Graph Algorithms and Related Data Structures

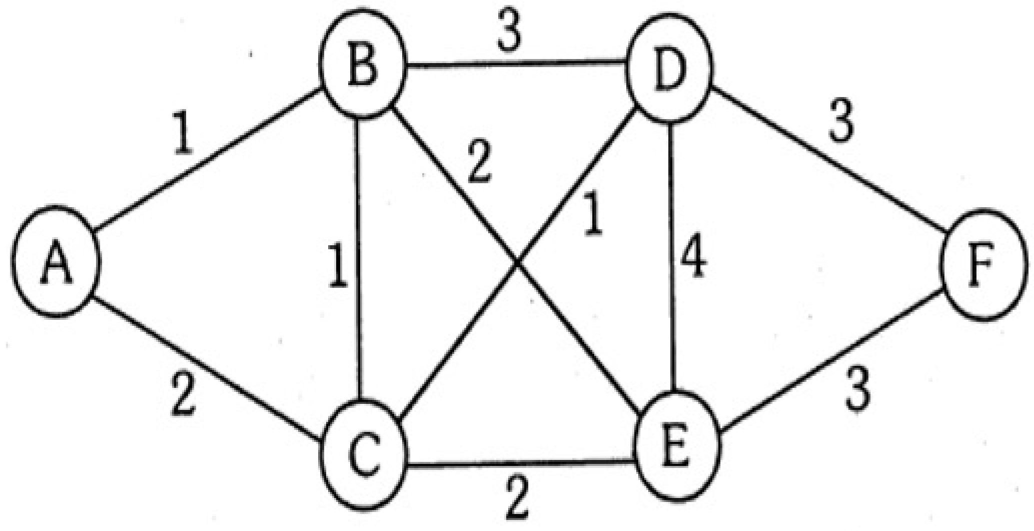
Project Report

Fall 2021

Devashri Gadgil
801243925

Graph Problems :

A graph is a collection of vertices and edges. It can be represented as follows :
G = (V,E)
A graph can be stored in form of an adjacency matrix and an adjacency list. A graph with n vertices can be mapped to n by n adjacency matrix. Assuming the name of the matrix to be A,
A[i,j] = 1 / Weight of edge - only if edge(i,j) belongs to E
A[i,j] = 0 if edge(i,j) does not belong to E
The adjacency matrix is a dense representation. However, it is very efficient for small graphs.
Example of adjacency matrix :
Undirected Weighted Graph -



Adjacency Matrix :

	A	B	C	D	E	F
A	0	1	2	0	0	0
B	1	0	1	3	2	0
C	2	1	0	1	2	0
D	0	3	1	0	4	3
E	0	2	2	4	0	3
F	0	0	0	3	3	0

Algorithms

Dijkstra’s Shortest Path Algorithm

Problem statement :

Let G be a weighted graph. The length of a path P is the sum of the weights of the edges of P. If P consists of $e_0, e_1, \ldots, e_{k-1}$ then the length of P, denoted as $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

The distance of a vertex v from a vertex s is the length of the shortest path between s and v. denoted as $d(s,v)$.
$d(s,v) = +\infty$ if no path exists.
The aim of this problem is to find out the shortest distances of all vertices from the source vertex.

Assumptions :

1. graph is connected
2. edges are directed/undirected
3. edge weights are nonnegative, i.e. $w(e) \geq 0$

Working :

Dijkstra’s algorithm employs the greedy approach.
Theme - We grow a “cloud” of vertices, beginning with start vertex s and eventually covering all the vertices. For each vertex v a label d(v): the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices. We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$. We update the labels of the vertices adjacent to u. This process is called relaxing adjacent vertices.

Pseudocode :

```
Initialize(G,s)
{
        // G is graph and s is the start vertex
        For each v ε G.V
                distance(v) = ∞
                parent(v) = 'NIL'
        distance(s) = 0
}

Dijkstra(G,w,s)
{
        // G is a graph, s is the start vertex
        cloud = ∅
        priorityQueue = G.V
        While priorityQueue is not empty:
                u = extractMin(priorityQueue)
                cloud = cloud U {u}
        For each vertex v ε G.Adj[u]:
                relax(u,v,w)
}
relax(u,v,w)
{
        // w is edge cost
        if distance(v) > distance(u) + w :
                distance(v) = distance(u) + w
                parent(v) = u
}
```

Runtime Analysis:

Please note n = number of vertices, m = no f edges

1. The time to build the binary min-heap in form of a priority queue takes O($n$log$n$).
2. Each extractMin operation that removes the smallest element then takes time O (log$n$).
3. Each Relax operation takes time O(log$n$). and there are still at most $O(m)$ such operations.
4. The total running time is = O(nlog$n$ + $m$log$n$) or **O((n + m) log$n$)**, which is O(m log$n$) if all vertices are reachable from the source.

The data structure used :

1. I have used the n by n matrix to store graph details. This matrix is constructed using NumPy and Pandas library from python. Initialized with all zeros, the matrix will put in the weight of an edge specified in the input file.
2. Parent data for each vertex is maintained in a data dictionary.
3. Priority Queue to keep track of distances is constructed using a Python list. It is sorted after every operation to maintain the min-heap property.
4. Cloud is again a data dictionary. All vertices have a False value associated with them. This is changed to True after a particular vertex is added to the cloud.

Impact on performance -

Assuming the graph has n vertices, the time complexity to build such a matrix is O(n^2). The space complexity is also O(n^2). Given a graph, to build the adjacency matrix, we need to create a square n by n matrix and fill its values with 0 and 1. It costs us O(n^2) space.
Due to the use of an adjacency matrix, accessing elements is possible in O(1) time. However, if the graph has a lot of vertices but less number of edges then a lot of space is wasted in the adjacency matrix since most of the cells would contain 0 value

Code :

```
import numpy as np
import pandas as pd

class ShortestPathDijkstras :
 def __init__(self):
   self.noOfVertices = 0
   self.noOfEdges = 0
   self.graphType = ''
   self.startVertex = ''
   self.graph = ''
   self.vertices = []
   self.dataDictParent = {}
   self.cloud = {}
   self.priorityQueue = []
   self.shortestDistanceList =[]
  def constructGraph(self,contents):
   #Purpose - This method is used to construct adjacency matrix based on
grah input
   vertices = []
   line = contents[0].split(" ")
   self.noOfVertices = line[0]
   self.noOfEdges = line[1]
   self.graphType = line[2]
   line = contents[-1].split(" ")
   self.startVertex = line[0]
   # Creating adjacency matrix
   for i in range (1,len(contents) -1):
     line = contents[i].split(" ")
     if line[0] not in vertices:
       vertices.append(line[0])
     if line[1] not in vertices:
       vertices.append(line[1])
   self.noOfVertices = len(vertices)
   matrix = np.zeros(shape=(self.noOfVertices,self.noOfVertices))
   df = pd.DataFrame(matrix, columns=vertices, index=vertices)
   for i in range(1,len(contents) -1 ):
     line = contents[i].split(" ")
     # first column then row
     df[line[1]][line[0]] = line[2]
     # Edge is both ways in an undirected graph
     if self.graphType == 'U':
       df[line[0]][line[1]] = line[2]
   self.graph = df
   self.vertices = vertices
   # Intialize cloud with all False values
   for vertex in self.vertices:
     self.cloud[vertex] = False
   self.findShortestPath()
  def findShortestPath(self):
   # Dijikstras Algorithm start
   # Initialize
   self.printInput()
   self.initialize()
   # Remove min from queue
   while(len(self.priorityQueue)!= 0):
     tup = self.priorityQueue.pop(0)
     self.shortestDistanceList.append(tup)
     src = tup[0]
     srcDist = tup[1]
     # Add source vertex to cloud
     self.cloud[src] = True
     for vertex in self.vertices:
       if self.cloud[vertex] == False:
         if self.graph.at[src,vertex] != 0.0:
           self.relax(src,vertex,self.graph[vertex][src],srcDist)

   if all(value == True for value in self.cloud.values()):
     self.printResults()

  def initialize(self):
   # Purpose - Intialize data structure
   dataDictParent = {}
   for vertex in self.vertices:
     dataDictParent[vertex] = 'NIL'
     if vertex == self.startVertex :
       self.priorityQueue.append((vertex, 0))
     else :
       self.priorityQueue.append((vertex, 100000))
   self.dataDictParent = dataDictParent
   self.priorityQueue.sort(key=lambda x:x[1])

 def relax(self,src,dest,edgeLenSrcDist,srcDist):
   #Purpose - Relax adjecent vertices
   #print("Relaxing edges for adjecent vertex " +dest)
   self.priorityQueue = dict(self.priorityQueue)
   if self.priorityQueue[dest] > srcDist + edgeLenSrcDist :
     self.priorityQueue[dest] = srcDist + edgeLenSrcDist
     self.dataDictParent[dest] = src

   self.priorityQueue = list(self.priorityQueue.items())
   self.priorityQueue.sort(key=lambda x:x[1])
 def printResults(self):
   # Purpose : Dedicated method to print results on console
   print(self.shortestDistanceList)
   hierarchy = []
   print("--------------------------------")
   print("Shortest Distance calculated!")
   for tup in self.shortestDistanceList:
     hierarchy = []
     if tup[0] != self.startVertex:
       tempParent = self.dataDictParent[tup[0]]
       if tempParent != self.startVertex:
         # new logic
         while tempParent != self.startVertex:
           hierarchy.insert(0,tempParent)
           if tup[0] not in hierarchy:
             hierarchy.append(tup[0])
           tempParent = self.dataDictParent[tempParent]
         hierarchy.insert(0,tempParent)
         hierarchy.append(tup[1])
         print(*hierarchy, sep = "--> ")
       else:
         print(tempParent,tup[0],tup[1],sep = "--> ")
  def printInput(self):
   # Purpose - Print Input
   print("Graph Type : " +self.graphType)
   print("Total number of vertices: "+ str(self.noOfVertices))
   print("Total number of edges:"+ str(self.noOfEdges))
   print("Start Vertex: "+ self.startVertex)
   print("Set of vertices: "+str(self.vertices))
   # print("Initial state of cloud\n" +str(self.cloud))
   print("Adjacency matrix: \n"+ str(self.graph))



# Driver code
def main():
 print("Welcome to Dijkstra's shortest path algorithm")
 print("Choose graph input file")
 graphNum = input("1,\n2,\n3,\n4 \nInput :")
 print(graphNum)
 if graphNum == "1":
   filePath = 'input_one.txt'
 elif graphNum == "2":
   filePath = 'input_two.txt'
 elif graphNum == "3":
   filePath = 'input_three.txt'
 elif graphNum == "4":
   filePath = 'input_four.txt'
 else:
   print("Wrong input. Terminating program")

 print("Reading graph one from file : " +str(graphNum))
 with open(filePath) as f:
   contents = f.read().splitlines()
   f.close()
 g = ShortestPathDijkstras()
 g.constructGraph(contents)

if __name__ == "__main__":
 main()
```

Run I - graph input is chosen from input_one.txt:

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

Output :

```
Welcome to Dijkstra's shortest path algorithm
Choose graph input file
1,
2,
3,
4
Input :1
1
Reading graph one from file : 1
Welcome to Dijkstra's shortest path algorithm
Choose graph input file
1,
2,
3,
4
Input :1
1
Reading graph one from file : 1
Graph Type : U
Total number of vertices: 9
Total number of edges:14
Start Vertex: A
Set of vertices: ['A', 'B', 'H', 'C', 'D', 'F', 'I', 'E', 'G']
Adjacency matrix:
     A     B     H    C     D     F    I     E    G
A  0.0   4.0   8.0  0.0   0.0   0.0  0.0   0.0  0.0
B  4.0   0.0  11.0  8.0   0.0   0.0  0.0   0.0  0.0
H  8.0  11.0   0.0  0.0   0.0   0.0  7.0   0.0  1.0
C  0.0   8.0   0.0  0.0   7.0   4.0  2.0   0.0  0.0
D  0.0   0.0   0.0  7.0   0.0  14.0  0.0   9.0  0.0
F  0.0   0.0   0.0  4.0  14.0   0.0  0.0  10.0  2.0
I  0.0   0.0   7.0  2.0   0.0   0.0  0.0   0.0  6.0
E  0.0   0.0   0.0  0.0   9.0  10.0  0.0   0.0  0.0
G  0.0   0.0   1.0  0.0   0.0   2.0  6.0   0.0  0.0
--------------------------------
Shortest Distance calculated!
A--> B--> 4.0
A--> H--> 8.0
A--> H--> G--> 9.0
A--> H--> G--> F--> 11.0
A--> B--> C--> 12.0
```

```
A--> B--> C--> I--> 14.0
A--> B--> C--> D--> 19.0
A--> H--> G--> F--> E--> 21.0
```

Run II - graph input is chosen from input_two.txt:

```
12 21 D
A B 7
A H 5
A J 3
A F 1
B F 5
F E 11
E D 7
C D 5
F C 1
B E 5
G B 5
H G 3
J H 7
J I 1
I H 3
I C 4
C K 3
K L 3
C L 1
I L 1
J L 4
A
```

```
Output :
Welcome to Dijkstra's shortest path algorithm
Choose graph input file
1,
2,
3,
4
Input :2
2
```

```
Reading graph one from file : 2
Graph Type : D
Total number of vertices: 12
Total number of edges:21
Start Vertex: A
Set of vertices: ['A', 'B', 'H', 'J', 'F', 'E', 'D', 'C', 'G', 'I', 'K', 'L']
Adjacency matrix:
    A    B    H    J    F     E    D    C    G    I    K    L
A  0.0  7.0  5.0  3.0  1.0   0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  0.0  0.0  0.0  0.0  5.0   5.0  0.0  0.0  0.0  0.0  0.0  0.0
H  0.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0  3.0  0.0  0.0  0.0
J  0.0  0.0  7.0  0.0  0.0   0.0  0.0  0.0  0.0  1.0  0.0  4.0
F  0.0  0.0  0.0  0.0  0.0  11.0  0.0  1.0  0.0  0.0  0.0  0.0
E  0.0  0.0  0.0  0.0  0.0   0.0  7.0  0.0  0.0  0.0  0.0  0.0
D  0.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  0.0  0.0  0.0  0.0   0.0  5.0  0.0  0.0  0.0  3.0  1.0
G  0.0  5.0  0.0  0.0  0.0   0.0  0.0  0.0  0.0  0.0  0.0  0.0
I  0.0  0.0  3.0  0.0  0.0   0.0  0.0  4.0  0.0  0.0  0.0  1.0
K  0.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0  0.0  0.0  0.0  3.0
L  0.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0  0.0  0.0  0.0  0.0
--------------------------------
Shortest Distance calculated!
A--> F--> 1.0
A--> F--> C--> 2.0
A--> J--> 3.0
A--> F--> C--> L--> 3.0
A--> J--> I--> 4.0
A--> H--> 5.0
A--> F--> C--> K--> 5.0
A--> B--> 7.0
A--> F--> C--> D--> 7.0
A--> H--> G--> 8.0
A--> F--> E--> 12.0
```

Run III - graph input is chosen from input_three.txt

```
10 16 U
S A 4
S B 7
S C 7
B A 5
B D 6
```

```
B E 3
A D 2
C E 1
E H 10
D F 4
D G 7
D H 4
F G 2
F T 1
G T 10
H T 11
S

Output:
Welcome to Dijkstra's shortest path algorithm
Choose graph input file
1,
2,
3,
4
Input :3
3
Reading graph one from file : 3
Graph Type : U
Total number of vertices: 10
Total number of edges:16
Start Vertex: S
Set of vertices: ['S', 'A', 'B', 'C', 'D', 'E', 'H', 'F', 'G', 'T']
Adjacency matrix:
     S    A    B    C    D     E     H    F     G     T
S  0.0  4.0  7.0  7.0  0.0   0.0   0.0  0.0   0.0   0.0
A  4.0  0.0  5.0  0.0  2.0   0.0   0.0  0.0   0.0   0.0
B  7.0  5.0  0.0  0.0  6.0   3.0   0.0  0.0   0.0   0.0
C  7.0  0.0  0.0  0.0  0.0   1.0   0.0  0.0   0.0   0.0
D  0.0  2.0  6.0  0.0  0.0   0.0   4.0  4.0   7.0   0.0
E  0.0  0.0  3.0  1.0  0.0   0.0  10.0  0.0   0.0   0.0
H  0.0  0.0  0.0  0.0  4.0  10.0   0.0  0.0   0.0  11.0
F  0.0  0.0  0.0  0.0  4.0   0.0   0.0  0.0   2.0   1.0
G  0.0  0.0  0.0  0.0  7.0   0.0   0.0  2.0   0.0  10.0
T  0.0  0.0  0.0  0.0  0.0   0.0  11.0  1.0  10.0   0.0
--------------------------------
Shortest Distance calculated!
```

```
S--> A--> 4.0
S--> A--> D--> 6.0
S--> B--> 7.0
S--> C--> 7.0
S--> C--> E--> 8.0
S--> A--> D--> H--> 10.0
S--> A--> D--> F--> 10.0
S--> A--> D--> F--> T--> 11.0
S--> A--> D--> F--> G--> 12.0
```

Run IV - graph input is chosen from input_four.txt

```
14 23 D
M Q 2
M R 2
M X 1
M P 10
N O 3
N Q 5
N U 4
O R 3
O S 6
O V 1
P O 2
P S 2
P Z 1
Q T 7
R U 12
R Y 3
S R 5
T N 3
U T 4
V W 9
V X 6
W Z 11
Y V 1
M
```

```
Output :
Welcome to Dijkstra's shortest path algorithm
Choose graph input file
1,
2,
3,
4
Input :4
4
Reading graph one from file : 4
Graph Type : D
Total number of vertices: 14
Total number of edges:23
Start Vertex: M
Set of vertices: ['M', 'Q', 'R', 'X', 'P', 'N', 'O', 'U', 'S', 'V', 'Z', 'T', 'Y', 'W']
Adjacency matrix:
    M    Q    R    X    P    N    O    U    S    V    Z    T    Y    W
M  0.0  2.0  2.0  1.0  10.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  0.0
Q  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  7.0  0.0  0.0
R  0.0  0.0  0.0  0.0   0.0  0.0  0.0  12.0  0.0  0.0   0.0  0.0  3.0  0.0
X  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  0.0
P  0.0  0.0  0.0  0.0   0.0  0.0  2.0   0.0  2.0  0.0   1.0  0.0  0.0  0.0
N  0.0  5.0  0.0  0.0   0.0  0.0  3.0   4.0  0.0  0.0   0.0  0.0  0.0  0.0
O  0.0  0.0  3.0  0.0   0.0  0.0  0.0   0.0  6.0  1.0   0.0  0.0  0.0  0.0
U  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  4.0  0.0  0.0
S  0.0  0.0  5.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  0.0
V  0.0  0.0  0.0  6.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  9.0
Z  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  0.0
T  0.0  0.0  0.0  0.0   0.0  3.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  0.0
Y  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  1.0   0.0  0.0  0.0  0.0
W  0.0  0.0  0.0  0.0   0.0  0.0  0.0   0.0  0.0  0.0  11.0  0.0  0.0  0.0
--------------------------------
Shortest Distance calculated!
M--> X--> 1.0
M--> Q--> 2.0
M--> R--> 2.0
M--> R--> Y--> 5.0
M--> R--> Y--> V--> 6.0
M--> Q--> T--> 9.0
M--> P--> 10.0
M--> P--> Z--> 11.0
M--> Q--> T--> N--> 12.0
M--> P--> O--> 12.0
M--> P--> S--> 12.0
M--> R--> U--> 14.0
M--> R--> Y--> V--> W--> 15.0
```

Prim’s Algorithm: Minimum Spanning Tree

Problem Statement :

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

$$W(T) = \Sigma_{(u,v)\ \varepsilon\ T} W(u,v)$$

Minimum spanning tree :

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Working :

Prim’s algorithm also employs greedy approach. The strategy is similar to Dijkstra’s for finding shortest path in a graph. Tree starts with an arbitrary root vertex r and grows until the tree spans all vertices in V. Prim’s algorithm has the property that edges in the cloud always form a single tree. We store with each vertex V a label v.key = the smallest weight of an edge connecting V to a vertex in the cloud.
V.key is also known as d(v).
At each step:
1. We add to cloud a vertex v outside the cloud with the smallest key label
2. We update the labels of the vertices adjacent to V (also known as relax process)

Pseudocode:

```
findMSTUsingPrims(G,w,r)
{
    For each u ε G.V
        key(u) = ∞
        parent(u) = ‘NIL’
    key(r) = 0
    priorityQueue = G.V
    while !priorityQueue .isEmpty() :
        u = extractMin(priorityQueue)
        for each v ε G.Adj[u]:
            if v ε priorityQueue and w(u,v) < key(v):
                parent(v) = u
                key(v) = w(u,v)         // decrease key operation

}
```

Runtime analysis :

1. We use binary heap for priority queue. It takes O(n log n) for creation where n is number of nodes in the graph.
2. Each extractMin operation that removes the smallest element then takes time O ($\log n$).
3. Each decrease key operation takes time O($\log n$) and there are still at most $O(m)$ such operations.
4. The total running time is = O(n$\log n$ + $m\log n$) or **O((n + m) log$n$)**

The data structure used :

1. I have used the n by n matrix to store graph details. This matrix is constructed using NumPy and Pandas library from python. Initialized with all zeros, the matrix will put in the weight of an edge specified in the input file.
2. Parent data for each vertex is maintained in a data dictionary.
3. Priority Queue to keep track of distances is constructed using a Python list. It is sorted after every operation to maintain the min-heap property.
4. Cloud is again a data dictionary. All vertices have a False value associated with them. This is changed to True after a particular vertex is added to the cloud.

Impact on performance -

Assuming the graph has n vertices, the time complexity to build such a matrix is O(n^2). The space complexity is also O(n^2). Given a graph, to build the adjacency matrix, we need to create a square n by n matrix and fill its values with 0 and 1. It costs us O(n^2) space. Due to the use of an adjacency matrix in Dijkstra’s algorithm, accessing elements is possible in O(1) time. However, if the graph has a lot of vertices but less number of edges then a lot of space is wasted in the adjacency matrix since most of the cells would contain 0 value

Code:

```
import numpy as np
import pandas as pd

class PrimsMST:
 def __init__(self):
   self.noOfVertices = 0
   self.noOfEdges = 0
   self.graphType = ''
   self.startVertex = ''
   self.graph = ''
   self.vertices = []
   self.dataDictParent = {}
   self.cloud = {}
   self.priorityQueue = []
   self.mstDistanceList =[]
  def constructGraph(self,contents):
   # Purpose - Creating adj matrix from input file
   vertices = []
   line = contents[0].split(" ")
   self.noOfVertices = line[0]
   self.noOfEdges = line[1]
   self.graphType = line[2]
   line = contents[-1].split(" ")
   self.startVertex = line[0]
   # Creating adjacency matrix
   for i in range (1,len(contents) -1):
     line = contents[i].split(" ")
     if line[0] not in vertices:
       vertices.append(line[0])
     if line[1] not in vertices:
       vertices.append(line[1])
   self.noOfVertices = len(vertices)
   matrix = np.zeros(shape=(self.noOfVertices,self.noOfVertices))
   df = pd.DataFrame(matrix, columns=vertices, index=vertices)
   for i in range(1,len(contents) -1 ):
```

```
   line = contents[i].split(" ")
   # first column then row
   df[line[1]][line[0]] = line[2]
   # Edge is both ways in an undirected graph
   if self.graphType == 'U':
     df[line[0]][line[1]] = line[2]
 self.graph = df
 self.vertices = vertices
 for vertex in self.vertices:
   self.cloud[vertex] = False
 self.primsAlgo()
def primsAlgo(self):
 # Purpose - Prims Algorithm to find MST
 self.printInput()
 self.initialize()
 while(len(self.priorityQueue)!= 0):
   tup = self.priorityQueue.pop(0)
   self.mstDistanceList.append(tup)
   src = tup[0]
   srcDist = tup[1]
   # Add  vertex to cloud
   self.cloud[src] = True
   for vertex in self.vertices:
     if self.cloud[vertex] == False:
       if self.graph.at[src,vertex] != 0.0:
         # Relax vertex only if its adjecent to source and not in cloud
         self.relax(src,vertex,self.graph[vertex][src],srcDist)

 if all(value == True for value in self.cloud.values()):
   self.printResults()
def printInput(self):
 # Purpose - Print Input
 print("Graph Type : " +self.graphType)
 print("Total number of vertices: "+ str(self.noOfVertices))
 print("Total number of edges:"+ str(self.noOfEdges))
 print("Start Vertex: "+ self.startVertex)
 print("Set of vertices: "+str(self.vertices))
 # print("Initial state of cloud\n" +str(self.cloud))
 print("Adjacency matrix: \n"+ str(self.graph))
def initialize(self):
   # Initialize all vertices with NIL parent and very large key value
   dataDictParent = {}
   for vertex in self.vertices:
     dataDictParent[vertex] = 'NIL'
     if vertex == self.startVertex :
       self.priorityQueue.append((vertex, 0))
     else :
       self.priorityQueue.append((vertex, 100000))
   self.dataDictParent = dataDictParent
   self.priorityQueue.sort(key=lambda x:x[1])
  def relax(self,src,dest,edgeLenSrcDist,srcDist):
   # Purpose - Recalculate distances for all aj vertices to src to find
shortest path
   #print("Relaxing edges for adjecent vertex " +dest)
   self.priorityQueue = dict(self.priorityQueue)
   if edgeLenSrcDist < self.priorityQueue[dest]:
     self.priorityQueue[dest] = edgeLenSrcDist
     self.dataDictParent[dest] = src

   self.priorityQueue = list(self.priorityQueue.items())
   self.priorityQueue.sort(key=lambda x:x[1])
  def printResults(self):
   # Purpose - Print Output
   print("--------------------------------")
   print("Minimum spanning tree calculated!")
   print("Parent ---> Child :: Key")
   self.mstDistanceList = dict(self.mstDistanceList)
   for (k,v) in self.dataDictParent.items():
     print (v +"--->" + k + " :: " +str(self.mstDistanceList[k]))
   print("")
   totalDist = 0
   for dist in self.mstDistanceList:
     totalDist = totalDist + self.mstDistanceList[dist]
   print("Total cost of minimum spanning tree = " +str(totalDist))



def main():
 print("Welcome to Prim's Minimum Spanning Tree algorithm")
 print("Choose graph input file")
 graphNum = input("1,\n2,\n3,\n4 \nInput :")
 if graphNum == "1":
   filePath = 'input_one.txt'
 elif graphNum == "2":
   filePath = 'input_two.txt'
 elif graphNum == "3":
   filePath = 'input_three.txt'
 elif graphNum == "4":
   filePath = 'input_four.txt'
 else:
   print("Wrong input. Terminating program")

 print("Reading graph one from file : " +str(graphNum))
 with open(filePath) as f:
   contents = f.read().splitlines()
   f.close()
 g = PrimsMST()
 g.constructGraph(contents)

if __name__ == "__main__":
 main()
```

Run I - graph input is chosen from input_one.txt:

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

A

Output:

```
Welcome to Prim's Minimum Spanning Tree algorithm
Choose graph input file
1,
2,
3,
4
Input :1
Reading graph one from file : 1
Graph Type : U
Total number of vertices: 9
Total number of edges:14
Start Vertex: A
Set of vertices: ['A', 'B', 'H', 'C', 'D', 'F', 'I', 'E', 'G']
Adjacency matrix:
    A    B    H   C    D    F   I    E   G
A  0.0   4.0   8.0  0.0   0.0   0.0  0.0   0.0  0.0
B  4.0   0.0  11.0  8.0   0.0   0.0  0.0   0.0  0.0
H  8.0  11.0   0.0  0.0   0.0   0.0  7.0   0.0  1.0
C  0.0   8.0   0.0  0.0   7.0   4.0  2.0   0.0  0.0
D  0.0   0.0   0.0  7.0   0.0  14.0  0.0   9.0  0.0
F  0.0   0.0   0.0  4.0  14.0   0.0  0.0  10.0  2.0
I  0.0   0.0   7.0  2.0   0.0   0.0  0.0   0.0  6.0
E  0.0   0.0   0.0  0.0   9.0  10.0  0.0   0.0  0.0
G  0.0   0.0   1.0  0.0   0.0   2.0  6.0   0.0  0.0
--------------------------------
Minimum spanning tree calculated!
Parent ---> Child :: Key

A--->B :: 4.0
A--->H :: 8.0
F--->C :: 4.0
C--->D :: 7.0
G--->F :: 2.0
C--->I :: 2.0
D--->E :: 9.0
H--->G :: 1.0

Total cost of minimum spanning tree = 37.0
```

Run II - graph input is chosen from input_two.txt:

```
10 16 U
S A 4
S B 7
S C 7
B A 5
B D 6
B E 3
A D 2
C E 1
E H 10
D F 4
D G 7
D H 4
F G 2
F T 1
G T 10
H T 11
S
```

```
Output:
Welcome to Prim's Minimum Spanning Tree algorithm
Choose graph input file
1,
2,
3,
4
Input :2
Reading graph one from file : 2
Graph Type : U
Total number of vertices: 10
Total number of edges:16
Start Vertex: S
Set of vertices: ['S', 'A', 'B', 'C', 'D', 'E', 'H', 'F', 'G', 'T']
Adjacency matrix:
     S    A    B    C    D     E     H    F     G     T
S  0.0  4.0  7.0  7.0  0.0   0.0   0.0  0.0   0.0   0.0
A  4.0  0.0  5.0  0.0  2.0   0.0   0.0  0.0   0.0   0.0
B  7.0  5.0  0.0  0.0  6.0   3.0   0.0  0.0   0.0   0.0
```

```
C  7.0  0.0  0.0  0.0  0.0   1.0   0.0  0.0   0.0   0.0
D  0.0  2.0  6.0  0.0  0.0   0.0   4.0  4.0   7.0   0.0
E  0.0  0.0  3.0  1.0  0.0   0.0  10.0  0.0   0.0   0.0
H  0.0  0.0  0.0  0.0  4.0  10.0   0.0  0.0   0.0  11.0
F  0.0  0.0  0.0  0.0  4.0   0.0   0.0  0.0   2.0   1.0
G  0.0  0.0  0.0  0.0  7.0   0.0   0.0  2.0   0.0  10.0
T  0.0  0.0  0.0  0.0  0.0   0.0  11.0  1.0  10.0   0.0
-------------------------------
Minimum spanning tree calculated!
Parent ---> Child :: Key
S--->A :: 4.0
A--->B :: 5.0
E--->C :: 1.0
A--->D :: 2.0
B--->E :: 3.0
D--->H :: 4.0
D--->F :: 4.0
F--->G :: 2.0
F--->T :: 1.0

Total cost of minimum spanning tree = 26.0
```

Run III - graph input is chosen from input_three.txt

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

```
F K 13
F J 19
G H 3
I A 15
G

Output :
Welcome to Prim's Minimum Spanning Tree algorithm
Choose graph input file
1,
2,
3,
4
Input :3
Reading graph one from file : 3
Graph Type : U
Total number of vertices: 11
Total number of edges:21
Start Vertex: G
Set of vertices: ['A', 'B', 'G', 'E', 'J', 'C', 'D', 'K', 'H', 'F', 'I']
Adjacency matrix:
     A    B    G    E    J    C    D    K    H    F    I
A   0.0  5.0 21.0 12.0  1.0  0.0  0.0  0.0  0.0  0.0 15.0
B   5.0  0.0 18.0  0.0 20.0  9.0  0.0  0.0  0.0  0.0  0.0
G  21.0 18.0  0.0  2.0  0.0 17.0 11.0  0.0  3.0  0.0  0.0
E  12.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  6.0 10.0
J   1.0 20.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 19.0  0.0
C   0.0  9.0 17.0  0.0  0.0  0.0 16.0  8.0  0.0  0.0  0.0
D   0.0  0.0 11.0  0.0  0.0 16.0  0.0  0.0 14.0  7.0  0.0
K   0.0  0.0  0.0  0.0  0.0  8.0  0.0  0.0  0.0 13.0  0.0
H   0.0  0.0  3.0  0.0  0.0  0.0 14.0  0.0  0.0  4.0  0.0
F   0.0  0.0  0.0  6.0 19.0  0.0  7.0 13.0  4.0  0.0  0.0
I  15.0  0.0  0.0 10.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
--------------------------------
Minimum spanning tree calculated!
Parent ---> Child :: Key
E--->A :: 12.0
A--->B :: 5.0
G--->E :: 2.0
A--->J :: 1.0
B--->C :: 9.0
F--->D :: 7.0
```

```
C--->K :: 8.0
G--->H :: 3.0
H--->F :: 4.0
E--->I :: 10.0

Total cost of minimum spanning tree = 61.0
```

Run IV - graph input is chosen from input_four.txt

```
9 16 U
A B 4
B C 11
B D 9
C A 8
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
A
```

```
Output:
Welcome to Prim's Minimum Spanning Tree algorithm
Choose graph input file
1,
2,
3,
4
Input :4
Reading graph one from file : 4
Graph Type : U
Total number of vertices: 9
Total number of edges:16
Start Vertex: A
Set of vertices: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
```

```
Adjacency matrix:
    A    B    C   D   E   F    G    H    I
A  0.0   4.0   8.0  0.0  0.0  0.0   0.0   0.0   0.0
B  4.0   0.0  11.0  9.0  8.0  0.0   0.0   0.0   0.0
C  8.0  11.0   0.0  7.0  0.0  1.0   0.0   0.0   0.0
D  0.0   9.0   7.0  0.0  2.0  6.0   0.0   0.0   0.0
E  0.0   8.0   0.0  2.0  0.0  5.0   7.0   4.0   0.0
F  0.0   0.0   1.0  6.0  5.0  0.0   0.0   2.0   0.0
G  0.0   0.0   0.0  0.0  7.0  0.0   0.0  14.0   9.0
H  0.0   0.0   0.0  0.0  4.0  2.0  14.0   0.0  10.0
I  0.0   0.0   0.0  0.0  0.0  0.0   9.0  10.0   0.0
--------------------------------
Minimum spanning tree calculated!
Parent ---> Child :: Key
A--->B :: 4.0
A--->C :: 8.0
E--->D :: 2.0
H--->E :: 4.0
C--->F :: 1.0
E--->G :: 7.0
F--->H :: 2.0
G--->I :: 9.0

Total cost of minimum spanning tree = 37.0
```

Strongly connected components

Problem Statement:

Given a digraph G with n vertices and m edges. Decompose this graph into Strongly Connected Components (SCCs). A digraph is a graph whose edges are all connected. Each edge goes in one direction.

Strong Connectivity :

A graph is said to be strongly connected if each vertex can reach every other vertex in the graph.

Working :

1. Pick a vertex v in Graph G.
2. Perform a DFS from v in G.
3. If there’s a vertex w not visited, print “no”.
4. Let G’ be G with edges reversed(Transpose of G).
5. Perform a DFS from v in G’.
6. If there’s a vertex w not visited, print “no”.
7. else, print “yes”.
8. If every vertex of G’ is visited by this second DFS, then the graph is strongly connected, for each of the vertices visited in this DFS can reach S.

Pseudocode:

```
StronglyConnectedComponents(G):
1. Call DFS(G) to compute finish times for each vertex
2. Construct GT ( Transpose of graph G)
3. Call DFS(GT) but in the main loop of DFS consider vertices in decreasing order of finish times as computed in step one
4. Output the vertices of each tree in the DFS forest formed in the previous step as a separate strongly connected component.
```

Runtime analysis:

1. The above method calls DFS, finds the transpose of the graph, and again calls DFS on the transpose.
2. DFS takes O(n+m) for a graph represented using an adjacency list. Reversing a graph also takes O(n+m) time. For reversing the graph, we simply traverse all adjacency lists.
3. The time complexity to find strongly connected components using adjacency list is O(n+m)
4. Assuming the graph has n vertices, the time complexity to run DFS using an adjacency matrix is $O(n^2)$. Reversing a graph also takes $O(n^2)$ time.
5. The time complexity to find strongly connected components using adjacency matrix is $O(n^2)$

The data structure used :

1. Graph G and its transpose are stored as an adjacency matrix
2. Vertices are stored as a python list

3. Python dictionary is used to keep track of color, parent, start time, and finish time information for each vertex for both DFS runs.
4. The output of the first DFS is stored in a python and then reversed to find vertices in decreasing order of their finish times.

Impact on performance:

Since I am using an adjacency matrix instead of adjacent list, the algorithm takes $O(n^2)$ to mention individual strongly connected components.

Code:

```
import numpy as np
import pandas as pd

class StronglyConnectedComponents:
 def __init__(self):
   self.noOfVertices = 0
   self.noOfEdges = 0
   self.graphType = ''
   self.startVertex = ''
   self.graph = ''
   self.vertices = []
   self.transposeGraph = ''
   self.dataDict = {}
   self.time = 0
   self.DFS_Output = []
   self.isInitialRun = ''
  def constructGraph(self,contents):
   # Function to construct main graph from input file
   vertices = []
   line = contents[0].split(" ")
   self.noOfVertices = line[0]
   self.noOfEdges = line[1]
   self.graphType = line[2]
   line = contents[-1].split(" ")
   self.startVertex = line[0]
   # Creating adjacency matrix
   for i in range (1,len(contents) -1):
     line = contents[i].split(" ")
     if line[0] not in vertices:
       vertices.append(line[0])
     if line[1] not in vertices:
       vertices.append(line[1])
   self.noOfVertices = len(vertices)
   matrix = np.zeros(shape=(self.noOfVertices,self.noOfVertices))
   df = pd.DataFrame(matrix, columns=vertices, index=vertices)
   for i in range(1,len(contents) -1 ):
     line = contents[i].split(" ")
     # first column then row
     df[line[1]][line[0]] = line[2]
     # Edge is both ways in an undirected graph
     if self.graphType == 'U':
       df[line[0]][line[1]] = line[2]
   self.graph = df
   self.vertices = vertices


  def findStronglyConnectedComponents(self):
   # Function to call sub ordinate functions
   self.printInput()
   self.runDFS()

 def printInput(self):
   # Purpose - Print Input
   print("Graph Type : " +self.graphType)
   print("Total number of vertices: "+ str(self.noOfVertices))
   print("Total number of edges:"+ str(self.noOfEdges))
   print("Start Vertex: "+ self.startVertex)
   print("Set of vertices: "+str(self.vertices))
   # print("Initial state of cloud\n" +str(self.cloud))
   print("Adjacency matrix: \n"+ str(self.graph))
   print("Transpose Adjacency matrix: \n" +str(self.transposeGraph))
  def runDFS(self):
   #Purpose - DFS Entry Function
   print("Running Depth First Search")
   self.initialize(self.vertices)
   self.isInitialRun = True
   self.time = 0
   for vertex in self.vertices:
     if self.dataDict[vertex]['color'] == 'white':
       self.DFS_Visit(vertex)

   # Arraging vertices in decresing order of their finish times
   self.DFS_Output.reverse()
   print("DFS_Output : \n"+ str(self.DFS_Output))

   # DFS On transpose graph as per DFS_Output order
   print("Running Depth First Search on transpose")
   self.initialize(self.DFS_Output)
   self.isInitialRun = False
   self.time = 0
   for vertex in self.DFS_Output:
     if self.dataDict[vertex]['color'] == 'white':
       print("-----Component-----")
       self.DFS_Visit(vertex)

 def initialize(self, seq):
   # Function to initialize data structure for DFS
   for vertex in seq:
     self.dataDict[vertex] = {
       'color':'white',
       'parent':'NIL',
       'startTime': 0,
       'endTime': 0
     }

  def constructTranspose(self,contents):
   # Function to construct reverse graph
   print("Constructing transpose")
   matrix = np.zeros(shape=(self.noOfVertices,self.noOfVertices))
   df = pd.DataFrame(matrix, columns=self.vertices, index=self.vertices)
   for i in range(1,len(contents) -1 ):
     line = contents[i].split(" ")
     # Reverse the direction of orginal edges
     df[line[0]][line[1]] = line[2]
     # Edge is both ways in an undirected graph
     if self.graphType == 'U':
       df[line[0]][line[1]] = line[2]
   self.transposeGraph = df
   self.findStronglyConnectedComponents()

 def DFS_Visit(self,vertex):
   # DFS Main function
   if self.isInitialRun:
     seq = self.vertices
     graph = self.graph
   else:
     seq = self.DFS_Output
     graph = self.transposeGraph

   self.time = self.time + 1
   self.dataDict[vertex]['startTime'] = self.time
   self.dataDict[vertex]['color'] = 'gray'

   # for loop
   for v in seq:
     if graph.at[vertex,v] != 0.0:
       if self.dataDict[v]['color'] == 'white':
         self.dataDict[v]['parent'] = vertex
         self.DFS_Visit(v)
    self.dataDict[vertex]['color'] = 'black'
   self.time = self.time + 1
   self.dataDict[vertex]['endTime'] = self.time

   if self.isInitialRun == True:
     self.DFS_Output.append(vertex)
   else:
     print(vertex)
# Drivers code
def main():
 print("Lets find SCC for given directed graph")
 print("Choose graph input file")
 graphNum = input("1,\n2,\n3,\n4 \nInput :")
 print(graphNum)
 if graphNum == "1":
   filePath = 'input_one.txt'
 elif graphNum == "2":
   filePath = 'input_two.txt'
```

```
 elif graphNum == "3":
   filePath = 'input_three.txt'
 elif graphNum == "4":
   filePath = 'input_four.txt'
 else:
   print("Wrong input. Terminating program")

 print("Reading graph one from file : " +str(graphNum))
 with open(filePath) as f:
   contents = f.read().splitlines()
   f.close()
  g = StronglyConnectedComponents()
 g.constructGraph(contents)
 g.constructTranspose(contents)

if __name__ == "__main__":
 main()
```

Run I - graph input is chosen from input_one.txt:

```
11 18 D
A B 5
A C 1
A J 3
B C 2
B K 4
C E 1
D J 2
D G 6
E A 3
F D 3
F I 7
G K 5
H F 1
I A 8
I E 1
I H 2
J G 5
K J 7
```

A

```
Output:
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :2
2
Reading graph one from file : 2
Constructing transpose
Graph Type : D
Total number of vertices: 5
Total number of edges:5
Start Vertex: 1
Set of vertices: ['1', '0', '2', '3', '4']
Adjacency matrix:
     1    0    2    3    4
1  0.0  1.0  0.0  0.0  0.0
0  0.0  0.0  1.0  1.0  0.0
2  1.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0  1.0
4  0.0  0.0  0.0  0.0  0.0
Transpose Adjacency matrix:
     1    0    2    3    4
1  0.0  0.0  1.0  0.0  0.0
0  1.0  0.0  0.0  0.0  0.0
2  0.0  1.0  0.0  0.0  0.0
3  0.0  1.0  0.0  0.0  0.0
4  0.0  0.0  0.0  1.0  0.0
Running Depth First Search
DFS_Output :
['1', '0', '3', '4', '2']
Running Depth First Search on transpose
-----Component-----
0
2
1
-----Component-----
3
```

```
-----Component-----
4
```

Run II - graph input is chosen from input_two.txt:

```
11 18 D
A B 2
B C 3
B E 1
C D 7
D C 1
D G 5
E B 6
E D 2
E F 2
F H 1
F G 4
G J 3
H I 7
H J 1
I F 4
J K 2
K H 1
K J 2
```

```
Output:
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :2
2
Reading graph one from file : 2
Constructing transpose
Graph Type : D
Total number of vertices: 5
Total number of edges:5
```

```
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :2
2
Reading graph one from file : 2
Constructing transpose
Graph Type : D
Total number of vertices: 11
Total number of edges:18
Start Vertex: K
Set of vertices: ['A', 'B', 'C', 'E', 'D', 'G', 'F', 'H', 'J', 'I', 'K']
Adjacency matrix:
     A    B    C    E    D    G    F    H    J    I    K
A  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  0.0  0.0  3.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  0.0  0.0  0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0
E  0.0  6.0  0.0  0.0  2.0  0.0  2.0  0.0  0.0  0.0  0.0
D  0.0  0.0  1.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0  0.0  0.0
F  0.0  0.0  0.0  0.0  0.0  4.0  0.0  1.0  0.0  0.0  0.0
H  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  7.0  0.0
J  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0
I  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0
K  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
Transpose Adjacency matrix:
     A    B    C    E    D    G    F    H    J    I    K
A  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  2.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  3.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
E  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
D  0.0  0.0  7.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  0.0  5.0  0.0  4.0  0.0  0.0  0.0  0.0
F  0.0  0.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0
H  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  1.0
J  0.0  0.0  0.0  0.0  0.0  3.0  0.0  1.0  0.0  0.0  0.0
I  0.0  0.0  0.0  0.0  0.0  0.0  0.0  7.0  0.0  0.0  0.0
K  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  0.0  0.0
Running Depth First Search
```

```
DFS_Output :
['A', 'B', 'E', 'C', 'D', 'G', 'J', 'K', 'H', 'I', 'F']
Running Depth First Search on transpose
-----Component-----
A
-----Component-----
E
B
-----Component-----
D
C
-----Component-----
J
K
H
I
F
G
```

Run III - graph input is chosen from input_three.txt

```
11 17 D
A B 4
B C 3
B D 1
B F 3
C A 2
C E 2
C H 3
D G 4
E D 5
E H 1
E K 5
F B 2
F D 1
G I 4
I D 4
J E 5
K J 6
A
```

Output:

```
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :2
2
Reading graph one from file : 2
Constructing transpose
Graph Type : D
Total number of vertices: 5
Total number of edges:5
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :2
2
Reading graph one from file : 2
Constructing transpose
Graph Type : D
Total number of vertices: 11
Total number of edges:18
Start Vertex: K
Set of vertices: ['A', 'B', 'C', 'E', 'D', 'G', 'F', 'H', 'J', 'I', 'K']
Adjacency matrix:
     A    B    C    E    D    G    F    H    J    I    K
A  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  0.0  0.0  3.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  0.0  0.0  0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0
E  0.0  6.0  0.0  0.0  2.0  0.0  2.0  0.0  0.0  0.0  0.0
D  0.0  0.0  1.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0  0.0  0.0
F  0.0  0.0  0.0  0.0  0.0  4.0  0.0  1.0  0.0  0.0  0.0
H  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  7.0  0.0
J  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0
I  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0
K  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
Transpose Adjacency matrix:
    A   B   C   E   D   G   F   H   J   I   K
A  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  2.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :3
3
Reading graph one from file : 3
Constructing transpose
Graph Type : D
Total number of vertices: 11
Total number of edges:17
Start Vertex: A
Set of vertices: ['A', 'B', 'C', 'D', 'F', 'E', 'H', 'G', 'K', 'I', 'J']
Adjacency matrix:
    A   B   C   D   F   E   H   G   K   I   J
A  0.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  0.0  0.0  3.0  1.0  3.0  0.0  0.0  0.0  0.0  0.0  0.0
C  2.0  0.0  0.0  0.0  0.0  2.0  3.0  0.0  0.0  0.0  0.0
D  0.0  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0
F  0.0  2.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
E  0.0  0.0  0.0  5.0  0.0  0.0  1.0  0.0  5.0  0.0  0.0
H  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0
K  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  6.0
I  0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
J  0.0  0.0  0.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0
Transpose Adjacency matrix:
    A   B   C   D   F   E   H   G   K   I   J
A  0.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
B  4.0  0.0  0.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
D  0.0  1.0  0.0  0.0  1.0  5.0  0.0  0.0  0.0  4.0  0.0
F  0.0  3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
E  0.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  5.0
H  0.0  0.0  3.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
K  0.0  0.0  0.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0
```

```
I  0.0  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0
J  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  6.0  0.0  0.0
Running Depth First Search
DFS_Output :
['A', 'B', 'F', 'C', 'E', 'K', 'J', 'H', 'D', 'G', 'I']
Running Depth First Search on transpose
-----Component-----
F
B
C
A
-----Component-----
K
J
E
-----Component-----
H
-----Component-----
G
I
D
```

Run IV - graph input is chosen from input_four.txt

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

Output :

```
Lets find SCC for given directed graph
Choose graph input file
1,
2,
3,
4
Input :4
4
Reading graph one from file : 4
Constructing transpose
Graph Type : D
Total number of vertices: 10
Total number of edges:14
Start Vertex: A
Set of vertices: ['A', 'C', 'H', 'B', 'G', 'D', 'F', 'E', 'I', 'J']
Adjacency matrix:
     A    C    H    B    G    D    F    E    I    J
A  0.0  3.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
C  0.0  0.0  0.0  0.0  0.0  7.0  0.0  0.0  0.0  0.0
H  0.0  0.0  0.0  0.0  3.0  0.0  1.0  0.0  0.0  0.0
B  5.0  0.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  4.0  0.0
D  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
F  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0
E  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  0.0
I  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
J  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
Transpose Adjacency matrix:
     A    C    H    B    G    D    F    E    I    J
A  0.0  0.0  0.0  5.0  0.0  0.0  0.0  2.0  0.0  0.0
C  3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0
H  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  5.0  0.0
B  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
G  0.0  0.0  3.0  6.0  0.0  0.0  0.0  0.0  0.0  0.0
D  0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
F  0.0  0.0  1.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
E  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
I  0.0  0.0  0.0  0.0  4.0  0.0  0.0  2.0  0.0  0.0
J  0.0  0.0  0.0  0.0  0.0  0.0  3.0  0.0  0.0  0.0
Running Depth First Search
DFS_Output :
['E', 'B', 'A', 'H', 'G', 'I', 'C', 'D', 'F', 'J']
```

```
Running Depth First Search on transpose
-----Component-----
E
-----Component-----
B
-----Component-----
A
-----Component-----
G
I
H
-----Component-----
D
F
J
C
```