Project 2: Graph Algorithms & Related Data Structures

ITCS 6114 – Algorithms and Data Structures

By Ria Banerjee & TJ Bah

Problem 1: Single-source Shortest Path Algorithm

Find the shortest path tree in both directed and undirected weighted graphs for a given source vertex. (Assume there are no negative edges in the graph). Print each path and path cost for a given source.

We can utilize Dijkstra's Algorithm for Both Directed & Undirected Graphs (Using a Priority Queue). The Algorithm is only concerned with the Vertices and Costs. Nodes are added into the Priority Queue only if there is an edge to travel to from the Adjacency List. Ex. Point A to B in a directed graph adds a single edge, while an Undirected graph adds two edges. Using a binary heap for our Priority Queue gives us an overall running time of O(m log n). Data Structures used: Hash Set, Priority Queue and List data structures were utilized.

Pseudocode

Line # -
- - Subline # (From Call) -

0 - Algorithm dijkstras_algo(G, w, s)

1 - Initialize-Single-Source(G, s) ---------- --> O(n)

- - 1 - **for** each vertex v ∈ G.V

- - 2 - v.d = ∞

- - 3 - v.π = NIL

- - 4 - s.d = 0 // Distance to the source from itself is 0

2 - S = 0

3 - Q = G.V ---------- --> O(n log n)

4 - **while** Q != 0

// u is removed from Priority Queue and has min distance

5 - u = Extract-Min(Q) --> O(log n) (Called n times) = O(n log n)

6 - S = S U {u} // add node to finalized list (visited)

7 - **for** each vertex v ∈ G.Adj[u]

8 - Relax(u, v, w) ---------- --> O(log n) (Called m times) = O(m log n)

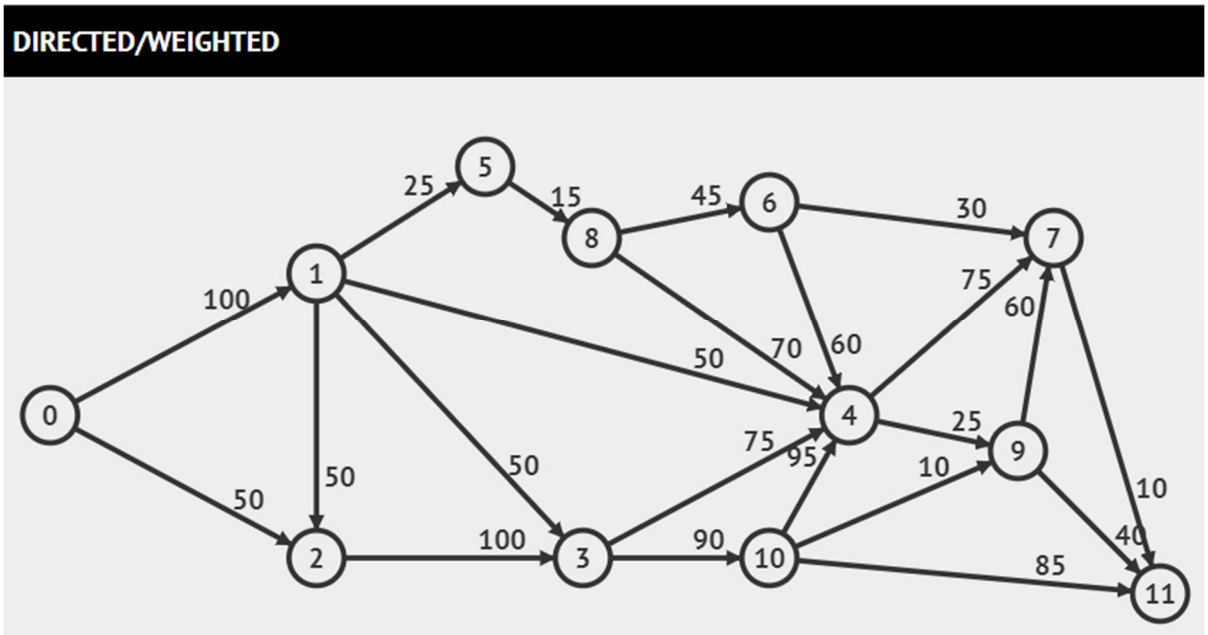
- - 1 - if v.d > u.d + w(u,v)

- - 2 - v.d = u.d + w(u,v)

- - 3 - v.π = u

Total Running time = O(m log n)

Test Case 1 – Directed / Weighted



• **V=12, E=22** • DAG? **Yes**

Edge List

0:	0	1	100
1:	0	2	50
2:	1	2	50
3:	1	3	50
4:	1	4	50
5:	1	5	25
6:	2	3	100
7:	3	4	75
8:	3	10	90
9:	4	7	75
10:	4	9	25
11:	5	8	15
12:	6	4	60
13:	6	7	30
14:	7	11	10
15:	8	4	70
16:	8	6	45
17:	9	7	60
18:	9	11	40
19:	10	4	95
20:	10	9	10
21:	10	11	85

Adjacency List

0:	(1, 100)	(2, 50)		
1:	(2, 50)	(3, 50)	(4, 50)	(5, 25)
2:	(3, 100)			
3:	(4, 75)	(10, 90)		
4:	(7, 75)	(9, 25)		
5:	(8, 15)			
6:	(4, 60)	(7, 30)		
7:	(11, 10)			
8:	(4, 70)	(6, 45)		
9:	(7, 60)	(11, 40)		
10:	(4, 95)	(9, 10)	(11, 85)	
11:				

Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	100	50	0	0	0	0	0	0	0	0	0
1	0	0	50	50	50	25	0	0	0	0	0	0
2	0	0	0	100	0	0	0	0	0	0	0	0
3	0	0	0	0	75	0	0	0	0	0	90	0
4	0	0	0	0	0	0	0	75	0	25	0	0
5	0	0	0	0	0	0	0	0	15	0	0	0
6	0	0	0	0	60	0	0	30	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	10
8	0	0	0	0	70	0	45	0	0	0	0	0
9	0	0	0	0	0	0	0	60	0	0	0	40
10	0	0	0	0	95	0	0	0	0	10	0	85
11	0	0	0	0	0	0	0	0	0	0	0	0

Input

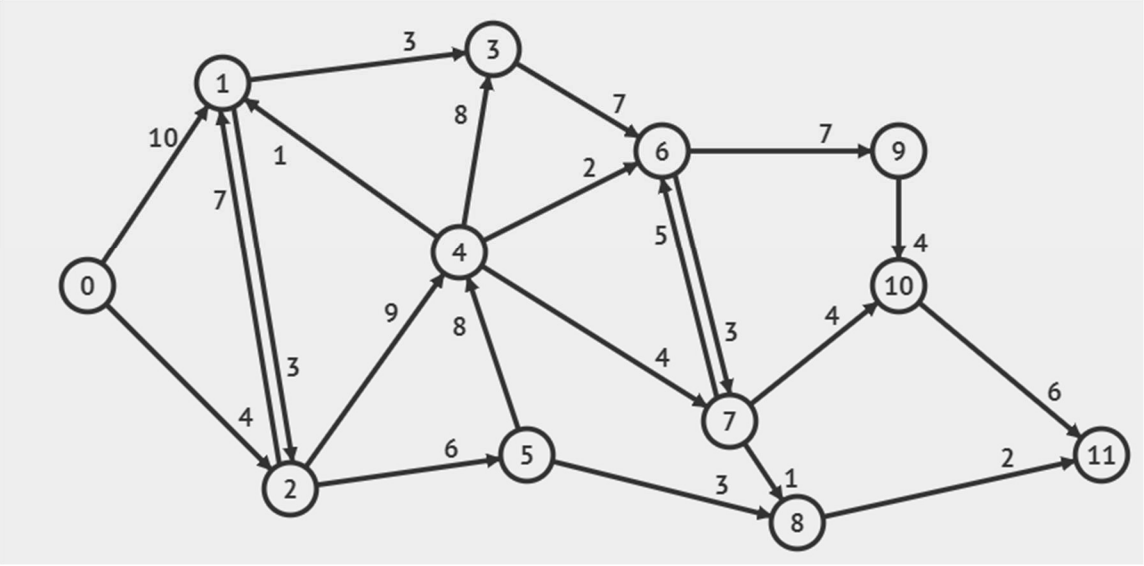
```
// to travel from vertex [X = src] to [Y = dest], one has to cover [Z = cost] units of distance
// adj_list.get(X).add(new Node(Y, Z));

// Test Case / Graph 1 (Directed)
adj_list.get(0).add(new Node( vertex: 1,  cost: 100));
adj_list.get(0).add(new Node( vertex: 2,  cost: 50));

adj_list.get(1).add(new Node( vertex: 2,  cost: 50));
adj_list.get(1).add(new Node( vertex: 3,  cost: 50));
adj_list.get(1).add(new Node( vertex: 4,  cost: 50));
adj_list.get(1).add(new Node( vertex: 5,  cost: 25));

adj_list.get(2).add(new Node( vertex: 3,  cost: 100));

adj_list.get(3).add(new Node( vertex: 4,  cost: 75));
adj_list.get(3).add(new Node( vertex: 10,  cost: 90));

adj_list.get(4).add(new Node( vertex: 7,  cost: 75));
adj_list.get(4).add(new Node( vertex: 9,  cost: 25));

adj_list.get(5).add(new Node( vertex: 8,  cost: 15));

adj_list.get(6).add(new Node( vertex: 4,  cost: 60));
adj_list.get(6).add(new Node( vertex: 7,  cost: 30));

adj_list.get(7).add(new Node( vertex: 11,  cost: 10));

adj_list.get(8).add(new Node( vertex: 4,  cost: 70));
adj_list.get(8).add(new Node( vertex: 6,  cost: 45));

adj_list.get(9).add(new Node( vertex: 7,  cost: 60));
adj_list.get(9).add(new Node( vertex: 11,  cost: 40));

adj_list.get(10).add(new Node( vertex: 4,  cost: 95));
adj_list.get(10).add(new Node( vertex: 9,  cost: 10));
adj_list.get(10).add(new Node( vertex: 11,  cost: 85));
```

Output

```
Path (0 → 1): Path Cost = 100, Path Route = [0, 1]
Path (0 → 2): Path Cost = 50, Path Route = [0, 2]
Path (0 → 3): Path Cost = 150, Path Route = [0, 2, 3]
Path (0 → 4): Path Cost = 150, Path Route = [0, 1, 4]
Path (0 → 5): Path Cost = 125, Path Route = [0, 1, 5]
Path (0 → 6): Path Cost = 185, Path Route = [0, 1, 5, 8, 6]
Path (0 → 7): Path Cost = 215, Path Route = [0, 1, 5, 8, 6, 7]
Path (0 → 8): Path Cost = 140, Path Route = [0, 1, 5, 8]
Path (0 → 9): Path Cost = 175, Path Route = [0, 1, 4, 9]
Path (0 → 10): Path Cost = 240, Path Route = [0, 2, 3, 10]
Path (0 → 11): Path Cost = 215, Path Route = [0, 1, 4, 9, 11]
--------------------------------------------------------------------------------
The shortest path from the source node to other nodes:
[Source Node] →        [Node #]        Path (Cost):        Parent Node:
    0                       0               0                   -1
    0                       1               100                     0
    0                       2               50                      0
    0                       3               150                     2
    0                       4               150                     1
    0                       5               125                     1
    0                       6               185                     8
    0                       7               215                     6
    0                       8               140                     5
    0                       9               175                     4
    0                       10                  240                     3
    0                       11                  215                     9
```

Test Case 2 – Directed / Weighted

DIRECTED/WEIGHTED



• **V=12, E=22** • DAG? **No**

Edge List

0:	0	1	10
1:	0	2	4
2:	1	2	3
3:	1	3	3
4:	2	1	7
5:	2	4	9
6:	2	5	6
7:	3	6	7
8:	4	1	1
9:	4	3	8
10:	4	6	2
11:	4	7	4
12:	5	4	8
13:	5	8	3
14:	6	7	3
15:	6	9	7
16:	7	6	5
17:	7	8	1
18:	7	10	4
19:	8	11	2
20:	9	10	4
21:	10	11	6

Adjacency List

0:	(1, 10)	(2, 4)		
1:	(2, 3)	(3, 3)		
2:	(1, 7)	(4, 9)	(5, 6)	
3:	(6, 7)			
4:	(1, 1)	(3, 8)	(6, 2)	(7, 4)
5:	(4, 8)	(8, 3)		
6:	(7, 3)	(9, 7)		
7:	(6, 5)	(8, 1)	(10, 4)	
8:	(11, 2)			
9:	(10, 4)			
10:	(11, 6)			
11:				

Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	10	4	0	0	0	0	0	0	0	0	0
1	0	0	3	3	0	0	0	0	0	0	0	0
2	0	7	0	0	9	6	0	0	0	0	0	0
3	0	0	0	0	0	0	7	0	0	0	0	0
4	0	1	0	8	0	0	2	4	0	0	0	0
5	0	0	0	0	8	0	0	0	3	0	0	0
6	0	0	0	0	0	0	0	3	0	7	0	0
7	0	0	0	0	0	0	5	0	1	0	4	0
8	0	0	0	0	0	0	0	0	0	0	0	2
9	0	0	0	0	0	0	0	0	0	0	4	0
10	0	0	0	0	0	0	0	0	0	0	0	6
11	0	0	0	0	0	0	0	0	0	0	0	0

Input

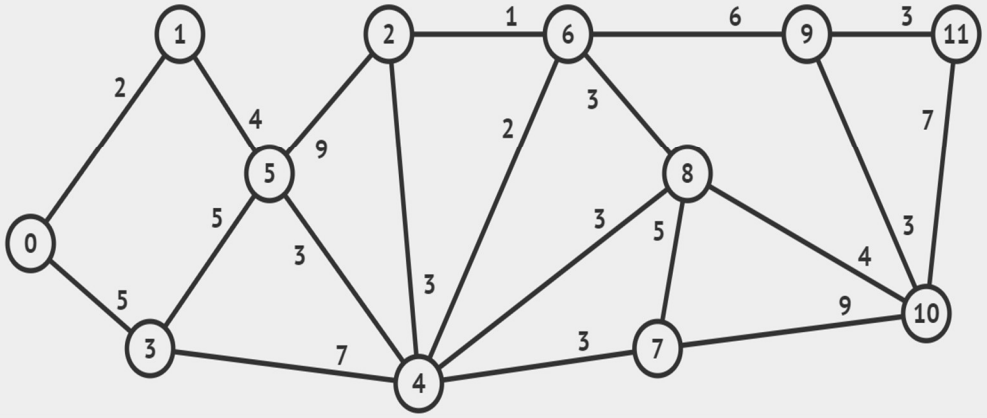
```
// to travel from vertex [X = src] to [Y = dest], one has to cover [Z = cost] units of distance
// adj_list.get(X).add(new Node(Y, Z));

// Test Case 2 / Graph 2 (Directed)
adj_list.get(0).add(new Node( vertex: 1,  cost: 10));
adj_list.get(0).add(new Node( vertex: 2,  cost: 4));

adj_list.get(1).add(new Node( vertex: 2,  cost: 3));
adj_list.get(1).add(new Node( vertex: 3,  cost: 3));

adj_list.get(2).add(new Node( vertex: 1,  cost: 7));
adj_list.get(2).add(new Node( vertex: 4,  cost: 9));
adj_list.get(2).add(new Node( vertex: 5,  cost: 6));

adj_list.get(3).add(new Node( vertex: 6,  cost: 7));

adj_list.get(4).add(new Node( vertex: 1,  cost: 1));
adj_list.get(4).add(new Node( vertex: 3,  cost: 8));
adj_list.get(4).add(new Node( vertex: 6,  cost: 2));
adj_list.get(4).add(new Node( vertex: 7,  cost: 4));

adj_list.get(5).add(new Node( vertex: 4,  cost: 8));
adj_list.get(5).add(new Node( vertex: 8,  cost: 3));

adj_list.get(6).add(new Node( vertex: 7,  cost: 3));
adj_list.get(6).add(new Node( vertex: 9,  cost: 7));

adj_list.get(7).add(new Node( vertex: 6,  cost: 5));
adj_list.get(7).add(new Node( vertex: 8,  cost: 1));
adj_list.get(7).add(new Node( vertex: 10,  cost: 4));

adj_list.get(8).add(new Node( vertex: 11,  cost: 2));

adj_list.get(9).add(new Node( vertex: 10,  cost: 4));

adj_list.get(10).add(new Node( vertex: 11,  cost: 6));
```

Output

```
Path (0 → 1): Path Cost = 10, Path Route = [0, 1]
Path (0 → 2): Path Cost = 4, Path Route = [0, 2]
Path (0 → 3): Path Cost = 13, Path Route = [0, 1, 3]
Path (0 → 4): Path Cost = 13, Path Route = [0, 2, 4]
Path (0 → 5): Path Cost = 10, Path Route = [0, 2, 5]
Path (0 → 6): Path Cost = 15, Path Route = [0, 2, 4, 6]
Path (0 → 7): Path Cost = 17, Path Route = [0, 2, 4, 7]
Path (0 → 8): Path Cost = 13, Path Route = [0, 2, 5, 8]
Path (0 → 9): Path Cost = 22, Path Route = [0, 2, 4, 6, 9]
Path (0 → 10): Path Cost = 21, Path Route = [0, 2, 4, 7, 10]
Path (0 → 11): Path Cost = 15, Path Route = [0, 2, 5, 8, 11]
--------------------------------------------------------------------------------
The shortest path from the source node to other nodes:
[Source Node] →        [Node #]        Path (Cost):        Parent Node:
    0                       0               0                   -1
    0                       1               10                      0
    0                       2               4                   0
    0                       3               13                      1
    0                       4               13                      2
    0                       5               10                      2
    0                       6               15                      4
    0                       7               17                      4
    0                       8               13                      5
    0                       9               22                      6
    0                       10                  21                      7
    0                       11                  15                      8
```

Test Case 3 – Undirected / Weighted

UNDIRECTED/WEIGHTED D/U D/W



V=12, E=20

Edge List

0:	0	1	2
1:	0	3	5
2:	1	5	4
3:	2	4	3
4:	2	5	9
5:	2	6	1
6:	3	4	7
7:	3	5	5
8:	4	5	3
9:	4	6	2
10:	4	7	3
11:	4	8	3
12:	6	8	3
13:	6	9	6
14:	7	8	5
15:	7	10	9
16:	8	10	4
17:	9	10	3
18:	9	11	3
19:	10	11	7

Adjacency List

0:	(1, 2)	(3, 5)				
1:	(0, 2)	(5, 4)				
2:	(4, 3)	(5, 9)	(6, 1)			
3:	(0, 5)	(4, 7)	(5, 5)			
4:	(2, 3)	(3, 7)	(5, 3)	(6, 2)	(7, 3)	(8, 3)
5:	(1, 4)	(2, 9)	(3, 5)	(4, 3)		
6:	(2, 1)	(4, 2)	(8, 3)	(9, 6)		
7:	(4, 3)	(8, 5)	(10, 9)			
8:	(4, 3)	(6, 3)	(7, 5)	(10, 4)		
9:	(6, 6)	(10, 3)	(11, 3)			
10:	(7, 9)	(8, 4)	(9, 3)	(11, 7)		
11:	(9, 3)	(10, 7)				

Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	2	0	5	0	0	0	0	0	0	0	0
1	2	0	0	0	0	4	0	0	0	0	0	0
2	0	0	0	0	3	9	1	0	0	0	0	0
3	5	0	0	0	7	5	0	0	0	0	0	0
4	0	0	3	7	0	3	2	3	3	0	0	0
5	0	4	9	5	3	0	0	0	0	0	0	0
6	0	0	1	0	2	0	0	0	3	6	0	0
7	0	0	0	0	3	0	0	0	5	0	9	0
8	0	0	0	0	3	0	3	5	0	0	4	0
9	0	0	0	0	0	0	6	0	0	0	3	3
10	0	0	0	0	0	0	0	9	4	3	0	7
11	0	0	0	0	0	0	0	0	0	3	7	0

Input

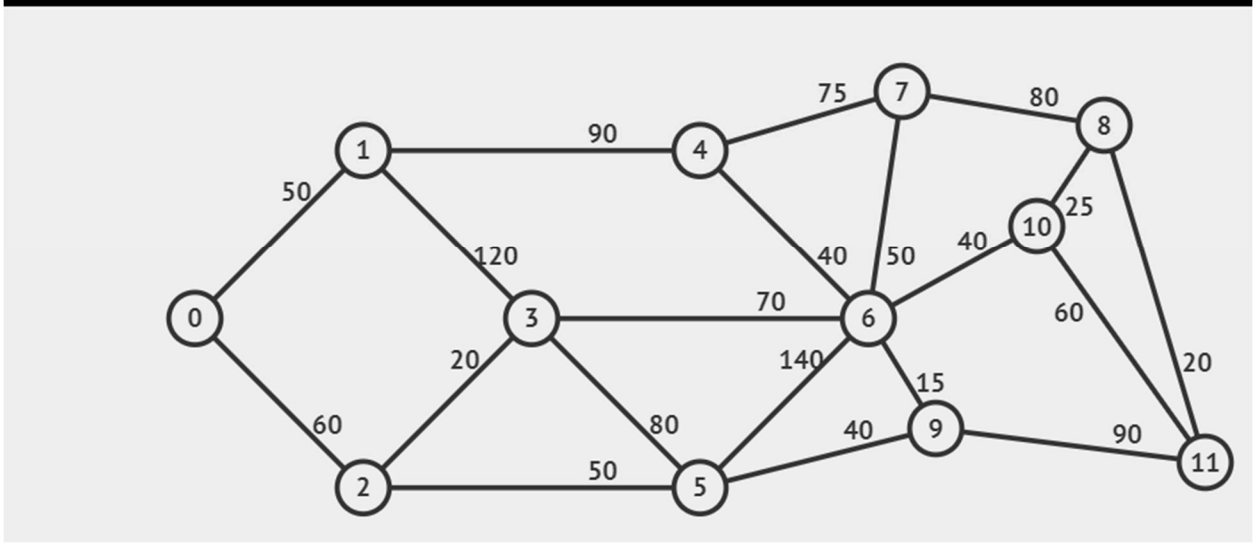
```
// to travel from vertex [X = src] to [Y = dest], one has to cover [Z = cost] units of distance
// adj_list.get(X).add(new Node(Y, Z));

// Test Case 3 / Graph 3 (Undirected)
adj_list.get(0).add(new Node( vertex: 1,  cost: 2));
adj_list.get(0).add(new Node( vertex: 3,  cost: 5));

adj_list.get(1).add(new Node( vertex: 0,  cost: 2));
adj_list.get(1).add(new Node( vertex: 5,  cost: 4));

adj_list.get(2).add(new Node( vertex: 4,  cost: 3));
adj_list.get(2).add(new Node( vertex: 5,  cost: 9));
adj_list.get(2).add(new Node( vertex: 6,  cost: 1));

adj_list.get(3).add(new Node( vertex: 0,  cost: 5));
adj_list.get(3).add(new Node( vertex: 4,  cost: 7));
adj_list.get(3).add(new Node( vertex: 5,  cost: 5));

adj_list.get(4).add(new Node( vertex: 2,  cost: 3));
adj_list.get(4).add(new Node( vertex: 3,  cost: 7));
adj_list.get(4).add(new Node( vertex: 5,  cost: 3));
adj_list.get(4).add(new Node( vertex: 6,  cost: 2));
adj_list.get(4).add(new Node( vertex: 7,  cost: 3));
adj_list.get(4).add(new Node( vertex: 8,  cost: 3));

adj_list.get(5).add(new Node( vertex: 1,  cost: 4));
adj_list.get(5).add(new Node( vertex: 2,  cost: 9));
adj_list.get(5).add(new Node( vertex: 3,  cost: 5));
adj_list.get(5).add(new Node( vertex: 4,  cost: 3));

adj_list.get(6).add(new Node( vertex: 2,  cost: 1));
adj_list.get(6).add(new Node( vertex: 4,  cost: 2));
adj_list.get(6).add(new Node( vertex: 8,  cost: 3));
adj_list.get(6).add(new Node( vertex: 9,  cost: 6));

adj_list.get(7).add(new Node( vertex: 4,  cost: 3));
adj_list.get(7).add(new Node( vertex: 8,  cost: 5));
adj_list.get(7).add(new Node( vertex: 10,  cost: 9));

adj_list.get(8).add(new Node( vertex: 4,  cost: 3));
adj_list.get(8).add(new Node( vertex: 6,  cost: 3));
adj_list.get(8).add(new Node( vertex: 7,  cost: 5));
adj_list.get(8).add(new Node( vertex: 10,  cost: 4));

adj_list.get(9).add(new Node( vertex: 6,  cost: 6));
adj_list.get(9).add(new Node( vertex: 10,  cost: 3));
adj_list.get(9).add(new Node( vertex: 11,  cost: 3));

adj_list.get(10).add(new Node( vertex: 7,  cost: 9));
adj_list.get(10).add(new Node( vertex: 8,  cost: 4));
adj_list.get(10).add(new Node( vertex: 9,  cost: 3));
adj_list.get(10).add(new Node( vertex: 11,  cost: 7));

adj_list.get(11).add(new Node( vertex: 9,  cost: 3));
adj_list.get(11).add(new Node( vertex: 10,  cost: 7));
```

Output

```
Path (0 → 1): Path Cost = 2, Path Route = [0, 1]
Path (0 → 2): Path Cost = 12, Path Route = [0, 1, 5, 4, 2]
Path (0 → 3): Path Cost = 5, Path Route = [0, 3]
Path (0 → 4): Path Cost = 9, Path Route = [0, 1, 5, 4]
Path (0 → 5): Path Cost = 6, Path Route = [0, 1, 5]
Path (0 → 6): Path Cost = 11, Path Route = [0, 1, 5, 4, 6]
Path (0 → 7): Path Cost = 12, Path Route = [0, 1, 5, 4, 7]
Path (0 → 8): Path Cost = 12, Path Route = [0, 1, 5, 4, 8]
Path (0 → 9): Path Cost = 17, Path Route = [0, 1, 5, 4, 6, 9]
Path (0 → 10): Path Cost = 16, Path Route = [0, 1, 5, 4, 8, 10]
Path (0 → 11): Path Cost = 20, Path Route = [0, 1, 5, 4, 6, 9, 11]
--------------------------------------------------------------------------------
The shortest path from the source node to other nodes:
[Source Node] →        [Node #]        Path (Cost):        Parent Node:
    0                       0               0                   -1
    0                       1               2                   0
    0                       2               12                      4
    0                       3               5                   0
    0                       4               9                   5
    0                       5               6                   1
    0                       6               11                      4
    0                       7               12                      4
    0                       8               12                      4
    0                       9               17                      6
    0                       10                  16                      8
    0                       11                  20                      9
```

Test Case 4 – Undirected / Weighted

UNDIRECTED/WEIGHTED D/U D/W



V=12, E=20

Edge List

0:	0	1	50
1:	0	2	60
2:	1	3	120
3:	1	4	90
4:	2	3	20
5:	2	5	50
6:	3	5	80
7:	3	6	70
8:	4	6	40
9:	4	7	75
10:	5	6	140
11:	5	9	40
12:	6	7	50
13:	6	9	15
14:	6	10	40
15:	7	8	80
16:	8	10	25
17:	8	11	20
18:	9	11	90
19:	10	11	60

Adjacency List

0:	(1, 50)	(2, 60)				
1:	(0, 50)	(3, 120)	(4, 90)			
2:	(0, 60)	(3, 20)	(5, 50)			
3:	(1, 120)	(2, 20)	(5, 80)	(6, 70)		
4:	(1, 90)	(6, 40)	(7, 75)			
5:	(2, 50)	(3, 80)	(6, 140)	(9, 40)		
6:	(3, 70)	(4, 40)	(5, 140)	(7, 50)	(9, 15)	(10, 40)
7:	(4, 75)	(6, 50)	(8, 80)			
8:	(7, 80)	(10, 25)	(11, 20)			
9:	(5, 40)	(6, 15)	(11, 90)			
10:	(6, 40)	(8, 25)	(11, 60)			
11:	(8, 20)	(9, 90)	(10, 60)			

Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	50	60	0	0	0	0	0	0	0	0	0
1	50	0	0	120	90	0	0	0	0	0	0	0
2	60	0	0	20	0	50	0	0	0	0	0	0
3	0	120	20	0	0	80	70	0	0	0	0	0
4	0	90	0	0	0	0	40	75	0	0	0	0
5	0	0	50	80	0	0	140	0	0	40	0	0
6	0	0	0	70	40	140	0	50	0	15	40	0
7	0	0	0	0	75	0	50	0	80	0	0	0
8	0	0	0	0	0	0	0	80	0	0	25	20
9	0	0	0	0	0	40	15	0	0	0	0	90
10	0	0	0	0	0	0	40	0	25	0	0	60
11	0	0	0	0	0	0	0	0	20	90	60	0

Input

```
// to travel from vertex [X = src] to [Y = dest], one has to cover [Z = cost] units of distance
// adj_list.get(X).add(new Node(Y, Z));

// Test Case 4 / Graph 4 (Undirected)
adj_list.get(0).add(new Node( vertex: 1,  cost: 50));
adj_list.get(0).add(new Node( vertex: 2,  cost: 60));

adj_list.get(1).add(new Node( vertex: 0,  cost: 50));
adj_list.get(1).add(new Node( vertex: 3,  cost: 120));
adj_list.get(1).add(new Node( vertex: 4,  cost: 90));

adj_list.get(2).add(new Node( vertex: 0,  cost: 60));
adj_list.get(2).add(new Node( vertex: 3,  cost: 20));
adj_list.get(2).add(new Node( vertex: 5,  cost: 50));

adj_list.get(3).add(new Node( vertex: 1,  cost: 120));
adj_list.get(3).add(new Node( vertex: 2,  cost: 20));
adj_list.get(3).add(new Node( vertex: 5,  cost: 80));
adj_list.get(3).add(new Node( vertex: 6,  cost: 70));

adj_list.get(4).add(new Node( vertex: 1,  cost: 90));
adj_list.get(4).add(new Node( vertex: 6,  cost: 40));
adj_list.get(4).add(new Node( vertex: 7,  cost: 75));

adj_list.get(5).add(new Node( vertex: 2,  cost: 50));
adj_list.get(5).add(new Node( vertex: 3,  cost: 80));
adj_list.get(5).add(new Node( vertex: 6,  cost: 140));
adj_list.get(5).add(new Node( vertex: 9,  cost: 40));

adj_list.get(6).add(new Node( vertex: 3,  cost: 70));
adj_list.get(6).add(new Node( vertex: 4,  cost: 40));
adj_list.get(6).add(new Node( vertex: 5,  cost: 140));
adj_list.get(6).add(new Node( vertex: 7,  cost: 50));
adj_list.get(6).add(new Node( vertex: 9,  cost: 15));
adj_list.get(6).add(new Node( vertex: 10,  cost: 40));

adj_list.get(7).add(new Node( vertex: 4,  cost: 75));
adj_list.get(7).add(new Node( vertex: 6,  cost: 50));
adj_list.get(7).add(new Node( vertex: 8,  cost: 80));

adj_list.get(8).add(new Node( vertex: 7,  cost: 80));
adj_list.get(8).add(new Node( vertex: 10,  cost: 25));
adj_list.get(8).add(new Node( vertex: 11,  cost: 20));

adj_list.get(9).add(new Node( vertex: 5,  cost: 40));
adj_list.get(9).add(new Node( vertex: 6,  cost: 15));
adj_list.get(9).add(new Node( vertex: 11,  cost: 90));

adj_list.get(10).add(new Node( vertex: 6,  cost: 40));
adj_list.get(10).add(new Node( vertex: 8,  cost: 25));
adj_list.get(10).add(new Node( vertex: 11,  cost: 60));

adj_list.get(11).add(new Node( vertex: 8,  cost: 20));
adj_list.get(11).add(new Node( vertex: 9,  cost: 90));
adj_list.get(11).add(new Node( vertex: 10,  cost: 60));
```

Output

```
Path (0 → 1): Path Cost = 50, Path Route = [0, 1]
Path (0 → 2): Path Cost = 60, Path Route = [0, 2]
Path (0 → 3): Path Cost = 80, Path Route = [0, 2, 3]
Path (0 → 4): Path Cost = 140, Path Route = [0, 1, 4]
Path (0 → 5): Path Cost = 110, Path Route = [0, 2, 5]
Path (0 → 6): Path Cost = 150, Path Route = [0, 2, 3, 6]
Path (0 → 7): Path Cost = 200, Path Route = [0, 2, 3, 6, 7]
Path (0 → 8): Path Cost = 215, Path Route = [0, 2, 3, 6, 10, 8]
Path (0 → 9): Path Cost = 150, Path Route = [0, 2, 5, 9]
Path (0 → 10): Path Cost = 190, Path Route = [0, 2, 3, 6, 10]
Path (0 → 11): Path Cost = 235, Path Route = [0, 2, 3, 6, 10, 8, 11]
--------------------------------------------------------------------------------
The shortest path from the source node to other nodes:
[Source Node] →        [Node #]        Path (Cost):        Parent Node:
    0                       0               0                   -1
    0                       1               50                      0
    0                       2               60                      0
    0                       3               80                      2
    0                       4               140                     1
    0                       5               110                     2
    0                       6               150                     3
    0                       7               200                     6
    0                       8               215                     10
    0                       9               150                     5
    0                       10                  190                     6
    0                       11                  235                     8
```

Problem 2: Minimum Spanning Tree Algorithm

Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight $w(T)=\sum_{(u,v)\in T} w(u,v)$. Use either Kruskal's or Prim's algorithm to find Minimum Spanning Tree (MST). You will printout edges of the tree and total cost of minimum spanning tree.

Input Format:

For each problem, you will take input from a text file. Say you want to run algorithm on the following undirected graph.

For this problem, we are using Prim's Minimum Spanning Tree algorithm

Prim's algorithm:

- *First, we have to initialize an MST with the randomly chosen vertex.*
- *Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.*
- *Repeat step 2 until the minimum spanning tree is formed.*

Pseudocode:

T = Ø;

U = { 1 };

while (U ≠ V)

let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;

T = T ∪ {(u, v)}

U = U ∪ {v}

Code:

```
class Graph():
  INF = 999999
  def __init__(self, num_vertices):
      self.V = num_vertices
      self.graph = [[0 for column in range(num_vertices)] for row in range(num_vertices)]


  def printMST(self, start_node):
      print("Edge     Weight")
      for i in range(1, self.V):
          print(f"{start_node[i]} - {i}      {self.graph[i][start_node[i]]}")
```

```
  def findMinimumKey(self, key, mst_list):
      mini = self.INF
      for v in range(self.V):
          if key[v] < mini and mst_list[v] == False:
              mini = key[v]
              minKey_index = v
      return minKey_index


  def primsAlgorithmImpl(self):
      key = [self.INF for _ in range(self.V)]

      start_node = [None for _ in range(self.V)]

      key[0] = 0

      mst_list = [False for _ in range(self.V)]

      start_node[0] = -1

      for _ in range(self.V):

          u = self.findMinimumKey(key, mst_list)
          # 2) add the new vertex to the MST
          mst_list[u] = True


          for v in range(self.V):
              if self.graph[u][v] > 0 and mst_list[v] == False and key[v] > self.graph[u][v]:
                  key[v] = self.graph[u][v]
                  start_node[v] = u

      self.printMST(start_node)
def toNumeral(a):
  return ord(a)-65



graph = []
adjacency_matrix = []
line = []
with open('Problem2_Input.txt','r') as f:
  l = f.readlines()
for s in l:
  temp = s.split()
  line.append(temp)



vertices = int(line[0][0])
for i in range(1,len(line)-1):
  from_node = toNumeral(line[i][0])
  to_node = toNumeral(line[i][1])
  distance = int(line[i][2])
  graph.append([from_node,to_node,distance])
```

```
# print(graph)
for i in range(vertices):
    row = []
    for j in range(vertices):
        row.append(0)
    adjacency_matrix.append(row)


if line[0][2]=='U':
  for edge in graph:
      from_edge = edge[0]
      to_edge = edge[1]
      weight = edge[2]
      adjacency_matrix[from_edge][to_edge] = weight
      adjacency_matrix[to_edge][from_edge] = weight
else:
  for edge in graph:
      from_edge = edge[0]
      to_edge = edge[1]
      weight = edge[2]
      adjacency_matrix[from_edge][to_edge] = weight

g = Graph(vertices)
g.graph = adjacency_matrix
print(adjacency_matrix)
g.primsAlgorithmImpl()
```

Code Analysis:

Time Complexity: O (VlogV + ElogV)
Data Structures used: Graph, Stacks
Implemented using: Lists, Set

Input #1

Undirected graph:

6 10 U

A B 1

A C 2

B C 1

B D 3

B E 2

C D 1

C E 2

D E 4

D F 3

E F 3

A

We feed the data through a text file

Output:

Adjacency matrix created:

[[0, 1, 2, 0, 0, 0],
[0, 0, 1, 3, 1, 0],
[0, 0, 0, 1, 2, 0],
[0, 0, 0, 0, 4, 6],
[0, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 0]]

Output received:

Edge Weight

0 - 1 1

1 - 2 1

2 - 3 1

1 - 4 2

3 - 5 3

Input #2:

7 10 U

A B 2

A G 1

A C 4

B C 6

B D 1

B E 3

C D 2

C G 2

C E 1

D E 5

D F 2

E F 3

F G 4

Output:

Edge	Weight
0 - 1	2
6 - 2	2
1 - 3	1
2 - 4	1
3 - 5	2
0 - 6	1

Input #3:

8 10 U

A B 2

A G 1

A C 4

B C 6

B D 1

B E 3

B H 1

C D 2

C G 2

C E 1

D E 5

D F 2

E F 3

F G 4

G H 2

Output:

Edge	Weight
0 - 1	2
6 - 2	2
1 - 3	1
2 - 4	1
3 - 5	2
0 - 6	1
1 - 7	1

Input #4:

9 10 U

A B 2

A G 1

A C 4

B C 6

B I 4

B D 1

B E 2

B H 6

C D 2

C I 2

C G 2

C E 1

D E 5

D F 2

F I 1

E F 3

F G 4

G H 2

Output:

Edge	Weight
0 - 1	2
6 - 2	2
1 - 3	1
2 - 4	1
3 - 5	2
0 - 6	1
6 - 7	2
5 - 8	1

Why does Prim's algorithm not work with directed graphs?

Prim's algorithm works on the concept that every node is accessible by every node. But in a directed graph, that is not possible. Therefore, Prim's algorithm fails for directed graphs.

Problem 3: Finding Strongly Connected Components

Given a directed graph G with n vertices and m edges. Decompose this graph into Strongly Connected Components (SCCs) and print the components.

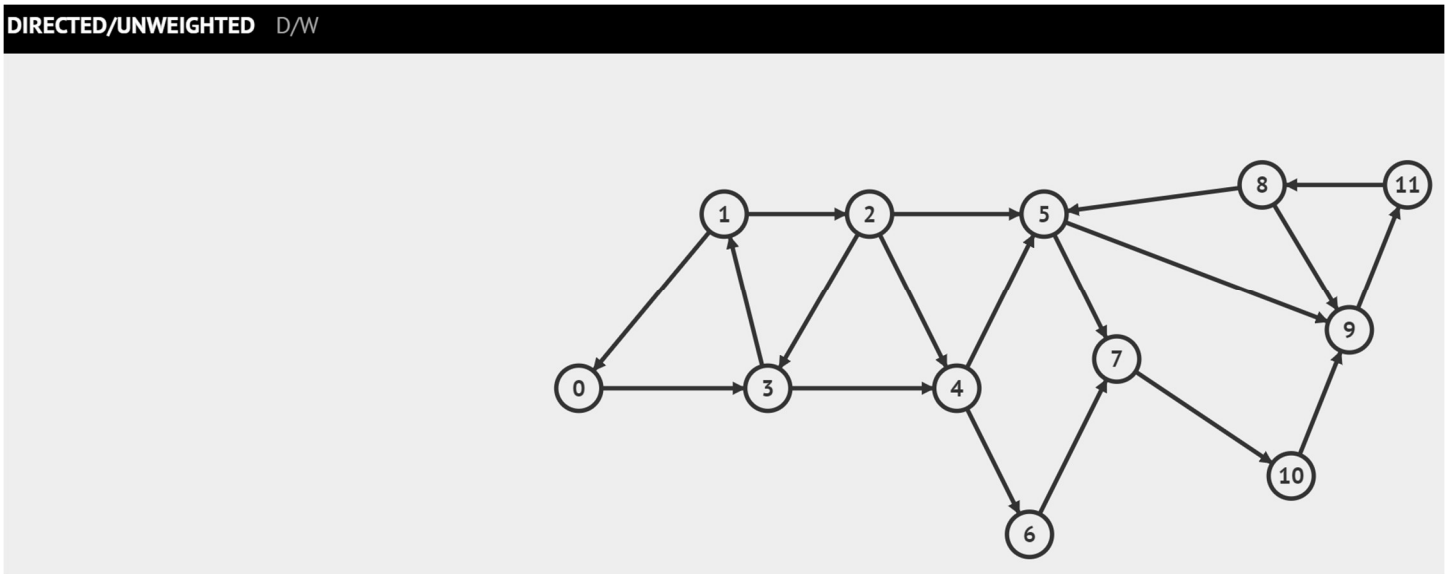
We can utilize the Strongly-Connected-Components Algorithm which utilizes DFS Algorithm. (Additional running time Analysis is in the code comments).

Pseudocode

1. Call DFS(Graph) to compute finishing times u.f for each vertex u.
2. Compute Graph transpose (G^T)
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing u.f (as computed on line 1)
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Total Running time = O(n + m) or O(V + E)

Test Case 1 – Directed / Unweighted



V=12, E=19

Edge List

0:	0	3
1:	1	0
2:	1	2
3:	2	3
4:	2	4
5:	2	5
6:	3	1
7:	3	4
8:	4	5
9:	4	6
10:	5	7
11:	5	9
12:	6	7
13:	7	10
14:	8	5
15:	8	9
16:	9	11
17:	10	9
18:	11	8

Adjacency List

0:	3		
1:	0	2	
2:	3	4	5
3:	1	4	
4:	5	6	
5:	7	9	
6:	7		
7:	10		
8:	5	9	
9:	11		
10:	9		
11:	8		

Adjacency Matrix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	
0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	1	1	1	0	0	0	0	0	0
3	0	1	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	1	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	1	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	1	0	0	0

Input

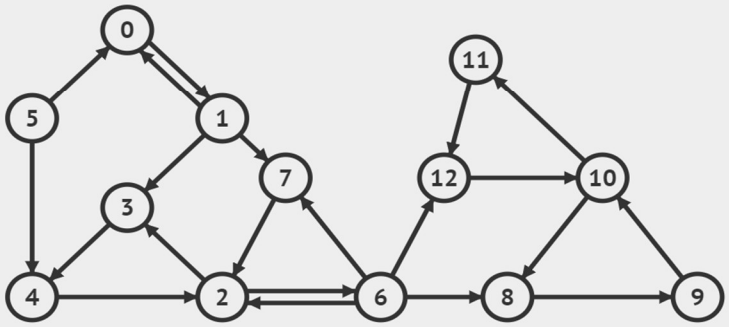
```
// Test Case 1
// Create a graph
Graph g = new Graph( v: 12);
g.addEdge( v: 0,  w: 3);

g.addEdge( v: 1,  w: 0);
g.addEdge( v: 1,  w: 2);

g.addEdge( v: 2,  w: 3);
g.addEdge( v: 2,  w: 4);
g.addEdge( v: 2,  w: 5);

g.addEdge( v: 3,  w: 1);
g.addEdge( v: 3,  w: 4);

g.addEdge( v: 4,  w: 5);
g.addEdge( v: 4,  w: 6);

g.addEdge( v: 5,  w: 7);
g.addEdge( v: 5,  w: 9);

g.addEdge( v: 6,  w: 7);

g.addEdge( v: 7,  w: 10);

g.addEdge( v: 8,  w: 5);
g.addEdge( v: 8,  w: 9);

g.addEdge( v: 9,  w: 11);

g.addEdge( v: 10,  w: 9);

g.addEdge( v: 11,  w: 8);
```

Output

```
Strongly Connected Components (Separated by new line):
0 1 3 2   <-- (Strongly Connected Component #: 1)
4   <-- (Strongly Connected Component #: 2)
6   <-- (Strongly Connected Component #: 3)
5 8 11 9 10 7   <-- (Strongly Connected Component #: 4)
```

Test Case 2 – Directed / Unweighted

DIRECTED/UNWEIGHTED D/W



V=13, E=21

Edge List

0:	0	1
1:	1	0
2:	1	3
3:	1	7
4:	2	3
5:	2	6
6:	3	4
7:	4	2
8:	5	0
9:	5	4
10:	6	2
11:	6	7
12:	6	8
13:	6	12
14:	7	2
15:	8	9
16:	9	10
17:	10	8
18:	10	11
19:	11	12
20:	12	10

Adjacency List

0:	1			
1:	0	3	7	
2:	3	6		
3:	4			
4:	2			
5:	0	4		
6:	2	7	8	12
7:	2			
8:	9			
9:	10			
10:	8	11		
11:	12			
12:	10			

Adjacency Matrix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	1	0	0	0	0	0
2	0	0	0	1	0	0	1	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	1	0	0	0	0	1	1	0	0	0	1
7	0	0	1	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	0	0
10	0	0	0	0	0	0	0	0	1	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	1	0	0

Input

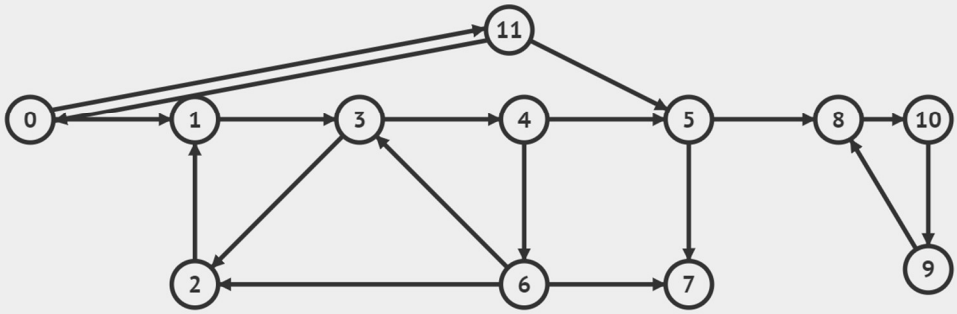
```
// Test Case 2
// Create a graph
Graph g = new Graph( v: 13);
g.addEdge( v: 0,  w: 1);
g.addEdge( v: 1,  w: 3);
g.addEdge( v: 1,  w: 7);
g.addEdge( v: 1,  w: 0);
g.addEdge( v: 2,  w: 3);
g.addEdge( v: 2,  w: 6);
g.addEdge( v: 3,  w: 4);
g.addEdge( v: 4,  w: 2);
g.addEdge( v: 5,  w: 0);
g.addEdge( v: 5,  w: 4);
g.addEdge( v: 6,  w: 2);
g.addEdge( v: 6,  w: 7);
g.addEdge( v: 6,  w: 8);
g.addEdge( v: 6,  w: 12);
g.addEdge( v: 7,  w: 2);
g.addEdge( v: 8,  w: 9);
g.addEdge( v: 9,  w: 10);
g.addEdge( v: 10,  w: 8);
g.addEdge( v: 10,  w: 11);
g.addEdge( v: 11,  w: 12);
g.addEdge( v: 12,  w: 10);
```

Output

```
Strongly Connected Components (Separated by new line):
5   <-- (Strongly Connected Component #: 1)
0 1   <-- (Strongly Connected Component #: 2)
3 2 4 6 7   <-- (Strongly Connected Component #: 3)
8 10 9 12 11   <-- (Strongly Connected Component #: 4)
```

Test Case 3 – Directed / Unweighted

DIRECTED/UNWEIGHTED D/W



V=12, E=18

Edge List

0:	0	1
1:	0	11
2:	1	3
3:	2	1
4:	3	2
5:	3	4
6:	4	5
7:	4	6
8:	5	7
9:	5	8
10:	6	2
11:	6	3
12:	6	7
13:	8	10
14:	9	8
15:	10	9
16:	11	0
17:	11	5

Adjacency List

0:	1	11	
1:	3		
2:	1		
3:	2	4	
4:	5	6	
5:	7	8	
6:	2	3	7
7:			
8:	10		
9:	8		
10:	9		
11:	0	5	

Adjacency Matrix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	
0	0	1	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	1	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	0	0	0	0	0
5	0	0	0	0	0	0	0	1	1	0	0	0
6	0	0	1	1	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	0	0	1	0	0
11	1	0	0	0	0	1	0	0	0	0	0	0

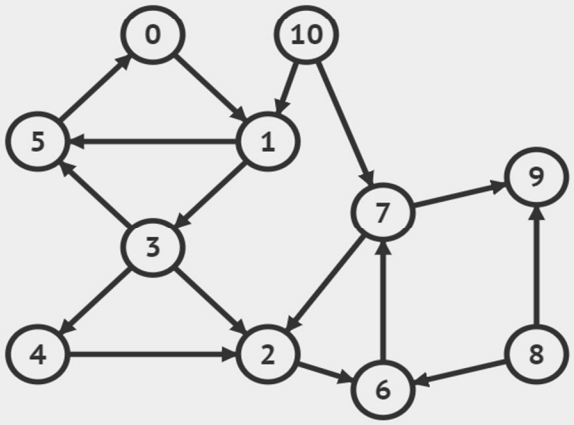
Input

```
// Test Case 3
// Create a graph
Graph g = new Graph( v: 12);
g.addEdge( v: 0,  w: 1);
g.addEdge( v: 0,  w: 11);
g.addEdge( v: 1,  w: 3);
g.addEdge( v: 2,  w: 1);
g.addEdge( v: 3,  w: 2);
g.addEdge( v: 3,  w: 4);
g.addEdge( v: 4,  w: 5);
g.addEdge( v: 4,  w: 6);

g.addEdge( v: 5,  w: 7);
g.addEdge( v: 5,  w: 8);

g.addEdge( v: 6,  w: 2);
g.addEdge( v: 6,  w: 3);
g.addEdge( v: 6,  w: 7);



g.addEdge( v: 8,  w: 10);

g.addEdge( v: 9,  w: 8);

g.addEdge( v: 10,  w: 9);

g.addEdge( v: 11,  w: 5);
g.addEdge( v: 11,  w: 0);
```

Output

```
Strongly Connected Components (Separated by new line):
0 11   <-- (Strongly Connected Component #: 1)
1 2 3 6 4   <-- (Strongly Connected Component #: 2)
5   <-- (Strongly Connected Component #: 3)
8 9 10   <-- (Strongly Connected Component #: 4)
7   <-- (Strongly Connected Component #: 5)
```

DIRECTED/UNWEIGHTED D/W



V=11, E=15

Edge List

0:	0	1
1:	1	3
2:	1	5
3:	2	6
4:	3	2
5:	3	4
6:	3	5
7:	4	2
8:	5	0
9:	6	7
10:	7	2
11:	7	9
12:	8	6
13:	8	9
14:	10	1
15:	10	7

Adjacency List

0:	1		
1:	3	5	
2:	6		
3:	2	4	5
4:	2		
5:	0		
6:	7		
7:	2	9	
8:	6	9	
9:			
10:	1	7	

Adjacency Matrix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0
3	0	0	1	0	1	1	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0	0	1	0
9	0	0	0	0	0	0	0	0	0	0	0
10	0	1	0	0	0	0	0	1	0	0	0

Input

```
// Test Case 4
// Create a graph
Graph g = new Graph( v: 11);
g.addEdge( v: 0,  w: 1);

g.addEdge( v: 1,  w: 5);
g.addEdge( v: 1,  w: 3);

g.addEdge( v: 2,  w: 6);

g.addEdge( v: 3,  w: 2);
g.addEdge( v: 3,  w: 4);
g.addEdge( v: 3,  w: 5);

g.addEdge( v: 4,  w: 2);

g.addEdge( v: 5,  w: 0);

g.addEdge( v: 6,  w: 7);

g.addEdge( v: 7,  w: 2);
g.addEdge( v: 7,  w: 9);

g.addEdge( v: 8,  w: 9);
g.addEdge( v: 8,  w: 6);



g.addEdge( v: 10,  w: 1);
g.addEdge( v: 10,  w: 7);
```

Output

```
Strongly Connected Components (Separated by new line):
10   <-- (Strongly Connected Component #: 1)
8   <-- (Strongly Connected Component #: 2)
0 5 1 3   <-- (Strongly Connected Component #: 3)
4   <-- (Strongly Connected Component #: 4)
2 7 6   <-- (Strongly Connected Component #: 5)
9   <-- (Strongly Connected Component #: 6)
```