CHARLOTTE

Project - 2
Graph Algorithms and Related Data Structures
ITCS-6114 FALL 2021

Submitted by

Ramanathan Sivaramakrishnan
[801243565]

Sharat Sindoor
[80123539]

Problem Statement:

Problem 1: Single-source Shortest Path Algorithm

Find shortest path tree in both directed and undirected weighted graphs for a given source vertex. Assume there is no negative edge in your graph.

Problem 2: Minimum Spanning Tree Algorithm

Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight

$w(T)=\sum(u,v)\in Tw(u,v)w(T)=\sum(u,v)\in Tw(u,v)$. Use either Kruskal's or Prim's algorithm to find Minimum Spanning Tree (MST).

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.

Problem 1: Single-source Shortest Path Algorithm

The Single-Source Shortest Path problem consists of finding the shortest paths between a given vertex *v* and all other vertices in the graph. This problem is mostly solved using Dijkstra, though in this case a single result is kept and other shortest paths are discarded.

Dijkstra's Algorithm:

• It computes the shortest distances of all the vertices from a given start vertex s.

• Dijkstra's algorithm is a kind of greedy algorithm. It assumes that all the edges in G are nonnegative.

• It grows a cloud or set of vertices beginning with the start vertex s, and eventually covering all other vertices.

• It maintains a set S of vertices whose final shortest path weights from the source s have already been determined.For each vertex v, a label d(v) is present which represents the distance of vertex v from s in the subgraph consisting of the cloud and its adjacent vertices.

Data Structures Used and Analysis of Runtime:

• The data structures used to implement Dijkstra's algorithm for finding the shortest path in directed and undirected graphs are Array List, Adjacency List, Graphs, Hash Map, Priority Queue, Tree Set.

• In Dijkstra's algorithm, we used binary min-heap for priority queue. This takes O (m log n) time where m is the number of edges and n is the number of vertices.

• Each EXTRACT-MIN operation takes O (log n) time whereas each RELAX operation takes O (log n) time for at most O (m) such operations.

• Therefore, the total running time for Dijkstra's algorithm would be, O ((n + m) log n).

• If all vertices v ∈G are reachable from the source s, then the running time would be O(mlog n).

Pseudocode:

```
Algorithm DIJKSTRA ( G, w, s)
        INITIALIZE_SINGLE_SOURCE (G, s)
        S = ∅
        Q = G.V
         while Q ≠∅ do
                u = EXTRACT-MIN(Q)
                S = S ∪ {u}
                 for each vertex v ∈G.Adj[u] do
                        RELAX (u, v, w)

 Algorithm INITIALIZE_SINGLE_SOURCE (G, s)
        for each vertex v ∈G.V do
                d[v] = ∞
                π[v] = NIL d[s] = 0
```

Implementation of Dijkstra's Directed Graph:

```
from collections import deque, namedtuple
"""INITIALIZING ALL THE VERTICES WITH THE COST AS INFINITY."""
intial_values = float('inf')
edge = namedtuple('edge', 'start, end, cost')
def make_edge(start, end, cost=1):
    return edge(start, end, cost)

class Graph:
    """
    THIS CONSTRUCTOR IS USED TO INTIALSE THE GRAPH WITH THE BASIC VALUES.
    """
    def __init__(self, edges):
        not_correct_edges = [i for i in edges if len(i) not in [2, 3]]
        if not_correct_edges:
            raise ValueError('EDGES ARE IN WRONG FORMAT: {}'.format(not_correct_edges))
        self.edges = [make_edge(*_) for _ in edges]

    """
    IMPLEMENTING THE DIJSTRA'S ALGORITHM
    FOR THE DIRECTED GRAPH
    """
    def implementing_dijkstra_algorithm(self, start, destination):
        global path_cost
        assert start in self.vertices, 'THERE IS NO SUCH START NODE IN THE GRAPH'
        path_cost = {_: intial_values for _ in self.vertices}
        previous_node = {
        _: None for _ in self.vertices
        }
        path_cost[start] = 0
        vertex_all = self.vertices.copy()
        while vertex_all:
            present_node = min(
                vertex_all, key=lambda vertex: path_cost[vertex])
            vertex_all.remove(present_node)
            if path_cost[present_node] == intial_values:
                break
            for adjacent_node, cost_per in self.neighbours[present_node]:
                alternate_path = path_cost[present_node] + int(cost_per)
                if alternate_path < path_cost[adjacent_node]:
```

```
                path_cost[adjacent_node] = alternate_path
                previous_node[adjacent_node] = present_node
        path, present_node = deque(), destination
        while previous_node[present_node] is not None:
            path.appendleft(present_node)
            present_node = previous_node[present_node]
        if path:
            path.appendleft(present_node)
        return path,path_cost
    """
    THIS IS FUNCTION IS USED TO CREATE THE CONNECTION BETWEEN THE TWO
VERTICES
    """
    def create_connection_vertices(self, vertex1, vertex2, path_cost=1, both_ends=True):
        node_pairs = self.getter_adjacent_nodes(vertex1, vertex2, both_ends)
        for edge in self.edges:
            if [edge.start, edge.end] in node_pairs:
                return ValueError('Edge {} {} already exists'.format(vertex1, vertex2))
            self.edges.append(edge(start=vertex1, end=vertex2, cost=path_cost))
            if both_ends:
                self.edges.append(edge(start=vertex2, end=vertex1, cost=path_cost))


    """
    THIS TWO ARE THE PROPERTIES FOR THE GRAPH
    -->WHERE THE VERICES REPRESENT THE EACH VERTEX PRESENT IN THE
    GRAPH.
    -->AND THE NEIGHBOURS IS USED TO FIND THE ADJACENT NODE IN THE GRAPH.
    """
    @property
    def vertices(self):
        return set(
                sum(
            ([edge.start, edge.end] for edge in self.edges), []) )
    @property
    def neighbours(self):
        neighbours = {vertex: set() for vertex in self.vertices}
        for edge in self.edges:
            neighbours[edge.start].add((edge.end, edge.cost))
        return neighbours
    """
```

```
    HIS FUNCTION IS USED TO GET THE VERTICES
    THAT CONNECTED OR THERE IS CONNECTION BETWEEN VERTICE
    """
    def getter_adjacent_nodes(self, vertex1, vertex2, bidirectional=True):
        if bidirectional:
            adjacent_nodes= [[vertex1, vertex2], [vertex2, vertex1]]
        else:
            adjacent_nodes = [[vertex1, vertex2]]
        return adjacent_nodes
    """
    THIS FUNCTION IS USED TO REMOVE THE EDGES FROM THE GIVEN GRAPH
    IF THERE IS AN ALTERNATE PATH THAT IS USED TO CONNECT THE VERTEX WITH
LESSER COST
    """
    def delete_edge(self, n1, n2, both_ends=True):
        adjacent_nodes = self.getter_adjacent_nodes(n1, n2, both_ends)
        edges = self.edges[:]
        for _ in edges:
            if [_.start, _.end] in adjacent_nodes:
                self.edges.remove(_)
```

Implementation of Dijkstra's UnDirected Graph:

```
from collections import defaultdict
from heapq import *


def dijkstra_undirected(edges, source,destination):
    graph = defaultdict(list)
    for a,b,c in edges:
        graph[a].append((c,b))
        start, visited, mins = [(0,source,())], set(), {source: 0}
    while start:
        (path_cost,vertex1,path) = heappop(start)
        if vertex1 not in visited:
            path = (vertex1, path)
            visited.add(vertex1)
            print(path,path_cost)
            if vertex1 == destination:
                return ( path_cost, path )
            for _ , vertex2 in graph.get(vertex1, ()):
                if vertex2 in visited:
```

```
                continue
            previous_node = mins.get(vertex2, None)
            next_node = path_cost + int( _ )
            if previous_node is None or next_node < previous_node:
                mins[vertex2] = next_node
                heappush(start, (next_node, vertex2, path))
    return (dijkstra_undirected(edges,destination,source))
```

Main Program Implementing Dijkstra's directed and undirected graphs:

```
from dijkstra import Graph
from dijkstra_undirected import dijkstra_undirected
def intialise(vertices_from_file):
    length_of_input= len(vertices_from_file)
    graph = vertices_from_file[1:length_of_input-1]
    total_edges = int(vertices_from_file[0][1])
    total_vertices =int( vertices_from_file[0][0])
    type_of_graph = str(vertices_from_file[0][2])
    return length_of_input,graph,total_edges,total_vertices,type_of_graph

def main():

    l =[]
    input_file = open(r'input.txt','r')
    path_distance = ()
    vertices_from_file = []
    var =[]
    for _ in input_file.readlines():
        k=_.split()
        vertices_from_file.append(k)
    length_of_input,graph,total_edges,total_vertices,type_of_graph = intialise(vertices_from_file)
    if(type_of_graph == 'D'):
        for _ in range(1,total_edges+1):
            if(vertices_from_file[_][0] not in l):
                l.append(vertices_from_file[_][0])
        print("THE TOTAL NUMBER OF VERTICES::",total_vertices)
        print("THE TOTAL NUMBER OF EDGES::",total_edges)
        graph = Graph(graph)
        print("THE PATH FROM SOURCE TO EVERY OTHER NODE IN THE DIRECTED
GRAPH\n")
        for _ in range(1,len(l)):
```

```
            result =graph.implementation_of_dijkstra("A",l[_])
            print(result[0])
        print ("THE SHORTEST PATH FROM A TO EVERY OTHER NODE IN THE DIRECTED GRAPH\n ")
        print(path_distance)


    """
    ------------------------------
    THIS IS FOR THE UNDIRECTED GRAPH
    ------------------------------
    """
    temp_list = []
    input_file = open(r'input2.txt','r')
    for _ in input_file.readlines():
        k=_.split()
        temp_list.append(k)
    var= []
    length_of_input,graph,total_edges,total_vertices,type_of_graph = intialise(temp_list)
    edges=temp_list[1:length_of_input-1]
    print("THE TOTAL NUMBER OF VERTICES:",total_vertices)
    print("THE TOTAL NUMBER OF EDGES:",total_edges)
    print ("DIJKSTRA'S UNDIRECTED GRAPH ALGORITHM")
    print("The path traversal is:")

    path = (dijkstra_undirected(edges, "E", "A"))
    #out2 = (dijkstra(edges, "A", "D"))
    print(path)
if __name__ == "__main__":
    start_time = datetime.now()
    main()
    print("Program Runtime: %s seconds" % (datetime.now() - start_time))
```

Sample Inputs For the Directed Graph:

Sample 1:Input_DGraph1.txt

Input:

```
1 9 16 D
2 A B 2
3 A C 1
4 B D 4
5 B E 2
6 C D 2
7 C E 4
8 D F 3
9 D G 8
10 E F 2
11 E G 1
12 F G 4
13 F H 3
14 F I 8
15 G H 2
16 G I 1
17 H I 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_dijkstra's.py"
THE TOTAL NUMBER OF VERTICES:: 9
THE TOTAL NUMBER OF EDGES:: 16
THE PATH FROM SOURCE(A) TO EVERY OTHER NODE IN THE DIRECTED GRAPH

deque(['A', 'B'])
deque(['A', 'C'])
deque(['A', 'C', 'D'])
deque(['A', 'B', 'E'])
deque(['A', 'C', 'D', 'F'])
deque(['A', 'B', 'E', 'G'])
deque(['A', 'B', 'E', 'G', 'H'])
THE SHORTEST PATH COST FROM A TO EVERY OTHER NODE IN THE DIRECTED GRAPH

{'C': 1, 'H': 7, 'B': 2, 'D': 3, 'I': 6, 'A': 0, 'E': 4, 'G': 5, 'F': 6}
THE RUNNING TIME OF THE DIRECTED DIJKSTRA ALGORITHM 0.0012416839599609375 seconds
```

Sample 2:Input_DGraph2.txt

Input:

```
1 9 18 D
2 A B 1
3 A C 2
4 B C 3
5 B D 1
6 B E 2
7 C D 2
8 C E 2
9 D E 3
10 D F 3
11 D G 1
12 E F 2
13 E G 1
14 F G 1
15 F H 2
16 F I 3
17 G H 4
18 G I 8
19 H I 6
```

Output :

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_dijkstra's.py"
THE TOTAL NUMBER OF VERTICES:: 9
THE TOTAL NUMBER OF EDGES:: 18
THE PATH FROM SOURCE(A) TO EVERY OTHER NODE IN THE DIRECTED GRAPH

deque(['A', 'B'])
deque(['A', 'C'])
deque(['A', 'B', 'D'])
deque(['A', 'B', 'E'])
deque(['A', 'B', 'D', 'F'])
deque(['A', 'B', 'D', 'G'])
deque(['A', 'B', 'D', 'G', 'H'])
THE SHORTEST PATH COST FROM A TO EVERY OTHER NODE IN THE DIRECTED GRAPH

{'C': 2, 'F': 5, 'B': 1, 'H': 7, 'A': 0, 'D': 2, 'E': 3, 'G': 3, 'I': 8}
THE RUNNING TIME OF THE DIRECTED DIJKSTRA ALGORITHM 0.0013842582702636719 seconds
```

Sample 3:Input_DGraph3.txt

Input:

```
10 20 D
A B 1
A C 4
B C 1
B D 2
B E 2
C D 3
C E 4
D E 1
D F 1
D G 2
E F 3
E G 1
F G 2
F H 2
F I 1
G H 1
G I 2
H I 3
H J 2
I J 8
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_dijkstra's.py"
THE TOTAL NUMBER OF VERTICES:: 10
THE TOTAL NUMBER OF EDGES:: 20
THE PATH FROM SOURCE(A) TO EVERY OTHER NODE IN THE DIRECTED GRAPH

deque(['A', 'B'])
deque(['A', 'B', 'C'])
deque(['A', 'B', 'D'])
deque(['A', 'B', 'E'])
deque(['A', 'B', 'D', 'F'])
deque(['A', 'B', 'E', 'G'])
deque(['A', 'B', 'E', 'G', 'H'])
deque(['A', 'B', 'D', 'F', 'I'])
THE SHORTEST PATH COST FROM A TO EVERY OTHER NODE IN THE DIRECTED GRAPH

{'E': 3, 'J': 7, 'H': 5, 'I': 5, 'C': 2, 'G': 4, 'B': 1, 'F': 4, 'A': 0, 'D': 3}
```

Sample 4:Input_DGraph4.txt

Input:

```
10 20 D
A B 4
A J 3
B C 11
B D 9
C A 8
C J 4
D C 7
D E 2
D F 6
E B 8
E G 7
E H 4
F C 1
F E 5
G H 14
G I 9
H F 2
H I 10
J F 6
J H 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_dijkstra's.py"
THE TOTAL NUMBER OF VERTICES:: 10
THE TOTAL NUMBER OF EDGES:: 20
THE PATH FROM SOURCE(A) TO EVERY OTHER NODE IN THE DIRECTED GRAPH

deque(['A', 'B'])
deque(['A', 'J', 'F', 'C'])
deque(['A', 'B', 'D'])
deque(['A', 'J', 'F', 'E'])
deque(['A', 'J', 'F'])
deque(['A', 'J', 'F', 'E', 'G'])
deque(['A', 'J', 'F', 'E', 'H'])
deque(['A', 'J'])
THE SHORTEST PATH COST FROM A TO EVERY OTHER NODE IN THE DIRECTED GRAPH

{'B': 4, 'A': 0, 'F': 9, 'H': 18, 'G': 21, 'J': 3, 'D': 13, 'I': 28, 'E': 14, 'C': 10}
THE RUNNING TIME OF THE DIRECTED DIJKSTRA ALGORITHM 0.0015366077423095703 seconds
```

EXAMPLES FOR THE UNDIRECTED GRAPH:

Sample Inputs/Outputs:

Input1_UDGraph.txt

Input:

```
1 9 12 U
2 A B 1
3 A C 2
4 B C 1
5 B D 2
6 C D 3
7 D E 1
8 D F 1
9 E F 2
10 F G 4
11 G H 8
12 H I 6
13 I G 2
```

Output:

```
THE TOTAL NUMBER OF VERTICES: 9
THE TOTAL NUMBER OF EDGES: 12
DIJKSTRA'S UNDIRECTED GRAPH ALGORITHM
The path traversal is:
(4, ('E', ('D', ('B', ('A', ()))))))
THE RUNNING TIME OF THE UNDIRECTED_DIJKSTRA ALGORITHM 3.2901763916015625e-05 seconds
```

Input:Input2_UDGraph.txt

```
1 10 14 U
2 A B 3
3 A C 1
4 B C 2
5 B D 3
6 C D 4
7 D E 2
8 E F 2
9 F G 4
10 F H 1
11 G H 3
12 H I 4
13 I J 2
14 J F 2
15 H E 3
```

Output:

```
THE TOTAL NUMBER OF VERTICES: 10
THE TOTAL NUMBER OF EDGES: 14
DIJKSTRA'S UNDIRECTED GRAPH ALGORITHM
The path traversal is:
(7, ('E', ('D', ('C', ('A', ()))))))
THE RUNNING TIME OF THE UNDIRECTED_DIJKSTRA ALGORITHM 3.886222839355469e-05 seconds
```

Input:Input3_UDGraph.txt

```
9 12 U
A B 2
A C 3
A D 1
B C 4
B D 3
C D 2
D E 1
E F 1
F G 3
G H 1
H I 2
I G 3
```

Output:

```
THE TOTAL NUMBER OF VERTICES: 9
THE TOTAL NUMBER OF EDGES: 12
DIJKSTRA'S UNDIRECTED GRAPH ALGORITHM
The path traversal is:
(2, ('E', ('D', ('A', ()))))
THE RUNNING TIME OF THE UNDIRECTED_DIJKSTRA ALGORITHM 3.3855438232421875e-05 seconds
```

Input:Input4_UDGraph.txt

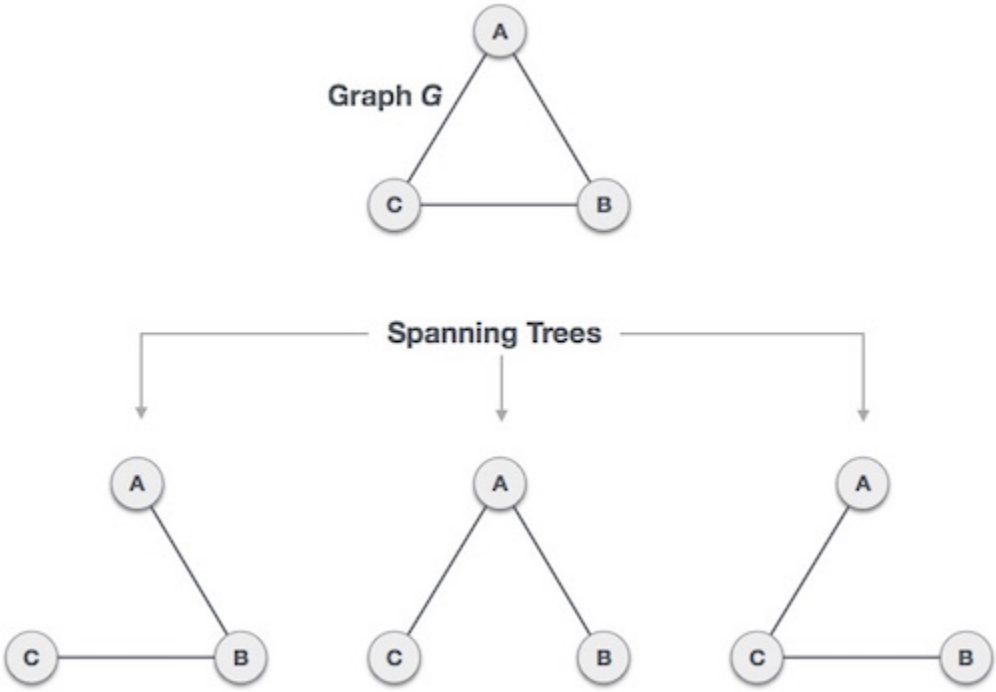
```
10 14 U
A B 1
A C 2
B C 4
B D 1
B E 2
C D 3
D E 1
E F 1
F G 2
G H 4
G I 3
H I 2
I J 1
J D 2
```

Output:

```
THE TOTAL NUMBER OF VERTICES: 10
THE TOTAL NUMBER OF EDGES: 14
DIJKSTRA'S UNDIRECTED GRAPH ALGORITHM
The path traversal is:
(3, ('E', ('B', ('A', ()))))
THE RUNNING TIME OF THE UNDIRECTED_DIJKSTRA ALGORITHM 3.0040740966796875e-05 seconds
```

Problem 2: Minimum Spanning Tree

A spanning tree of a graph is defined as a subgraph containing all the vertices of the graph and the minimum number of edges required to keep the graph connected. In a weighted graph, a minimum spanning tree is the spanning tree which has the minimum weight for the edges included than all other spanning trees of the graph. This project uses Prim's algorithm for calculating the minimum spanning tree.



Prim's Algorithm:

• The minimum_spanning_tree_prim calculates the minimum spanning tree using Python's heap library to maintain a priority queue prioritized by the edge weight of the vertex from the vertices already visited.

• The edges from the source vertex are first heapified.

• The heap prioritizes the outgoing edges. Then the edges are traversed, and if the end vertex obtained from heap pop has not been visited yet, it is added to the minimum spanning tree.

• The neighbors of this vertex are then checked to be added to the heap unless they have already been visited. Thus the runtime is O(m log n).

Data Structures Used and Analysis of Runtime:

• The data structures used to implement Prim's algorithm for finding the minimum spanning tree are Array List, Adjacency List, Graphs, Hash Map, Priority Queue, Tree Set.

• In Prim's algorithm, we used a binary heap for priority queue. This takes O (m log n) time where m is the number of edges and n is the number of vertices.

Pseudocode:

```
ReachSet = {0};
  UnReachSet = {1, 2, ..., N-1};
  SpanningTree = {};

  while ( UnReachSet ≠ empty )
  {
    Find edge e = (x, y) such that:
      1. x ∈ ReachSet
        2. y ∈ UnReachSet
        3. e has smallest cost

    SpanningTree = SpanningTree ∪ {e};

    ReachSet   = ReachSet ∪ {y};
    UnReachSet = UnReachSet - {y};
  }
```

Implementation of Prim's Minimum Spanning Tree algorithm:

Function:

```
from collections import defaultdict
from heapq import *

"""
PRIMS ALGORITHM IS IMPLEMENTED IN THE BELOW FUNCTION
WHICH IS USED TO CONSTRUCT THE MINIMUM SPANNING TREE
"""
def Algorithm_prims( nodes, edges ):
    vertices = defaultdict( list )
    minimum_spanning_tree = []
    for a1,a2,curr in edges:
        vertices[ a2 ].append( (curr, a2, a1) )
        vertices[ a1 ].append( (curr, a1, a2) )


    not_visited = vertices[ nodes[0] ][:]
    used = set( nodes[ 0 ] )
    heapify( not_visited )
    while not_visited:
```

```
        path_cost, a1, a2 = heappop( not_visited )
        if a2 not in used:
            used.add( a2 )
            minimum_spanning_tree.append( ( a1, a2, path_cost ) )
            for _ in vertices[ a2 ]:
                if _[ 2 ] not in used:
                    heappush( not_visited, _ )
    return minimum_spanning_tree
```

Main file:

```
from prims_algo import Algorithm_prims
from datetime import datetime
"""
READING THE CONNECTIONS BETWEEN THE EDGES FROM THE " INPUT FILE" IN TEXT
FORMAT.
"""


def main():
    input_file = open(r'input.txt','r')
    temp = []
    vertices =[]
    vertices_lable = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    vertices_lable =list(vertices_lable)
    path_cost = 0
    #READING THE DATA PRESENT THE GIVING
    for _ in input_file.readlines():
        temp_var=_.split()
        temp.append(temp_var)


    """
    FROM THE INPUT GIVEN WE ARE FINDING THE NUMBER OF VERTICES PRESENT
    IN THE GIVEN GRAPH AND VERTICES AND THE EDGES PRESENT BETWEEN.
    """
    length = len(temp)
    total_vertices =int(temp[0][0])
    graph_type = str(temp[0][2])
    total_edges = int(temp[0][1])
    edges = temp[1:length-1]
    for t in temp:
        print(t)
```

```
    """
    IF THE GRAPH IS AN UNDIRECTED GRAPH.
    """
    if(graph_type == 'U'):
        print("THE MINIMUM SPANNING TREE FOUND FOR AN UNDIRECTED GRAPH IS
FOUND USING THE PRIM'S ALGORITHM")
        print("THE TOTAL NUMBER OF VERTICES:", total_vertices)
        print("THE TOTAL NUMBER OF EDGES :", total_edges)


    for K in range(0,total_vertices):
        vertices.append(vertices_lable[K])


    result= Algorithm_prims( vertices, edges )
    l= 0


    """
    PRINTING THE MINIMUM SPANNING TREE THAT HAS BEEN FORMED
    """


    if(graph_type == 'U'):
        graph="Undirected Graph"
    else:
        graph = "Directed Graph"
    print()
    print("Printing th minimum spanning tree that has been formed for a "+graph)
    for _ in result:
        print ("FROM:'{from_ver}' ~~> TO:'{to}' COST:: {path_cost}".format( from_ver = _[0], to =
_[1], path_cost = _[2]))
        path_cost += int(result[l][2])
        l +=1
    print("TOTAL PATH COST IS :: " ,path_cost)


if __name__ == "__main__":
    start_time = datetime.now()
    main()
    print("Program Runtime: %s seconds" % (datetime.now() - start_time))
```

Sample Inputs For UnDirected Graph:

Samples Input1:Input1_UDGraph.txt

```
 1 9 12 U
 2 A B 1
 3 A C 2
 4 B C 1
 5 B D 2
 6 C D 3
 7 D E 1
 8 D F 1
 9 E F 2
10 F G 4
11 G H 8
12 H I 6
13 I G 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['9', '16', 'D'], ['A', 'B', '2'], ['A', 'C', '1'], ['B', 'D', '4'], ['B', 'E', '2'], ['C', 'D', '2'], ['C', 'E', '4'], ['D', 'F', '3'],
['D', 'G', '8'], ['E', 'F', '2'], ['E', 'G', '1'], ['F', 'G', '4'], ['F', 'H', '3'], ['F', 'I', '8'], ['G', 'H', '2'], ['G', 'I', '1'], ['
H', 'I', '2']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'C' COST:: 1
FROM:'A' --> TO:'B' COST:: 2
FROM:'B' --> TO:'E' COST:: 2
FROM:'E' --> TO:'G' COST:: 1
FROM:'G' --> TO:'I' COST:: 1
FROM:'C' --> TO:'D' COST:: 2
FROM:'E' --> TO:'F' COST:: 2
FROM:'G' --> TO:'H' COST:: 2
TOTAL PATH COST IS ::  13
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0001842975616455078 seconds
```

Sample Input2:Input2_UDGraph.txt

```
 1 10 14 U
 2 A B 3
 3 A C 1
 4 B C 2
 5 B D 3
 6 C D 4
 7 D E 2
 8 E F 2
 9 F G 4
10 F H 1
11 G H 3
12 H I 4
13 I J 2
14 J F 2
15 H E 3
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['9', '18', 'D'], ['A', 'B', '1'], ['A', 'C', '2'], ['B', 'C', '3'], ['B', 'D', '1'], ['B', 'E', '2'], ['C', 'D', '2'], ['C', 'E', '2'],
['D', 'E', '3'], ['D', 'F', '3'], ['D', 'G', '1'], ['E', 'F', '2'], ['E', 'G', '1'], ['F', 'G', '1'], ['F', 'H', '2'], ['F', 'I', '3'], ['
G', 'H', '4'], ['G', 'I', '8'], ['H', 'I', '6']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'B' COST:: 1
FROM:'B' --> TO:'D' COST:: 1
FROM:'D' --> TO:'G' COST:: 1
FROM:'G' --> TO:'E' COST:: 1
FROM:'G' --> TO:'F' COST:: 1
FROM:'A' --> TO:'C' COST:: 2
FROM:'F' --> TO:'H' COST:: 2
FROM:'F' --> TO:'I' COST:: 3
TOTAL PATH COST IS ::  12
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0001697540283203125 seconds
```

Sample Input3:Input3_UDGraph.txt

```
9 12 U
A B 2
A C 3
A D 1
B C 4
B D 3
C D 2
D E 1
E F 1
F G 3
G H 1
H I 2
I G 3
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['10', '20', 'D'], ['A', 'B', '1'], ['A', 'C', '4'], ['B', 'C', '1'], ['B', 'D', '2'], ['B', 'E', '2'], ['C', 'D', '3'], ['C', 'E', '4'],
 ['D', 'E', '1'], ['D', 'F', '1'], ['D', 'G', '2'], ['E', 'F', '3'], ['E', 'G', '1'], ['F', 'G', '2'], ['F', 'H', '2'], ['F', 'I', '1'], [
'G', 'H', '1'], ['G', 'I', '2'], ['H', 'I', '3'], ['H', 'J', '2'], ['I', 'J', '8']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'B' COST:: 1
FROM:'B' --> TO:'C' COST:: 1
FROM:'B' --> TO:'D' COST:: 2
FROM:'D' --> TO:'E' COST:: 1
FROM:'D' --> TO:'F' COST:: 1
FROM:'E' --> TO:'G' COST:: 1
FROM:'F' --> TO:'I' COST:: 1
FROM:'G' --> TO:'H' COST:: 1
FROM:'H' --> TO:'J' COST:: 2
TOTAL PATH COST IS ::  11
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0002942085266113281 seconds
```

Sample Input4:Input4_UDGraph.txt

```
10 14 U
A B 1
A C 2
B C 4
B D 1
B E 2
C D 3
D E 1
E F 1
F G 2
G H 4
G I 3
H I 2
I J 1
J D 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['10', '20', 'D'], ['A', 'B', '4'], ['A', 'J', '3'], ['B', 'C', '11'], ['B', 'D', '9'], ['C', 'A', '8'], ['C', 'J', '4'], ['D', 'C', '7']
, ['D', 'E', '2'], ['D', 'F', '6'], ['E', 'B', '8'], ['E', 'G', '7'], ['E', 'H', '4'], ['F', 'C', '1'], ['F', 'E', '5'], ['G', 'H', '14'],
 ['G', 'I', '9'], ['H', 'F', '2'], ['H', 'I', '10'], ['J', 'F', '6'], ['J', 'H', '2']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'J' COST:: 3
FROM:'A' --> TO:'B' COST:: 4
FROM:'B' --> TO:'C' COST:: 11
FROM:'C' --> TO:'F' COST:: 1
FROM:'F' --> TO:'H' COST:: 2
FROM:'H' --> TO:'I' COST:: 10
FROM:'H' --> TO:'G' COST:: 14
FROM:'H' --> TO:'E' COST:: 4
FROM:'E' --> TO:'D' COST:: 2
TOTAL PATH COST IS ::  51
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0002627372741699219 seconds
```

Sample Input for the Directed Graph:

Input:Input_DGraph1.txt

```
9 16 D
A B 2
A C 1
B D 4
B E 2
C D 2
C E 4
D F 3
D G 8
E F 2
E G 1
F G 4
F H 3
F I 8
G H 2
G I 1
H I 2
```

Ouput:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['9', '16', 'D'], ['A', 'B', '2'], ['A', 'C', '1'], ['B', 'D', '4'], ['B', 'E', '2'], ['C', 'D', '2'], ['C', 'E', '4'], ['D', 'F', '3'],
['D', 'G', '8'], ['E', 'F', '2'], ['E', 'G', '1'], ['F', 'G', '4'], ['F', 'H', '3'], ['F', 'I', '8'], ['G', 'H', '2'], ['G', 'I', '1'], ['
H', 'I', '2']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'C' COST:: 1
FROM:'A' --> TO:'B' COST:: 2
FROM:'B' --> TO:'E' COST:: 2
FROM:'E' --> TO:'G' COST:: 1
FROM:'G' --> TO:'I' COST:: 1
FROM:'C' --> TO:'D' COST:: 2
FROM:'E' --> TO:'F' COST:: 2
FROM:'G' --> TO:'H' COST:: 2
TOTAL PATH COST IS ::  13
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0001842975616455078 seconds
```

Input2:Input_DGraph2.txt

```
9 18 D
A B 1
A C 2
B C 3
B D 1
B E 2
C D 2
C E 2
D E 3
D F 3
D G 1
E F 2
E G 1
F G 1
F H 2
F I 3
G H 4
G I 8
H I 6
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['9', '18', 'D'], ['A', 'B', '1'], ['A', 'C', '2'], ['B', 'C', '3'], ['B', 'D', '1'], ['B', 'E', '2'], ['C', 'D', '2'], ['C', 'E', '2'],
['D', 'E', '3'], ['D', 'F', '3'], ['D', 'G', '1'], ['E', 'F', '2'], ['E', 'G', '1'], ['F', 'G', '1'], ['F', 'H', '2'], ['F', 'I', '3'], ['
G', 'H', '4'], ['G', 'I', '8'], ['H', 'I', '6']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'B' COST:: 1
FROM:'B' --> TO:'D' COST:: 1
FROM:'D' --> TO:'G' COST:: 1
FROM:'G' --> TO:'E' COST:: 1
FROM:'G' --> TO:'F' COST:: 1
FROM:'A' --> TO:'C' COST:: 2
FROM:'F' --> TO:'H' COST:: 2
FROM:'F' --> TO:'I' COST:: 3
TOTAL PATH COST IS ::  12
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0001697540283203125 seconds
```

Input:Input_DGraph3.txt

```
10 20 D
A B 1
A C 4
B C 1
B D 2
B E 2
C D 3
C E 4
D E 1
D F 1
D G 2
E F 3
E G 1
F G 2
F H 2
F I 1
G H 1
G I 2
H I 3
H J 2
I J 8
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['10', '20', 'D'], ['A', 'B', '1'], ['A', 'C', '4'], ['B', 'C', '1'], ['B', 'D', '2'], ['B', 'E', '2'], ['C', 'D', '3'], ['C', 'E', '4'],
 ['D', 'E', '1'], ['D', 'F', '1'], ['D', 'G', '2'], ['E', 'F', '3'], ['E', 'G', '1'], ['F', 'G', '2'], ['F', 'H', '2'], ['F', 'I', '1'], [
'G', 'H', '1'], ['G', 'I', '2'], ['H', 'I', '3'], ['H', 'J', '2'], ['I', 'J', '8']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'B' COST:: 1
FROM:'B' --> TO:'C' COST:: 1
FROM:'B' --> TO:'D' COST:: 2
FROM:'D' --> TO:'E' COST:: 1
FROM:'D' --> TO:'F' COST:: 1
FROM:'E' --> TO:'G' COST:: 1
FROM:'F' --> TO:'I' COST:: 1
FROM:'G' --> TO:'H' COST:: 1
FROM:'H' --> TO:'J' COST:: 2
TOTAL PATH COST IS ::  11
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0002942085266113281 seconds
```

Input:Input_DGraph4.txt

```
10 20 D
A B 4
A J 3
B C 11
B D 9
C A 8
C J 4
D C 7
D E 2
D F 6
E B 8
E G 7
E H 4
F C 1
F E 5
G H 14
G I 9
H F 2
H I 10
J F 6
J H 2
```

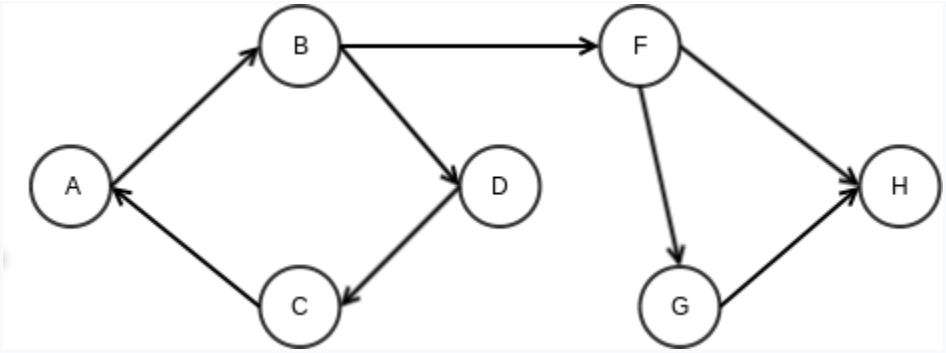
Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_prim's.py"
[['10', '20', 'D'], ['A', 'B', '4'], ['A', 'J', '3'], ['B', 'C', '11'], ['B', 'D', '9'], ['C', 'A', '8'], ['C', 'J', '4'], ['D', 'C', '7']
, ['D', 'E', '2'], ['D', 'F', '6'], ['E', 'B', '8'], ['E', 'G', '7'], ['E', 'H', '4'], ['F', 'C', '1'], ['F', 'E', '5'], ['G', 'H', '14'],
 ['G', 'I', '9'], ['H', 'F', '2'], ['H', 'I', '10'], ['J', 'F', '6'], ['J', 'H', '2']]

Printing th minimum spanning tree that has been formed for a Directed Graph
FROM:'A' --> TO:'J' COST:: 3
FROM:'A' --> TO:'B' COST:: 4
FROM:'B' --> TO:'C' COST:: 11
FROM:'C' --> TO:'F' COST:: 1
FROM:'F' --> TO:'H' COST:: 2
FROM:'H' --> TO:'I' COST:: 10
FROM:'H' --> TO:'G' COST:: 14
FROM:'H' --> TO:'E' COST:: 4
FROM:'E' --> TO:'D' COST:: 2
TOTAL PATH COST IS ::  51
THE RUNNING TIME OF THE PRIMS ALGORITHM IS THAT 0.0002627372741699219 seconds
```

Problem 3: Strongly Connected Components

A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. It is applicable only on a directed graph. In this project Kosaraju's Algorithm is used to check for strongly connected components.



Kosaraju's algorithm

• It is a DFS based algorithm used to find Strongly Connected Components in a graph.
• It is based on the idea that if one is able to reach a vertex v starting from vertex *u*, then one should be able to reach vertex *u* starting from vertex *v*.
• If such is the case, one can say that vertices *u* and *v* are strongly connected - they are in a strongly connected subgraph.

Data Structures Used and Analysis of Runtime:

• The data structures used to implement Prim's algorithm for finding the minimum spanning tree are Array List, Adjacency List, Graphs, Hash Map, Priority Queue, Tree Set.

• In **Kosaraju's algorithm**, we use an adjacency list which has a linear runtime. This takes O (m + n) time where m is the number of edges and n is the number of vertices.

Pseudocode:

```
for each vertex u of the graph do
        u = unvisited
        L = empty
for each vertex u of the graph do,
        Visit(u)
        If u is unvisited then:
                u = visited.
                for each out-neighbour v of u do
                        Visit(v)
                L = u
        Otherwise do nothing.
for each element u of L in order do:
        Assign(u,u)
        If u has not been assigned to a component then:
```

Assign u as belonging to the component whose root is root.

For each in-neighbour v of u, do:

Assign(v,root).

Otherwise do nothing.

IMPLEMENTATION OF KOSARAJU'S ALGORITHM:

Function:

```
from collections import defaultdict


class Graph:
    def __init__(self,vertex):
        self.V= vertex
        self.graph = defaultdict(list)
        self.connected = []

    """
    THIS FUNCTION IS USED TO TRANSPOSE THE GRAPH INORDER TO
    FIND WHETHER THE COMPONENTS OF THE GRAPH ARE STILL CONNECTED
    SO THAT E CAN FIND THE STRONGLY CONNECTED COMPONENT OF THE GRAPH.
    """
    def transpose_of_graph(self):
        g = Graph(self.V)
        for _ in self.graph:
            for k in self.graph[_]:
                g.add_edge(k,_)
        return g

    """
    THIS FUNCTION IS FOR IMPLEMENTING THE DPETH-FIRST-SEARCH
    WHICH IS USED TO FIND THE CONNECTED COMPONENT OF THE GRAPH.
    """
    def depth_first_search(self,v1,already_visited):
        already_visited[v1]= True
        print(chr(v1+65),end=" ")
        for _ in self.graph[v1]:
            if already_visited[_]==False:
                self.depth_first_search(_,already_visited)
```

```
    """
    THIS FUNCTION IS USED TO TAKE THE SOURCE VERTEX AND THE DESTINATION VERTEX
    (ANOTHER VERTEX) INORDER TO CREATE AN EDGE BETWEEN.
    """
    def add_edge(self,u,v):
        self.graph[u].append(v)



    ""THIS FUNCTION IS USED TO KEEP TRACK OF THE VERTICES THAT ARE BEING
    VISITED. """
    def vertices_visited_order(self,v1,already_visited, impl_stack):
        already_visited[v1]= True
        for i in self.graph[v1]:
            if already_visited[i]==False:
                self.vertices_visited_order(i, already_visited, impl_stack)
        impl_stack= impl_stack.append(v1)


    """
    THIS FUNCTIN IS USED TO FIND THE STRONGLY CONNECTED COMPONENT OF THE GRAPH
    WHERE THE GRAPH IS REVERSED TO FIND WHETHER THE CONNECTED COMPONENTS ARE STILL
    CONNECTED.AND THE CONNECTION IS DETECTED USING THE DEPTH-FIRST-SEARCH
    """
    def strongly_connected_graph(self):
        already_traversed=[False]*(self.V)
        impl_stack = []
        for _ in range(self.V):
            if already_traversed[_]==False:
                self.vertices_visited_order(_, already_traversed, impl_stack)
        gr = self.transpose_of_graph()
        already_traversed =[False]*(self.V)
        while impl_stack:
            vertex= impl_stack.pop()
            if already_traversed[vertex]==False:
                gr.depth_first_search(vertex, already_traversed)
                print()
```

Main file:

```
from kosaraju_graph import Graph
from datetime import datetime
def main():


    temp = []
    input_file = open(r'input2.txt','r')
    for _ in input_file.readlines():
        temp_var=_.split()
        temp.append(temp_var)
    """
    FROM THE INPUT GIVEN WE ARE FINDING THE NUMBER OF VERTICES PRESENT
    IN THE GIVEN GRAPH AND VERTICES AND THE EDGES PRESENT BETWEEN.
    """
    length = len(temp)
    total_vertices =int(temp[0][0])
    graph_type = str(temp[0][2])
    total_edges = int(temp[0][1])
    edges = temp[1:length-1]
    source = temp[length-1][0]
    print(source)
    # Create a graph given in the above diagram
    g1 = Graph(total_vertices)
    for i in range(1,len(temp)-1):
        g1.add_edge((ord(temp[i][0])%65),(ord(temp[i][1])%65))
    print ("THE BELOW IS THE STRONGLY CONNECTED COMPONENT(VERTEX) OF THE
GRAPH")
    g1.strongly_connected_graph()


    for i in range(0,len(g1.connected)):
        print(g1.connected[i],end=" ")


if __name__ == "__main__":
    start_time = datetime.now()
    main()
    print("Program Runtime: %s seconds" % (datetime.now() - start_time))
```

Sample Inputs:

Sample 1:
INPUT:input_2.txt

```
 1 8 9 D
 2 A B 1
 3 B C 1
 4 C D 1
 5 C E 1
 6 D A 1
 7 E F 1
 8 F G 1
 9 G E 1
10 G H 1
```

:

Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_kosaraju's.py"
THE BELOW IS THE STRONGLY CONNECTED COMPONENT(VERTEX) OF THE GRAPH
THE STRONGLY CONNECTED COMPONENT 1
A D C B
THE STRONGLY CONNECTED COMPONENT 2
E G F
THE STRONGLY CONNECTED COMPONENT 3
H
THE RUNNING TIME OF THE PROGRAM IS 0.0001728534698486328 seconds
```

Sample 2:
Input:

```
 1 10 20 D
 2 A B 4
 3 A J 3
 4 B C 11
 5 B D 9
 6 C A 8
 7 C J 4
 8 D C 7
 9 D E 2
10 D F 6
11 E B 8
12 E G 7
13 E H 4
14 F C 1
15 F E 5
16 G H 14
17 G I 9
18 H F 2
19 H I 10
20 J F 6
21 J H 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_kosaraju's.py"
THE BELOW IS THE STRONGLY CONNECTED COMPONENT(VERTEX) OF THE GRAPH
THE STRONGLY CONNECTED COMPONENT 1
A C B E D F H G J
THE STRONGLY CONNECTED COMPONENT 2
I
THE RUNNING TIME OF THE PROGRAM IS 0.00014448165893554688 seconds
```

Sample 3:

Input:

```
 1 10 20 D
 2 A B 1
 3 A C 4
 4 B C 1
 5 B D 2
 6 B E 2
 7 C D 3
 8 C E 4
 9 D E 1
10 D F 1
11 D G 2
12 E F 3
13 E G 1
14 F G 2
15 F H 2
16 F I 1
17 G H 1
18 G I 2
19 H I 3
20 H J 2
21 I J 8
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_kosaraju's.py"
THE BELOW IS THE STRONGLY CONNECTED COMPONENT(VERTEX) OF THE GRAPH
THE STRONGLY CONNECTED COMPONENT 1
A
THE STRONGLY CONNECTED COMPONENT 2
B
THE STRONGLY CONNECTED COMPONENT 3
C
THE STRONGLY CONNECTED COMPONENT 4
D
THE STRONGLY CONNECTED COMPONENT 5
E
THE STRONGLY CONNECTED COMPONENT 6
F
THE STRONGLY CONNECTED COMPONENT 7
G
THE STRONGLY CONNECTED COMPONENT 8
H
THE STRONGLY CONNECTED COMPONENT 9
I
THE STRONGLY CONNECTED COMPONENT 10
J
THE RUNNING TIME OF THE PROGRAM IS 0.000331878662109375 seconds
```

Sample 4:
Input:

```
 1 9 16 D
 2 A B 2
 3 A C 1
 4 B D 4
 5 B E 2
 6 C D 2
 7 C E 4
 8 D F 3
 9 D G 8
10 E F 2
11 E G 1
12 F G 4
13 F H 3
14 F I 8
15 G H 2
16 G I 1
17 H I 2
```

Output:

```
ram@Ram:~/Desktop/Interview Prep$ /bin/python3 "/home/ram/Desktop/Interview Prep/main_kosaraju's.py"
THE BELOW IS THE STRONGLY CONNECTED COMPONENT(VERTEX) OF THE GRAPH
THE STRONGLY CONNECTED COMPONENT 1
A
THE STRONGLY CONNECTED COMPONENT 2
C
THE STRONGLY CONNECTED COMPONENT 3
B
THE STRONGLY CONNECTED COMPONENT 4
E
THE STRONGLY CONNECTED COMPONENT 5
D
THE STRONGLY CONNECTED COMPONENT 6
F
THE STRONGLY CONNECTED COMPONENT 7
G
THE STRONGLY CONNECTED COMPONENT 8
H
THE STRONGLY CONNECTED COMPONENT 9
I
THE RUNNING TIME OF THE PROGRAM IS 0.0002503395080566406 seconds
```

References:

1.https://www.youtube.com/
2.https://www.geeksforgeeks.org/
3.https://www.meccanismocomplesso.org/en/programming-graphs-in-python-part-1/