19CSE302 Design and Analysis of Algorithms

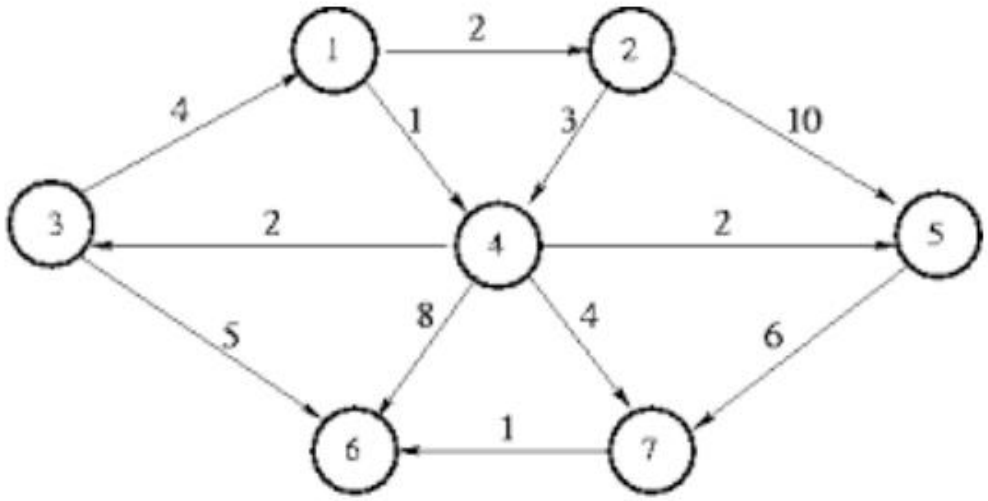
Lab Sheet 7

S Abhishek

AM.EN.U4CSE19147

Colab

Implement the Kosaraju Algorithm to find the SCC




```
graph_edgelist = [[1, 2],
[1, 4],
[2, 4],
[2, 5],
[3, 1],
[3, 6],
[4, 3],
[4, 5],
[4, 6],
[4, 7],
[5, 7],
```

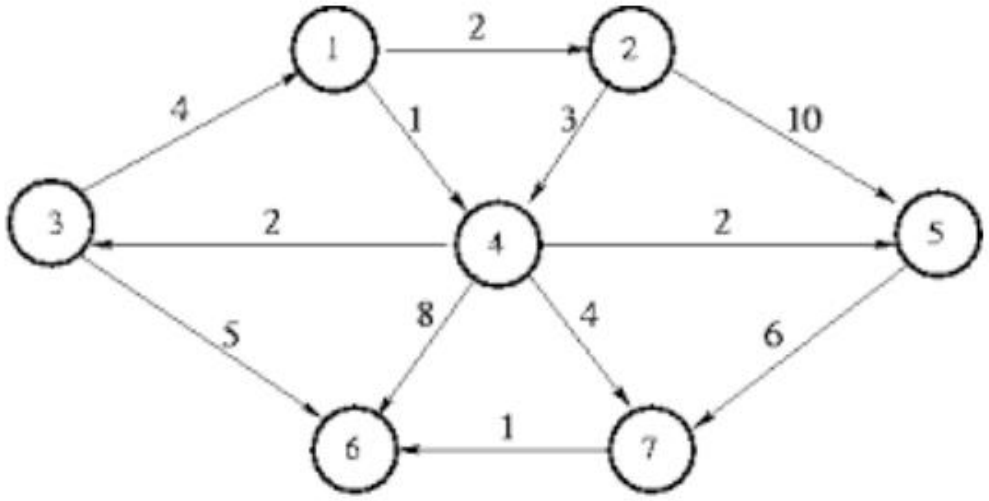
```
[7, 6] ]
class Kosaraju (object):

    def __init__(self, graph_edgelist):

        self.nodelist={}

        for i in range(len(graph_edgelist)):

            if graph_edgelist[i][0] != graph_edgelist[i][1]:

                if (graph_edgelist[i][0] in self.nodelist) &
(graph_edgelist[i][1] in self.nodelist) :

self.nodelist[graph_edgelist[i][0]]['connected_node'].extend([graph_edgelist[
i][1]])

self.nodelist[graph_edgelist[i][1]]['connected_node_reverse'].extend([graph_e
dgelist[i][0]])

                elif graph_edgelist[i][0] in self.nodelist:

self.nodelist[graph_edgelist[i][0]]['connected_node'].extend([graph_edgelist[
i][1]])

self.nodelist[graph_edgelist[i][1]]={'connected_node_reverse':[graph_edgelist
[i][0]],'connected_node':[]}

                elif graph_edgelist[i][1] in self.nodelist:

self.nodelist[graph_edgelist[i][1]]['connected_node_reverse'].extend([graph_e
dgelist[i][0]])

self.nodelist[graph_edgelist[i][0]]={'connected_node':[graph_edgelist[i][1]],
'connected_node_reverse':[]}

                else:

self.nodelist[graph_edgelist[i][1]]={'connected_node_reverse':[graph_edgelist
[i][0]],'connected_node':[]}

self.nodelist[graph_edgelist[i][0]]={'connected_node':[graph_edgelist[i][1]],
'connected_node_reverse':[]}

        for key in self.nodelist:

            self.nodelist[key]['finishing_time']=0
            self.nodelist[key]['explored']=False
```

```
        self.scc={}
    def get_scc_kosaraju(self):

        self.node_loop_stack=[]

        for node in self.nodelist:

            if self.nodelist[node]['explored']==False:
                self.DFS(node,True)


        for key in self.nodelist:

            self.nodelist[key]['leader']=None
            self.nodelist[key]['explored']=False

        while self.node_loop_stack != []:

            node = self.node_loop_stack.pop()

            if self.nodelist[node]['explored']==False:
                self.leader=node
                self.scc[self.leader]={'Path' : []}
                self.DFS(node,False)

    def DFS(self, node, reverse_flag):

        if reverse_flag==True:

            self.nodelist[node]['explored']=True

            for child_node in self.nodelist[node]['connected_node_reverse']:

                if self.nodelist[child_node]['explored']==False:
                    self.DFS(child_node, True)

            self.node_loop_stack.extend([node])

        else:

            self.nodelist[node]['explored']=True

            for child_node in self.nodelist[node]['connected_node']:

                if self.nodelist[child_node]['explored']==False:
                    self.DFS(child_node, False)

            self.scc[self.leader]['Path'].extend([node])
```

```
kosaraju=Kosaraju(graph_edgelist)
kosaraju.get_scc_kosaraju()
kosaraju.scc
```

```
{2: {'Path': [1, 3, 4, 2]},
 5: {'Path': [5]},
 6: {'Path': [6]},
 7: {'Path': [7]}}
```

Implement Dijkstra's Algorithm



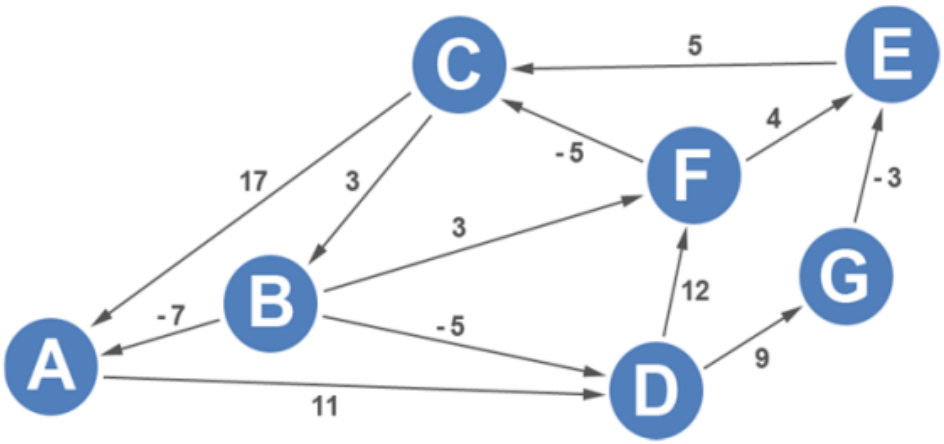

```
from queue import PriorityQueue

class Graph:

    def __init__(self, num_of_vertices):

        self.v = num_of_vertices
        self.edges = [[-1 for i in range(num_of_vertices)] for j in
range(num_of_vertices)]
        self.visited = []

    def add_edge(self, u, v, weight):

        self.edges[u][v] = weight
        self.edges[v][u] = weight

    def dijkstra(self, start_vertex):

        D = {v:float('inf') for v in range(self.v)}
```

```
        D[start_vertex] = 0

        pq = PriorityQueue()
        pq.put((0, start_vertex))

        while not pq.empty():

            (dist, current_vertex) = pq.get()
            self.visited.append(current_vertex)

            for neighbor in range(self.v):

                if self.edges[current_vertex][neighbor] != -1:
                    distance = self.edges[current_vertex][neighbor]

                    if neighbor not in self.visited:

                        old_cost = D[neighbor]
                        new_cost = D[current_vertex] + distance

                        if new_cost < old_cost:

                            pq.put((new_cost, neighbor))
                            D[neighbor] = new_cost

        return D

g = Graph(8)

g.add_edge(1, 2, 2)
g.add_edge(1, 4, 1)
g.add_edge(2, 4, 3)
g.add_edge(2, 5, 10)
g.add_edge(3, 1, 4)
g.add_edge(3, 6, 5)
g.add_edge(4, 3, 2)
g.add_edge(4, 5, 2)
g.add_edge(4, 6, 8)
g.add_edge(4, 7, 4)
g.add_edge(5, 7, 6)
g.add_edge(7, 6, 1)

s = 1

D = g.dijkstra(s)
```

```
for vertex in range(len(D)):

  if vertex == 0:

    D[vertex]

  else:

    print("Distance from vertex", vertex, "to vertex", vertex, "is",
D[vertex])
```

```
Distance from vertex 1 to vertex 1 is 0
Distance from vertex 2 to vertex 2 is 2
Distance from vertex 3 to vertex 3 is 3
Distance from vertex 4 to vertex 4 is 1
Distance from vertex 5 to vertex 5 is 3
Distance from vertex 6 to vertex 6 is 6
Distance from vertex 7 to vertex 7 is 5
```

Bellman - Ford Algorithm




```
def bellman_ford(graph, source):

    distance, predecessor = dict(), dict()

    for node in graph:

        distance[node], predecessor[node] = float('inf'), None
```

```
    distance[source] = 0

    for _ in range(len(graph) - 1):

        for node in graph:

            for neighbour in graph[node]:

                if distance[neighbour] > distance[node] +
graph[node][neighbour]:

                    distance[neighbour], predecessor[neighbour] =
distance[node] + graph[node][neighbour], node

    for node in graph:

        for neighbour in graph[node]:

            assert distance[neighbour] <= distance[node] +
graph[node][neighbour], "Negative weight cycle."

    return distance, predecessor


if __name__ == '__main__':

    graph = {

        'a': {'d': 11},
        'b': {'a': -7, 'd': -5, 'f': 3},
        'c': {'a': 17, 'b': 3},
        'd': {'f': 12, 'g': 9},
        'e': {'c': 5},
        'f': {'c': -5, 'e': 4},
        'g': {'e': -3}
    }

    distance, predecessor = bellman_ford(graph, source='a')

    print(distance)
```

```
{'a': 0, 'b': 21, 'c': 18, 'd': 11, 'e': 17, 'f': 23, 'g': 20}
```

Number of Operations to Make Network Connected

Submission Detail

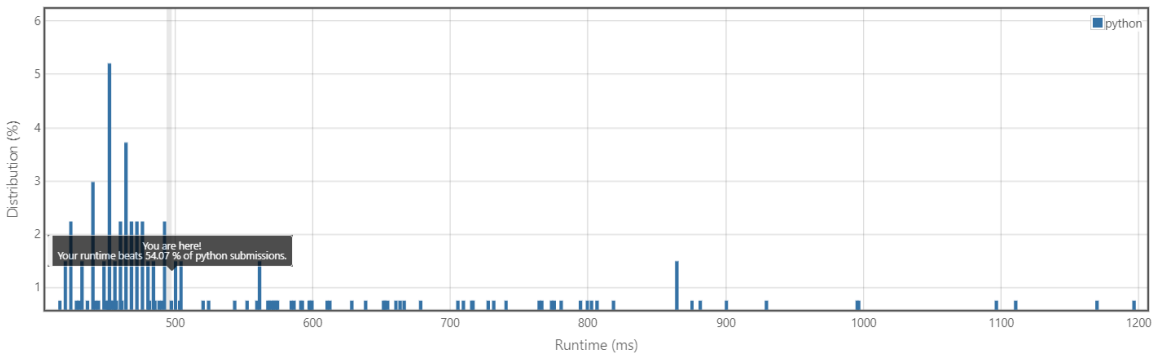
36 / 36 test cases passed.

Runtime: **496 ms**
Memory Usage: **34.9 MB**

Status: Accepted

Submitted: **0 minutes ago**

Accepted Solutions Runtime Distribution



Find if Path Exists in Graph

Submission Detail

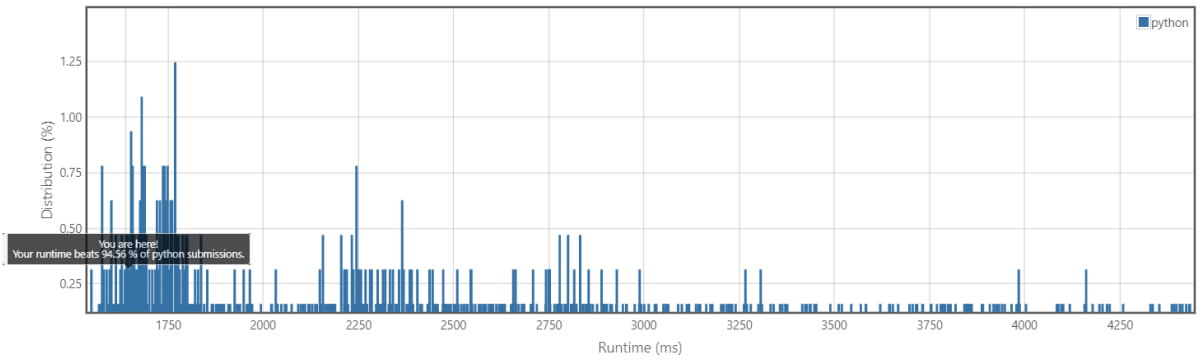
25 / 25 test cases passed.

Runtime: **1640 ms**
Memory Usage: **115.5 MB**

Status: Accepted

Submitted: **0 minutes ago**

Accepted Solutions Runtime Distribution



Thankyou!!