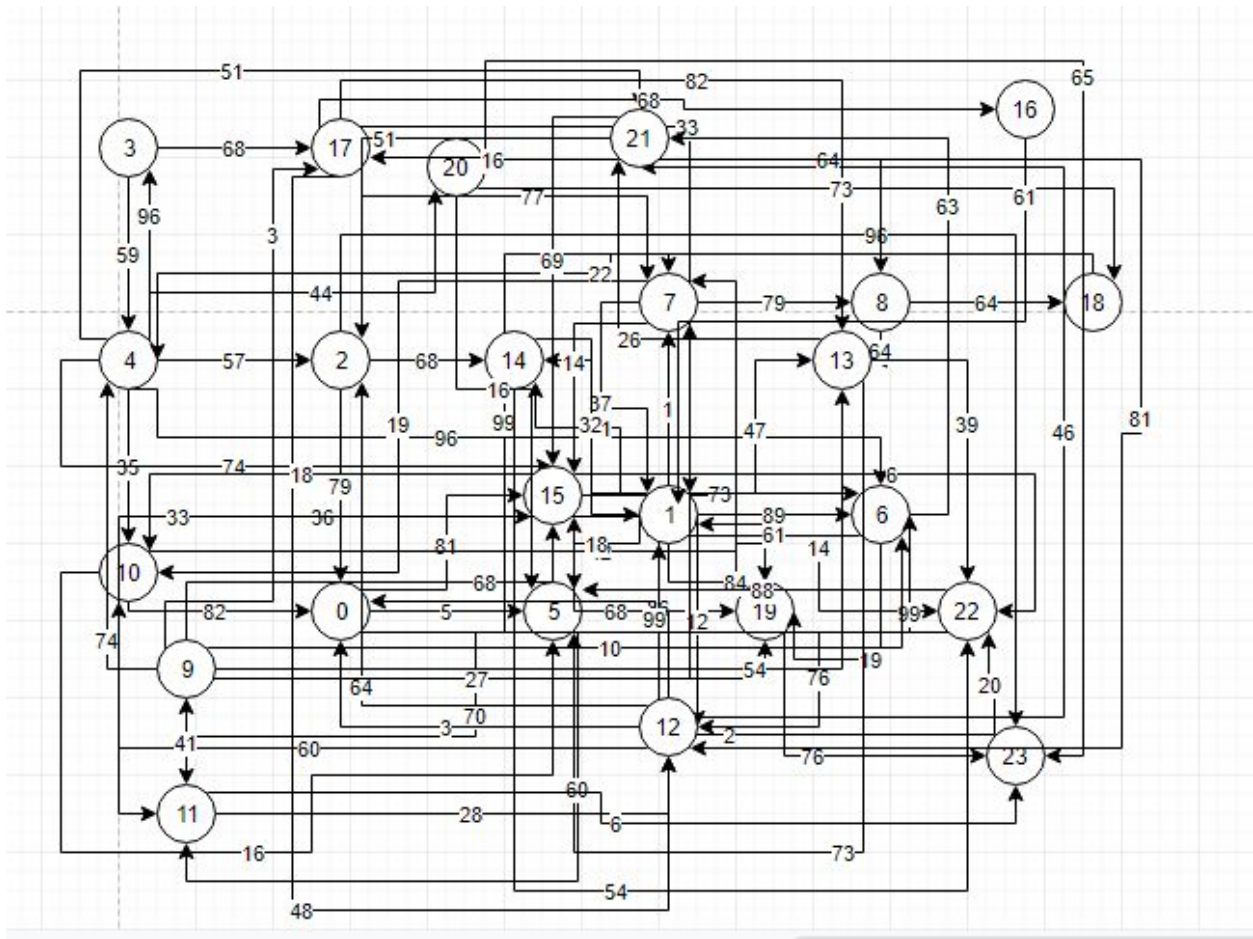
```
Result3.txt - Notepad
File Edit Format View Help
Node | Out-neighbors
Start vertex: 11
0 | 11:92
1 | 0:8 6:79 7:76 12:71
2 | 6:53 9:70 13:83
3 | 1:42
4 | 3:26 5:98
5 | 0:14 2:77 10:30
6 | 2:14
7 |
8 | 6:13 10:27
9 | 7:98 10:31
10 | 8:64
11 | 2:28 4:91 6:11 7:88
12 | 13:94
13 | 2:51 4:97 8:46 10:10
|
Start vertex :11

Total Dest | Weight | Path
-----
11->0 167 11 4 3 1 0
11->1 159 11 4 3 1
11->2 25 11 6 2
11->3 117 11 4 3
11->4 91 11 4
11->5 189 11 4 5
11->6 11 11 6
11->7 88 11 7
11->8 154 11 6 2 13 8
11->9 95 11 6 2 9
11->10 118 11 6 2 13 10
11->11 0 11 4 3 1 12
11->12 230 11 4 3 1 12
11->13 108 11 6 2 13
```

This image representing the shortest path found by the Dijkstra's SSAD algorithm & Screenshot of the Output(Result3.txt) for the file3.txt

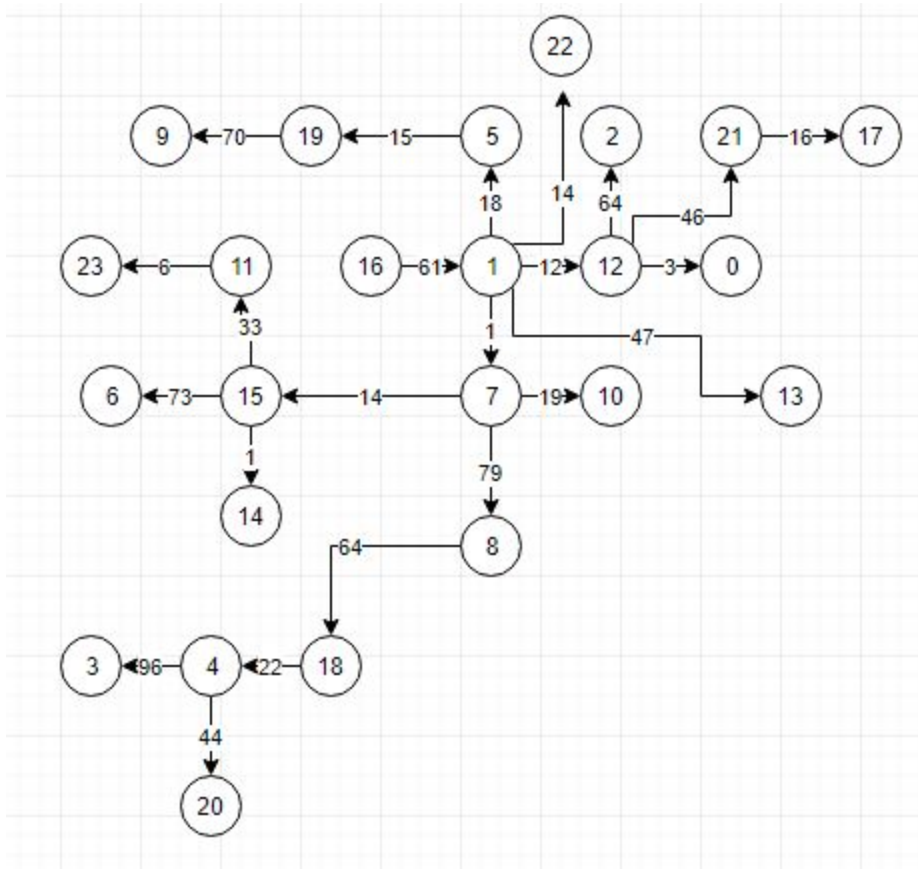
**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

File4.txt



This image above is representing the graph of the File4.txt

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**



The image above is the shortest path of the previous graph without INF vertices found by the Dijkstra's SSAD algorithm of File4.txt input

**SummerProject-2019**  
**GraphTheory: Representation and Algorithms**  
**Pirashanth Satkunabalasingam 002244891**

Result4.txt		Start vertex :16		
File Edit Format View Help		Total Dest   Weight   Path		
Node	Out-neighbors			
Start vertex: 16				
0	5:5 7:99 15:81	16->0	76	16 1 12 0
1	5:18 6:89 7:1 12:12 13:47 14:23 19:84 22:14	16->1	61	16 1
2	0:79 14:68 23:96	16->2	137	16 1 12 2
3	4:59 17:68	16->3	323	16 1 7 8 18 4 3
4	2:57 3:96 6:96 10:35 15:74 20:44 21:51	16->4	227	16 1 7 8 18 4
5	19:15 23:76	16->5	79	16 1 5
6	19:19 21:63	16->6	149	16 1 7 15 6
7	1:37 8:79 10:19 15:14	16->7	62	16 1 7
8	13:64 18:64	16->8	141	16 1 7 8
9	4:74 6:10 7:27 11:41 13:54 15:68 17:3	16->9	164	16 1 5 19 9
10	0:82 5:16 7:42 15:36	16->10	81	16 1 7 10
11	12:28 23:6	16->11	109	16 1 7 15 11
12	0:3 2:64 10:60 15:68 21:46	16->12	73	16 1 12
13	5:73 21:26 22:39	16->13	108	16 1 13
14	1:32 22:54	16->14	77	16 1 7 15 14
15	6:73 10:18 11:33 14:1 19:86 22:6	16->15	76	16 1 7 15
16	1:61	16->16	0	
17	1:33 7:77 12:48 13:82 16:68	16->17	135	16 1 12 21 17
18	4:22	16->18	205	16 1 7 8 18
19	9:70	16->19	94	16 1 5 19
20	5:16 18:73 23:65	16->20	271	16 1 7 8 18 4 20
21	2:51 8:64 12:81 15:69 17:16	16->21	119	16 1 12 21
22	0:99 5:88 6:99 11:60 12:76	16->22	75	16 1 22
23	1:2 22:20	16->23	115	16 1 7 15 11 23

This image representing the shortest path found by the Dijkstra's SSAD algorithm & Screenshot of the Output(Result4.txt) for the file4.txt