##ITCS-6114 Project 2 - M.B.Curlee This project demonstrates some algorithm fundamentals including:

1. Singles-source shortest path algorithm
2. Minimum Spanning Tree (MST)
3. Strongly Connected Components (SSCs)

I chose to demonstrate this process in jupyter notebook as this provides a viable and clean way to both display code and describe the implementation. Examples were implemented from demonstrated algorithms in class.

The raw code for my implementation can be be at:

1. https://github.com/mcurlee3/ITCS6114_Project2
2. https://colab.research.google.com/drive/1P1nUgikbu-RyVatybkWcpgnabckQDE5K#scrollTo=81wO0t5oeIj4

----Imports-----

```
import os
cwd = os.getcwd()
print(cwd)
```

```
/content
```

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
from collections import defaultdict
import sys
import time
import pandas as pd
#from Dykstra import dGraph
#from Kruskal import kGraph
```

Module provided for parsing the text file style graphs.

```
def parseText(txtFile):
    lst = []
    for line in txtFile:
        x = line.split()
        lst.append(x)
    stats = lst.pop(0)
    src = lst.pop()
    lst = [lst, stats, src]
    return lst
```

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. You will print each path and path cost for a given source.

I chose several graphs from the internet that looked visually interesting. I then transposed them into text files attached to this implementation and resembling the following, with line 1 indicating the number of vertices, edges and U or D to describe directed or undirected. The middle lines contain the starting node, the ending node and the weight/distance between. The final line is the starting node.

#samp1.txt

```
18 27 U
A B 6
A G 2
A F 5
B H 3
B C 7
C I 2
C D 6
D J 3
D E 5
E K 2
E F 6
F L 3
G N 4
G R 4
H M 3
H O 4
I N 3
I P 4
J O 2
J Q 3
K P 3
K R 2
L Q 3
L M 3
M P 5
N Q 4
O R 5
A
```

##Dikstra Dikstra's algorithm provides a clean method for determining the shortest path from a starting node and any other node, by traversing the lowest cost path. The idea is to continiously calculate the shortest distance from start to end, while excluding longer distances/weights when making updates. The algorithm works as follows:

1. Initialization of all nodes with distance "infinite"; initialization of the starting node with 0

2. Marking of the distance of the starting node as permanent, all other distances as temporarily.
3. Setting of starting node as active.
4. Calculation of the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges.
5. If such a calculated distance of a node is smaller as the current one, update the distance and set the current node as antecessor. This step is also called update and is Dijkstra's central idea.
6. Setting of the node with the minimal temporary distance as active. Mark its distance as permanent.
7. Repeating of steps 4 to 7 until there aren't any nodes left with a permanent distance, which neighbours still have temporary distances.

Dikstra's algorithm runs with an expected runtime of (m+n)(log(n).

#My Implementation I chose to use python for my implementation as defaultdict(list) addapts nicely to the unequal size of internal lists. In this implementation, I could find no real measurable increase in time complexity with operations dedicated to creating or maintaining the lists. I found that many of the graphs I decided to use worked perfectly for this implementation.

```
class dGraph:
    def __init__(self, directed):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, beg, end, wt):
        self.graph[beg].append([end, wt])

        if self.directed is False:
            self.graph[end].append([beg, wt])
        elif self.directed is True:
            self.graph[end] = self.graph[end]

    def find_min(self, dist, visited):
        min = float('inf')
        index = -1
        for vert in self.graph.keys():
            if visited[vert] is False and dist[vert] < min:
                min = dist[vert]
                index = vert

        return index

    def Dikstra(self, start):
        visited = {i: False for i in self.graph}
        dist = {i: float('inf') for i in self.graph}
        parent = {i: None for i in self.graph}
```

```
        dist[start] = 0

        # find shortest path for all vertices
        for i in range(len(self.graph) - 1):
            u = self.find_min(dist, visited)
            visited[u] = True
            for v, w in self.graph[u]:

                if visited[v] is False and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u
        return parent, dist

    def printPath(self, parent, v):
        if parent[v] is None:
            return
        self.printPath(parent, parent[v])
        print(v,end=" ")

    def printSolution(self, dist, parent, start):
        print('{}\t\t{}\t{}'.format('Vertex', 'Distance', 'Path'))

        for i in self.graph.keys():
            if i == start:
                continue
            if  dist[i]==float("inf"):
                continue
            src = ord(start)-65
            print('{} -> {}\t\t{}\t\t{}'.format(start, i, dist[i],
start), end=' ')
            self.printPath(parent, i)
            print()

def printDykstra():
    print('###-------------Dykstra--------------###')
    file1 = open('/content/gdrive/MyDrive/6114/Project2/samp1.txt',
'r')
    file2 = open('/content/gdrive/MyDrive/6114/Project2/samp2.txt',
'r')
    file3 = open('/content/gdrive/MyDrive/6114/Project2/samp3.txt',
'r')
    file4 = open('/content/gdrive/MyDrive/6114/Project2/samp4.txt',
'r')

    file1 = parseText(file1)
    file2 = parseText(file2)
    file3 = parseText(file3)
    file4 = parseText(file4)
```

```
    sample = [file1, file2, file3, file4]
    for file in sample:
        start = file.pop()
        start = start[0]
        stats = file.pop()
        file = file[0]

        if stats[2] == 'D':
            directed = True
            dir = 'Directed'
        else:
            directed = False
            dir = 'Undirected'
        graph = dGraph(directed)
        for l in file:
            graph.addEdge(l[0], l[1], int(l[2]))
        print('Starting node for this ' + dir + ' graph is: ' + start)
        startTime = time.time()
        parent, dist = graph.Dikstra(start)

        graph.printSolution(dist, parent, start)
        runTime = (time.time() - startTime)*1000

        print('The runtime for this graph took: ', runTime, '
microseconds')

print('************************************************************')
        print()

printDykstra()
```

```
###-------------Dykstra--------------###
Starting node for this Undirected graph is: A
Vertex          Distance   Path
A -> B          6          A B
A -> G          2          A G
A -> F          5          A F
A -> H          9          A B H
A -> C          11         A G N I C
A -> I          9          A G N I
A -> D          15         A G R K E D
A -> J          13         A G N Q J
A -> E          10         A G R K E
A -> K          8          A G R K
A -> L          8          A F L
A -> N          6          A G N
A -> R          6          A G R
A -> M          11         A F L M
A -> O          11         A G R O
A -> P          11         A G R K P
```

```
A -> Q          10         A G N Q
The runtime for this graph took:  1.1081695556640625
************************************************

Starting node for this Undirected graph is: A
Vertex          Distance   Path
A -> J          30         A J
A -> I          12         A I
A -> C          20         A C
A -> B          29         A B
A -> D          47         A C D
A -> E          38         A C E
A -> F          49         A I H F
A -> G          26         A I G
A -> H          34         A I H
The runtime for this graph took:  0.3571510314941406
************************************************

Starting node for this Undirected graph is: A
Vertex          Distance   Path
A -> B          2          A B
A -> D          10         A D
A -> C          8          A B C
A -> E          11         A D E
A -> F          17         A D E F
A -> G          21         A D E H G
A -> H          18         A D E H
The runtime for this graph took:  0.3714561462402344
************************************************

Starting node for this Directed graph is: A
Vertex          Distance   Path
A -> B          3          A B
A -> C          7          A B C
A -> D          1          A D
A -> H          8          A B H
A -> I          12         A D I
A -> E          9          A D E
A -> G          11         A D E G
A -> F          21         A D F
The runtime for this graph took:  0.3559589385986328
************************************************
```

Problem 2: Minimum Spanning Tree Algorithm

Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight. Use either Kruskal's or Prim's algorithm to find Minimum Spanning Tree (MST). You will printout edges of the tree and total cost of minimum spanning tree

##Kruskal Kruskal's algorithm is stated as a greedy algorithm that finds local minimums in an attempt to find an overal global minimum for a spanning tree graph.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

##My Implementation I used the same setup for defaultdict(list) in this implementation as noted above. Here again I could find no real time complexity increase by using these lists, in terms of measured microseconds. Kruskal's runs with an expected time complexity of nlogn. I found that the graphs I chose for this project may have been too highly connected and may not have been a good representative of the overall capability of this algorithm.

```
class kGraph:

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


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

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        x_root = self.find(parent, x)
        y_root = self.find(parent, y)

        if rank[x_root] < rank[y_root]:
            parent[x_root] = y_root
        elif rank[x_root] > rank[y_root]:
            parent[y_root] = x_root

        else:
            parent[y_root] = x_root
            rank[x_root] += 1

    def KruskalMST(self):
```

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

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        nodes = []
        while e < self.V - 1:
            nodes.append(chr(i+65))
            u, v, w = self.graph[i]

            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
        print('This graph contained the following nodes:')
        print(nodes)
        print('\033[1m' + "Edge-Child \t\t Weight" + '\033[0m')
        print()
        res=0
        for u, v, weight in result:
            print(chr(u + 65), "-", chr(v + 65), "\t\t\t", weight)
            res+=weight
        print('\033[1m'+"The total Cost for Minimum Spanning Tree is",
res,'\033[0m')

def printKruskal():
    print('###------------Kruskal---------------###')
    file1 = open('/content/gdrive/MyDrive/6114/Project2/samp1.txt',
'r')
    file2 = open('/content/gdrive/MyDrive/6114/Project2/samp2.txt',
'r')
    file3 = open('/content/gdrive/MyDrive/6114/Project2/samp3.txt',
'r')
    file4 = open('/content/gdrive/MyDrive/6114/Project2/samp4.txt',
'r')

    file1 = parseText(file1)
    file2 = parseText(file2)
    file3 = parseText(file3)
    file4 = parseText(file4)
```

```
    sample = [file1, file2, file3, file4]
    for file in sample:
        start = file.pop()
        start = start[0]
        stats = file.pop()
        file = file[0]
        numVert = stats[0]
        graph = kGraph(numVert)
        for l in file:
            graph.addEdge(ord(l[0])-65, ord(l[1])-65, int(l[2]))
        start_time = time.time()
        graph.KruskalMST()

        runtime = (time.time() - start_time) * 1000

        print('The runtime for this graph took: ', runtime, '
microseconds')

print('************************************************************')
        print()

printKruskal()
```

```
###------------Kruskal---------------###
This graph contained the following nodes:
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S']
Edge-Child             Weight

A - G                  2
C - I                  2
E - K                  2
J - O                  2
K - R                  2
B - H                  3
D - J                  3
F - L                  3
H - M                  3
I - N                  3
J - Q                  3
K - P                  3
L - Q                  3
L - M                  3
G - N                  4
G - R                  4
N - Q                  4
The total Cost for Minimum Spanning Tree is 49
The runtime for this graph took:  5.524396896362305  microseconds
************************************************************
```

```
This graph contained the following nodes:
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
Edge-Child             Weight

B - J                  11
A - I                  12
H - J                  13
G - I                  14
F - H                  15
E - G                  16
D - F                  17
C - E                  18
I - J                  21
The total Cost for Minimum Spanning Tree is 137
The runtime for this graph took:  2.285480499267578  microseconds
************************************************************

This graph contained the following nodes:
['A', 'B', 'C', 'D', 'E', 'F', 'G']
Edge-Child             Weight

D - E                  1
A - B                  2
G - H                  3
B - C                  6
E - F                  6
E - H                  7
C - D                  8
The total Cost for Minimum Spanning Tree is 33
The runtime for this graph took:  1.554250717163086  microseconds
************************************************************

This graph contained the following nodes:
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N']
Edge-Child             Weight

A - D                  1
E - G                  2
A - B                  3
C - H                  3
B - C                  4
C - I                  6
D - E                  8
G - F                  15
The total Cost for Minimum Spanning Tree is 42
The runtime for this graph took:  1.7290115356445312  microseconds
************************************************************
```

Problem 3: Finding Strongly Connected Components

Given a directed graph with vertices and edges. This graph may not be simple. Decompose this graph into Strongly Connected Components (SCCs) and print the components. You can use the same input format defined below.

##Strongly Connected Components A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. To find these components, one must first transverse a graph using either DFS or BFS with connected nodes being added to a list. The graph is then transposed and again transversed to find those nodes that are connected in both directions.

##My Implementation I again found that the graphs used in my implementation may have been too robust. I found also that most (and all in some cases) of my nodes were connected. I attribute this to the complex graphs that I chose for this implementation. Time complexity in this implementation was increased by the use of my lists in this case, as there were more lists per run than in the previous sections.

```
class cGraph:
    def __init__(self,vertices):
        self.V= vertices
        self.graph = defaultdict(list)

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

    def util(self,v,visited):
        l = []
        visited[v]= True
        #print(chr(v+65))
        l.append((chr(v+65)))
        for i in self.graph[v]:
            if visited[i]==False:
                l.append(self.util(i,visited))
        return l


    def fillOrder(self,v,visited, stack):
        visited[v]= True
        for i in self.graph[v]:
            if visited[i]==False:
                self.fillOrder(i, visited, stack)
        stack = stack.append(v)



    def getTranspose(self):
        g = cGraph(self.V)
```

```
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j,i)
        return g



    def printSCCs(self):

        stack = []
        visited =[False]*(self.V)
        for i in range(self.V):
            if visited[i]==False:
                self.fillOrder(i, visited, stack)
        gr = self.getTranspose()

        visited =[False]*(self.V)
        l = []
        while stack:
            i = stack.pop()
            if visited[i]==False:
                l.append(gr.util(i, visited))
        print(l)

def printConnected():
    print('###------------Connected----------------###')
    file1 = open('/content/gdrive/MyDrive/6114/Project2/samp1.txt',
'r')
    file2 = open('/content/gdrive/MyDrive/6114/Project2/samp2.txt',
'r')
    file3 = open('/content/gdrive/MyDrive/6114/Project2/samp3.txt',
'r')
    file4 = open('/content/gdrive/MyDrive/6114/Project2/samp4.txt',
'r')

    file1 = parseText(file1)
    file2 = parseText(file2)
    file3 = parseText(file3)
    file4 = parseText(file4)

    sample = [file1, file2, file3, file4]
    for file in sample:
        start = file.pop()
        start = start[0]
        stats = file.pop()
        file = file[0]
        numVert = stats[0]
        graph = cGraph(int(numVert))
        start_time = time.time()
        for l in file:
```

```
            graph.addEdge(ord(l[0])-65, ord(l[1])-65)
        print('The following nodes are strongly connected.')
        graph.printSCCs()
        runtime = (time.time() - start_time) * 1000

        print('The runtime for this calculation took: ', runtime, '
microseconds')

print('************************************************************')
        print()

printConnected()

###------------Connected---------------###
The following nodes are strongly connected.
[['A'], ['G'], ['B'], ['C'], ['D'], ['E'], ['F'], ['L'], ['K'], ['J'],
['I'], ['N'], ['Q'], ['H'], ['O'], ['R'], ['M'], ['P']]
The runtime for this calculation took:  1.4116764068603516
microseconds
************************************************************

The following nodes are strongly connected.
[['A'], ['B'], ['C'], ['D'], ['F'], ['E'], ['G'], ['H'], ['I'], ['J']]
The runtime for this calculation took:  0.5304813385009766
microseconds
************************************************************

The following nodes are strongly connected.
[['A'], ['B'], ['C'], ['D'], ['E'], ['F'], ['G'], ['H']]
The runtime for this calculation took:  0.07271766662597656
microseconds
************************************************************

The following nodes are strongly connected.
[['A'], ['B'], ['C'], ['H'], ['D'], ['E'], ['G'], ['F'], ['I']]
The runtime for this calculation took:  0.06341934204101562
microseconds
************************************************************
```