PROJECT-2

GRAPH ALGORITHMS AND RELATED DATA STRUCTURES

Single-Source Shortest Path Algorithm, Minimum Spanning Tree (MST), and Strongly Connected Components (SCCs)

ITCS-6114 SPRING 2022

ALGORITHMS AND DATA STRUCTURES

Submitted by:

Gopichand Tadapaneni

Student Id: 801274664

Student Mail ID: gtadapan@uncc.edu

Kamala Kumari Karuturi

Student Id: 801274525

Student Mail ID: kkarutur@uncc.edu

PROBLEM 1:

SINGLE-SOURCE SHORTEST PATH ALGORITHM:

Using the Single Source Shortest Path algorithm, we can determine the path which is shortest between two vertices in the graph given. Here the weight can either be negative or positive. In real-time, it is used in reserving flight tickets and giving directions while we drive a vehicle, etc. Of all algorithms, we have in order to compute the Single Source Shortest Path algorithm Dijkstra algorithm gives the best-desired output. But negative weights can't be held in the Dijkstra algorithm.

DESCRIPTION OF ALGORITHM:

Let G is the connected weighted graph, and P be the length of the path. The distance of vertex "v" from a vertex "s" is the length of the shortest path between "s" and "v" which is indicated as d(s,v).

ALGORITHM:

INITIALIZE (G, s)

1. $d[s] \leftarrow 0$
2. $p[s] \leftarrow NIL$
3. for all $v \in V-\{s\}$
4. do $d[v] \leftarrow \infty$
5. $p[v] \leftarrow NIL$

DIJKSTRA ALGORITHM:

- This algorithm is used to calculate the distance which is the shortest of all vertices with respect to the source vertex "s" that is given.
- Assumptions made by Dijkstra Algorithm are graph should be connected, edges can either be directed or undirected and weights of the edges shouldn't be negative.
- It has the cloud or set S where the weights of the final shortest path have been already derived from a particular source "s"

- For every vertex “v” we have d(v) which indicates the distance from particular vertex “v” from “s” which contains the cloud and vertices adjacent to it.
- At every step outside the cloud we usually add to the cloud of the vertex “u” with reference to the smallest distance label “d(u)”.
- Labels of vertices that are adjacent to “u” will be updated
- It also keeps a priority queue “Q” with all the vertices in V-S w.r.t the value of d.

ALGORITHM:

DIJKSTRA (G,w,s)

1. INITIALIZE (G,s)
2. s← ∅
3. Q← V
4. while Q=∅
5. do u ← EXTRACT-MIN (Q)
6. s ← s ∪{u}
7. for each vertex v∈ Adj [u]
8. do RELAX (u,v,w)

ANALYSIS OF DIJKSTRA’S ALGORTHM:

- Let the graph be G with “n” being the number of vertices and “m” being the number of edges.
- INITIALIZE(G,s) has the time complexity of “O(n)” as it has a for loop which in turn runs for all the vertices of the graph.
- In this algorithm the priority queue that has been used is min-heap and vertex will be inserted once and the time taken for insertion is O(log n) and with n vertices, the total time taken would be “O(n log n)”.
- At the same time in the EXTRACT-MIN(Q) in order to remove the heap it takes O(log n) will be called for all the vertices and each removal in a heap takes O(log n) time and with “n” vertices total time taken would be O(n log n).

- Similarly, the time taken by RELAX (u,v,w) operation is O(log n), and with respect to “m” edges total time taken would be “O(m log n)”.
- Total time complexity of Dijkstra’s algorithm is O((n+m) log n) which is “O(m log n)”.

DATA STRUCTURES USED:

Data structures used here in order to determine the shortest path are Priority Queue, TreeSet, Graph, Array List, HashMap, Adjacency List.

SOURCE CODE:

```
import sys
from collections import defaultdict
import time


class Graph:
    def __init__(self, dir_graph=False):
        self.graph = defaultdict(list)
        self.dir_graph = dir_graph

    def add_edge(self, frm, to, weight):
        self.graph[frm].append([to, weight])

        if self.dir_graph is False:
            self.graph[to].append([frm, weight])
        elif self.dir_graph is True:
            self.graph[to] = self.graph[to]

    def minimum_f(self, dis, visit):
        minimum = float('inf')
        index = -1
        for v in self.graph.keys():
            if visit[v] is False and dis[v] < minimum:
```

```
            minimum = dis[v]
            index = v

    return index

def Dijkstras(self, source):
    visit = {i: False for i in self.graph}
    dis = {i: float('inf') for i in self.graph}
    P = {i: None for i in self.graph}

    dis[source] = 0

    # find shortest path for all vertices
    for i in range(len(self.graph) - 1):
        u = self.minimum_f(dis, visit)
        visit[u] = True
        for v, weight in self.graph[u]:

            if visit[v] is False and dis[u] + weight < dis[v]:
                dis[v] = dis[u] + weight
                P[v] = u
    return P, dis

def print_path(self, path, v):
    if path[v] is None:
        return
    self.print_path(path, path[v])
    print(chr(v+65), end=" ")

def print_solution(self, dis, P, source):
    print('{}\t\t{}\t{}'.format('Vertex', 'Distance', 'Path'))
```

```
        for i in self.graph.keys():
            if i == source:
                continue
            if dis[i] == float("inf"):
                continue
            print('{} -> {}\t\t{}\t\t{}'.format(chr(source+65),
                  chr(i+65), dis[i], chr(source+65)), end=' ')
            self.print_path(P, i)
            print()

#program for the input
j = 0
print("**********************************************************************")
# graph=None
dir_graph = False
while(j < 4):
    input_file = open("input"+str(j)+".txt", "r")
    i = 0
    source = sys.maxsize
    print()
    print("For the given input file, the shortest paths are: ")
    print()

    for line in input_file.readlines():

        x_line = line.split()
        if i == 0:
            number_of_vertices = int(x_line[0])
            print('Number of Vertices in the graph=', number_of_vertices)
            print('Number of Edges in the graph=', int(x_line[1]))
```

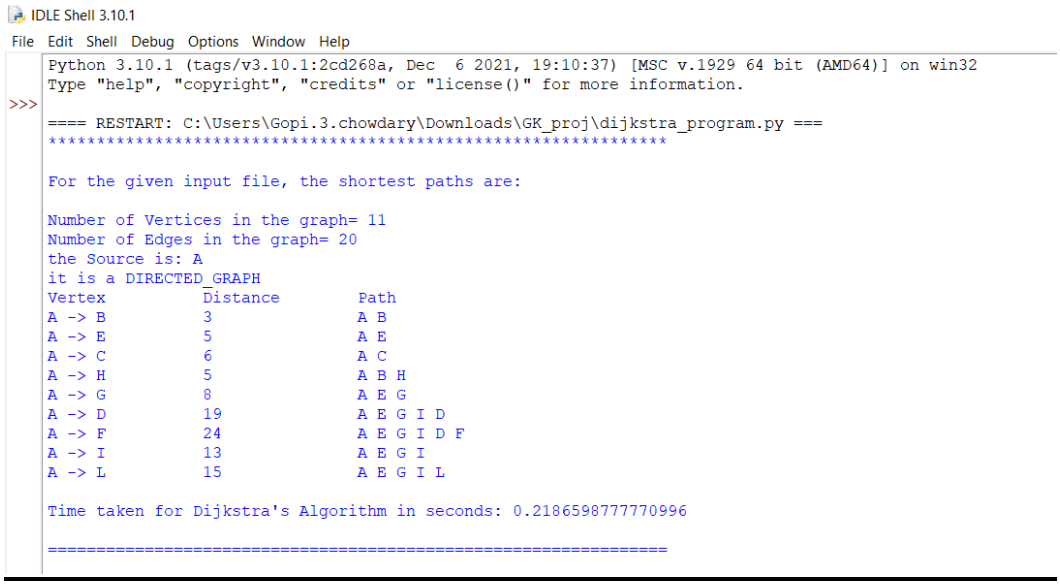
```
            dir = x_line[2]
            if dir == "U":
                dir_graph = False
            else:
                dir_graph = True
            graph = Graph(dir_graph)

        elif len(x_line) == 1:
            source = ord(x_line[0])-65
        else:
            graph.add_edge(ord(x_line[0])-65, ord(x_line[1])-65, int(x_line[2]))
        i = i+1
    print("the Source is:", chr(source+65))
    if dir == "U":

        print("it's an UNDIRECTED_GRAPH")
    elif dir == "D":
        print("it is a DIRECTED_GRAPH")

    started_time = time.time()
    path, distance = graph.Dijkstras(source)
    graph.print_solution(distance, path, source)
    runtime = (time.time() - started_time)
    print()
    print('Time taken for Dijkstra\'s Algorithm in seconds:', runtime)
    print()
    j += 1

print("================================================================
")
    print('\t')
```

SAMPLE INPUT/OUTPUT:

Input 1:

11 20 D
A B 3
A E 5
A C 6
B H 2
C H 4
C G 3
D E 2
D F 5
E G 3
F I 6
F A 2
G B 2
G I 5
H A 3
H E 5
I D 6
I E 4
I H 4
I L 2
L E 3
A

Output 1:



Input 2:

10 23 D
A B 9
A C 4
A F 14
B C 3
B 4 2
C B 4
C D 10
C E 3
D E 3
D G 7
D H 3
E A 6
E D 7
E F 6
E I 5
F I 8
F G 2
G H 5
G E 4
H I 2
H J 4
I G 6
I J 3
A

Output 2:

```
================================================================


For the given input file, the shortest paths are:

Number of Vertices in the graph= 10
Number of Edges in the graph= 23
the Source is: A
it is a DIRECTED_GRAPH
Vertex          Distance        Path
A -> B          8               A C B
A -> C          4               A C
A -> F          13              A C E F
A -> 4          10              A C B 4
A -> D          14              A C D
A -> E          7               A C E
A -> G          15              A C E F G
A -> H          17              A C D H
A -> I          12              A C E I
A -> J          15              A C E I J

Time taken for Dijkstra's Algorithm in seconds: 0.34399914741516113

================================================================
```

Input 3:

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

Output 3:

```
For the given input file, the shortest paths are:

Number of Vertices in the graph= 10
Number of Edges in the graph= 15
the Source is: A
it's an UNDIRECTED_GRAPH
Vertex          Distance        Path
A -> B          4               A B
A -> H          8               A H
A -> C          12              A B C
A -> I          14              A B C I
A -> D          19              A B C D
A -> F          11              A H G F
A -> E          21              A H G F E
A -> G          9               A H G
A -> J          22              A B C I J

Time taken for Dijkstra's Algorithm in seconds: 0.45298266410827637

================================================================
```

Input 4:

10 17 U
A B 8
A C 2
A D 5
B D 2
B F 13
C E 5
C D 2
C I 1
D E 1
D F 6
D G 3
E G 1
F G 2
F H 3
G H 6
I B 4
I J 6
C

Output 4:

```
======================================================================


For the given input file, the shortest paths are:

Number of Vertices in the graph= 10
Number of Edges in the graph= 17
the Source is: C
it's an UNDIRECTED_GRAPH
Vertex          Distance        Path
C -> A          2               C A
C -> B          4               C D B
C -> D          2               C D
C -> F          6               C D E G F
C -> E          3               C D E
C -> I          1               C I
C -> G          4               C D E G
C -> H          9               C D E G F H
C -> J          7               C I J

Time taken for Dijkstra's Algorithm in seconds: 0.4530181884765625

======================================================================

>>>
```

RUNTIME ANALYSIS OF DIJKSTRA'S ALGORITHM:

S.No	Graph Type	Edges	Vertices	Average Runtime (Seconds)
1	Directed	20	11	0.2186598777770996
2	Directed	23	10	0.34399914741516113
3	Undirected	15	10	0.45298266410827637
4	Undirected	17	10	0.4530181884765625

PROBLEM 2:

MINIMUM SPANNING TREE:

For a particular graph, the spanning tree is the one that is linked to some edges in a graph containing all the vertices. A graph consists of one or more spanning trees. If a graph has “N” vertices, the number of edges in the spanning tree can have are “N-1”.

When we collate all the spanning trees the one with the lowest weight in the graph that’s given is called as Minimum Spanning Tree (MST). A minimal spanning tree can be calculated using two techniques i.e., Prim's and Kruskal's algorithms. In this project in order to find the Minimum Spanning Tree (MST) Kruskal's technique is used.

KRUSKAL'S ALGORITHM:

- In order to find the minimum spanning tree for a graph that’s given this algorithm uses a greedy approach.
- In this method, each node is treated as an independent tree and it links to the other only if the cost is less compared to others.

Steps to create a Minimum Spanning tree using Kruskal’s Algorithm:

- Initially each vertex is considered as a tree within the forest that is considered.
- In order to add to the growing forest, it computes safe edge by finding the one with less weight edge (u, v) of all the edges that connect any two trees
- Kruskal’s algorithm is the one that’s known for its greedy approach because at every step it adds an edge with the least weight to the forest.
- In the end it leaves one cloud which is our MST of A tree T.

PSEUDO CODE:

KRUSKAL (G):

1. A =∅

2. for each vertex v ∈ G.V:

3. MAKE-SET(v)

4. Sort the edges of G.E into non-decreasing order by weight w

5. For each edge (u, v) in G.E, taken in non-decreasing order by weight

6. if FIND-SET(u) ≠ FIND-SET(v):

7. A = A ∪ {(u, v)}

8. UNION(u, v)

9. return A

ANALYSIS OF KRUSKAL'S ALGORITHM:

- For a given graph “G” let the number of vertices be “n” and edges be “m”
- Time taken by MAKE-SET(v) is O(n) as it’s called only once by each vertex.
- In order to sort the edges w.r.t weight the time taken is O(m log m).
- Time taken by FIND-SET() is O(m).
- As spanning trees have “N-1” edges UNION(u,v) is called “n-1” times and the time taken is O(n).
- In the end Kruskal’s algorithm takes total runtime of O(m log m).

DATA STRUCTURES USED:

Hashmap ,Graph, TreeSet , ArrayList , Map , are the data Structures is used in this program to find the Minimum Spanning Tree.

SOURCE CODE:

```
import time
class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def path_find(self, path, i):

        if path[i] == i:
            return i
        return self.path_find(path, path[i])

    def union_function(self, path, r, x, y):
        X = self.path_find(path, x)
        Y = self.path_find(path, y)

        if r[X] < r[Y]:
            path[X] = Y
        elif r[X] > r[Y]:
            path[Y] = X

        else:
            path[Y] = X
            r[X] += 1

    def Krushkal_function(self):
```

```
result = []
e = 0
i = 0
self.graph = sorted(self.graph, key=lambda item: item[2])

P = []
r = []

for node in range(self.V):
    P.append(node)
    r.append(0)

while e < self.V - 1:

    u, v, w = self.graph[i]

    i = i + 1
    x = self.path_find(P, u)
    y = self.path_find(P, v)

    if x != y:
        e = e + 1
        result.append([u, v, w])
        self.union_function(P, r, x, y)

print('Edge Selected \t\t Weight')
print()
res = 0
for u, v, weight in result:
    print(chr(u + 65), "-", chr(v + 65), "\t\t\t", weight)
```

```
            res += weight
        print('Minimum Spanning Tree\'s total cost =', res)


# program for the input
print("**********************************************************************************
********")
j = 2
print()
while (j < 6):
    print("For the given graph, Minimum Spanning Tree using Kruskal's Algorithm:")
    print()
    input_file = open("input" + str(j) + ".txt", "r")

    i = 0
    for line in input_file.readlines():
        x = line.split()
        if i == 0:
            number_of_vertices = int(x[0])
            # print("number of vertices=",number_of_vertices)
            graph = Graph(number_of_vertices)
        elif len(x) == 1:
            pass
        else:
            graph.add_edge(ord(x[0]) - 65, ord(x[1]) - 65, int(x[2]))
        i = i + 1
    started_time = time.time()
    graph.Krushkal_function()
    time_taken = (time.time() - started_time)
    print('----------------------------------------------------------------------')
    print('Time taken for running Kruskal Algorithm in seconds=', time_taken)
```

```
    print()
    j += 1

print("=============================================================
=====")
    print('\t')
```

SAMPLE INPUT/OUTPUT:

Input 1:

10 15 U

A B 4

A H 8

B H 11

B C 8

C I 2

C D 7

C F 4

D E 9

D F 14

E F 10

F G 2

G H 1

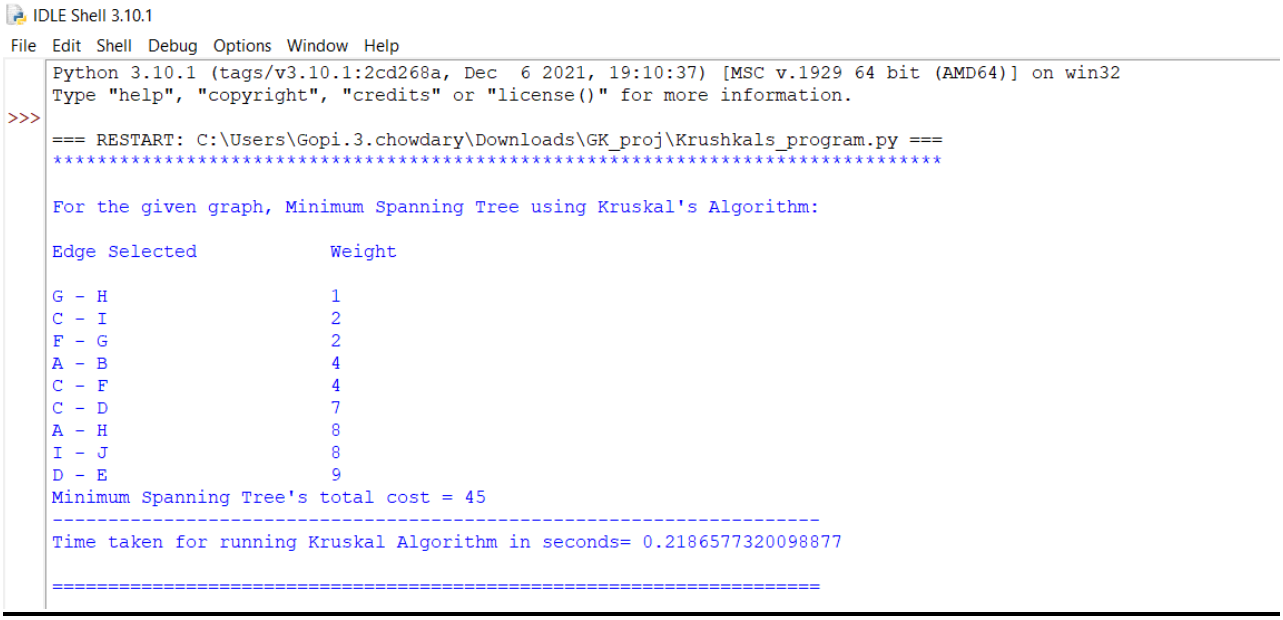
G I 6

H I 7

I J 8

A

Output 1:



Input 2:

10 17 U

A B 8

A C 2

A D 5

B D 2

B F 13

C E 5

C D 2

C I 1

D E 1

D F 6

D G 3

E G 1

F G 2

F H 3

G H 6

I B 4

I J 6

C

Output 2:

```
======================================================================

For the given graph, Minimum Spanning Tree using Kruskal's Algorithm:

Edge Selected            Weight

C - I                    1
D - E                    1
E - G                    1
A - C                    2
B - D                    2
C - D                    2
F - G                    2
F - H                    3
I - J                    6
Minimum Spanning Tree's total cost = 20
----------------------------------------------------------------------
Time taken for running Kruskal Algorithm in seconds= 0.2334156036376953

======================================================================
```

Input 3:

9 18 U

A C 9

A D 13

A B 22

B H 34

B C 35

B F 36

C D 4

C E 70

C F 45

D E 33

D I 40

E F 18

E G 23

F H 24

F G 39

G H 25

G I 20

H I 19

A

Output 3:

```
========================================================================

For the given graph, Minimum Spanning Tree using Kruskal's Algorithm:

Edge Selected           Weight

C - D                   4
A - C                   9
E - F                   18
H - I                   19
G - I                   20
A - B                   22
E - G                   23
D - E                   33
Minimum Spanning Tree's total cost = 148
------------------------------------------------------------------------
Time taken for running Kruskal Algorithm in seconds= 0.3280048370361328

========================================================================
```

Input 4:

9 14 U

A B 4

A C 3

A E 7

B C 6

B D 5

C D 11

C E 8

D E 2

D F 2

D G 10

E G 5

F G 3

H F 2

I H 1

A

Output 4:

```
------------------------------------------------------------------------

For the given graph, Minimum Spanning Tree using Kruskal's Algorithm:

Edge Selected            Weight

I - H                    1
D - E                    2
D - F                    2
H - F                    2
A - C                    3
F - G                    3
A - B                    4
B - D                    5
Minimum Spanning Tree's total cost = 22
------------------------------------------------------------------------
Time taken for running Kruskal Algorithm in seconds= 0.3593268394470215

========================================================================
>
```

RUNTIME ANALYSIS:

S.No	Graph Type	Edges	Vertices	Average Runtime (Seconds)
1	Directed	15	10	0.2186577320098877
2	Directed	17	10	0.2334156036376953
3	Undirected	18	9	0.3280048370361328
4	Undirected	14	9	0.3593268394470215

PROBLEM 3:

STRONGLY CONNECTED COMPONENTS (SCCs):

For a particular directed graph number of connected nodes that are maximum can be calculated with help of Strongly Connected Components (SCC). A component is said to be highly linked when there is a path that is directed within each pair of nodes in a set.

Steps to create strongly connected components:

With the help of Kosaraju's algorithm, we can determine all strongly connected components with time complexity of O(V+E).

Steps for Kosaraju's algorithm:

- Firstly, an empty stack "S" to be created and then DFS traversal of a graph to be done. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in the stack as 1, 2, 4, 3, 0.
- In order to obtain the transpose graph directions of all arcs to be reversed.
- Popping of vertex to be done from S while S is not empty. Let the vertex that is popped be 'v'. Then v will be taken as a source and DFS will be done (call DFSUtil(v)). The DFS is used to print strongly connected components of v by starting from v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

PSEUDO CODE:

1. Pick a vertex V in G
2. Perform DFS for V in G
3. If a W not visited Print No
4. Let G' be G with edges reversed
5. If a W not visited Print No
6. Else Print Yes

DATA STRUCTURES USED:

Queue, List, Map, stack the data structures used in strongly connected components.

SOURCE CODE:

```
from collections import defaultdict
from itertools import chain
import time


class Graph(object):
    def __init__(self, edges, vertices=()):
        self.edges = edges
        self.vertices = set(chain(*edges)).union(vertices)
        self.tails = defaultdict(list)
        for head, tail in self.edges:
            self.tails[head].append(tail)

    @classmethod
    def from_dict(cls, edge_dict):
        return cls((k, v) for k, vs in edge_dict.items() for v in vs)



class _StrongCC(object):
    def strong_connect(self, head):
        low_link, count, stack = self.low_link, self.count, self.stack
        low_link[head] = count[head] = self.counter = self.counter + 1
        stack.append(head)

        for tail in self.graph.tails[head]:
            if tail not in count:
                self.strong_connect(tail)
                low_link[head] = min(low_link[head], low_link[tail])
            elif count[tail] < count[head]:
```

```
                if tail in self.stack:
                    low_link[head] = min(low_link[head], count[tail])

            if low_link[head] == count[head]:
                component = []
                while stack and count[stack[-1]] >= count[head]:
                    component.append(chr(stack.pop()+65))

                self.connected_components.append(component)

        def __call__(self, graph):
            self.graph = graph
            self.counter = 0
            self.count = dict()
            self.low_link = dict()
            self.stack = []
            self.connected_components = []

            for v in self.graph.vertices:
                if v not in self.count:
                    self.strong_connect(v)

            return self.connected_components


    strongly_connected_components = _StrongCC()

    if __name__ == '__main__':

        count = 6
        print()
```

```
    while (count < 10):
        print("The Strongly Connected Components for the given graph is:")
        print()

        input_file = open("input" + str(count) + ".txt", "r")
        edges = []
        i = 0
        for line in input_file.readlines():
            l = line.split()
            if i == 0:
                no_of_vertices = int(l[0])
                # print("no_of_vertices", no_of_vertices)
            elif len(l) == 1:
                pass
            else:
                edges.append((ord(l[0]) - 65, ord(l[1]) - 65))

            i = i + 1
        start_time = time.time()
        print(strongly_connected_components(Graph(edges)))

        runtime = (time.time() - start_time)

print('================================================================
======')
        print('Time taken for running Strongly Connected Components Algorithm in seconds:',
runtime)
        print()
        count += 1

print("**************************************************************************")
```

```
    print('\t')
```

SAMPLE INPUT/OUTPUT:

Input 1:

```
10 17 D
A B
A D
B C
B E
C A
C G
D C
E F
E G
F G
F H
F I
F J
G E
H J
I J
J I
```

Input 2:

```
11 11 D
A B
B C
B D
C A
D E
E F
```

F D

G H

H I

I J

J G

J K

B

Input 3:

10 12 D

A B

B C

C A

B D

B E

B G

D F

E F

F H

H B

I A

J D

Input 4:

10 14 D

A C

A H

B A

B G

C D

D F

E A

E I

F J

J C

G I

H G

H F

I H

Outputs for 4 different Inputs:

RUNTIME ANALYSIS:

S.No	Graph Type	Edges	Vertices	Average Runtime (Seconds)
1	Directed	17	10	0.015621662139892578
2	Directed	11	11	0.015618085861206055
3	Directed	12	10	0.015621662139892578
4	Directed	14	10	0.015584230422973633