PROJECT 2

Graph Algorithms and Related Data Structures

Algorithms and Data Structures

ITCS 6114/8114-Fall 2021

UNIVERSITY OF NORTH CAROLINA AT CHARLOTTE

SUBMITTED BY:

SAI PRATHYUSHA POTU
Student Id: 801261324
Email id: spotu1@uncc.edu

THARANI KUMARESAN
Student Id: 801265490
Email id: tkumares@uncc.edu

PROBLEM 1:

SINGLE-SOURCE SHORTEST PATH ALGORITHM

Assume that G is 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..., e_{k-1}, then length of P, denoted as w(P), is defined as w(P)= $\sum_{i=0}^{k-1} w(ei)$

The distance of a vertex v from a vertex s is the length of a shortest path between s and v, denoted as d(s, v). If d(s, v) = +∞ if no path exists.

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a Greedy method of computing the shortest distances of all the vertices from a given source vertex s.

Assumption: Graphs are connected. Edges are undirected or directed, edge weights are nonnegative, i.e., w(e) ≥ 0

Edge Relaxation:

```
RELAX (u, v, w)
if v.d > u.d + w(u, v)
   v.d = u.d + w(u, v)
   v.π = u
```

Steps for finding shortest path using Dijkstra's algorithm:

- For all four inputs, created a graph with an adj list for all edges and their weight depending on whether the graph is directed or undirected.
- Created a shortest-path tree set visited that keeps track of the vertices in the shortest path tree.
- All vertices in the input graph were given a distance value. In the beginning, we set all distance values to infinite and set the distance value for the source vertex to 0 to be chosen first.
- Created a data structure for the parent that keeps track of the path taken and displays it in the output.
- Until all vertices have been visited, a vertex with the shortest distance has been chosen and added to visited.
- The distance value of all the adjacent vertices has been updated.

Algorithm:

DIJKSTRA-SHORT (G, w, s)

1 INITIALIZE (G, s)

```
S ← ϕ
Q ← V
while Q ≠ ϕ
      do u ← EXTRACT-MIN (Q)
      S ← S ∪{u}
      for each vertex v ∈ Adj [u]
           do RELAX (u, v, w)
```

Complexity Analysis of Dijkstra's algorithm:

When implemented with queues (which is a preferable approach), Dijkstra's algorithm has the following execution times:

Worst case time complexity: O(E+VlogV)
Average case time complexity: O(E+VlogV)
Best case time complexity: O(E+VlogV)
Space complexity: O(V)

Where E is the number of edges and V is the number of vertices in the graph.

We have used python's default dictionary to build an adjacency list to represent the graph. Time Complexity of this implementation is O(V2). As the relaxation of vertex happens for the minimum picked vertex weight. And have not used a priority Queue so the final complexity for my code would be O(E+V^2). V^2 for the Vertex relaxation and it happens for every edge.

Data Structures Used:

Graph, Adjacency List, python List, Python Default dictionary are the data Structures used in this program to find the shortest path.

PROBLEM 2:

MINIMUM SPANNING TREE

A spanning tree is a subset of Graph G, that covers all of the vertices with the minimum possible number of edges. Hence, there are no cycles in a spanning tree, and it cannot be disconnected. We can conclude from this definition that every connected and undirected Graph G contains at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices. Spanning trees have n-1 edges. Where, n is the number of vertices.

KRUSKAL’S ALGORITHM

Kruskal’s algorithm is the greedy method to find the minimum spanning tree for the given graph. Kruskal's algorithm treats each node as a separate tree, connecting them only if they have the lowest cost compared to all other possibilities.

Steps to create MST using Kruskal’s algorithm:

- It begins with a forest, with each vertex representing a tree (a single node tree).
- It finds a safe edge to add to the growing forest by finding the edge (u, v) with the least weight among all the edges that connect any two trees in the forest.
- Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.
- At the end of the algorithm: We are left with one cloud that encompasses the MST A tree T which is our MST

Algorithm:

```
KRUSKAL-MST (G, w)
A = ϕ
for each vertex v ∈ G.V
        MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, in nondecreasing order by weight
        if FIND-SET(u) ≠ FIND-SET(v)
                A=A U {(u, v)}
                UNION (u, v)
return A
```

Complexity Analysis of Kruskal’s algorithm:

O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE +ELogV) time. The value of E can be at most O(V2), so O(LogV) are O(LogE)same. Therefore, overall time complexity is O(ElogE) or O(ElogV).
Here E is number of edges in the graph and V is the number of vertices.

Data Structures used:

Graph, python List, Python default dictionary are the data Structures used in this program to find the MST using Kruskal’s algorithm.

PROBLEM 3:
STRONGLY CONNECTED COMPONENTS:
Algorithm:

- Run DFS on D and every time DFS finishes expanding a vertex v, add v to a stack S.
- Let D^R be the graph D with the direction of all the edges reversed.
- While the stack S is non-empty, pop a vertex v from S and perform DFS on D^R starting at v.
- Let U be the set of all the vertices visited by DFS(v). Then U is the strongly connected component containing v.
- Remove U from S and repeat.

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point.

Sequence of picking vertices as starting points of DFS:

There is no direct way to get this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we ensure that the finish time of a vertex that connects to other SCCs (other than its own SCC) will always be greater than the finish time of vertices in the other SCC.

Time Complexity:

The above algorithm calls DFS finds reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using an adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simply traverse all adjacency lists.

Data Structures used:

Graph, Adjacency List, python List, Python Default dictionary are the data Structures used in this program to find the SCCs.

Code:

PROBLEM 1:

Single-source Shortest Path Algorithm

```
from collections import defaultdict
import sys
import time
Dijkstra_times = []

class Graph:
    def __init__(self, directed = False, vertices = None):
        self.directed = directed
        self.graph = defaultdict(list)
        self.vertices = vertices

    def addEdge(self, frm, to, w):
        self.graph[frm].append([to, w])

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

    def getEdges(self):
        print(self.graph.keys())
        print(self.graph.values())


    def find_min_dist(self, distance, visit):
        min_dist = float('inf')
        index = -1
        for key in self.graph.keys():
            if visit[key] is False and distance[key] < min_dist:
                min_dist = distance[key]
                index = key
        return index
```

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

    def solution(self, distance, root, source):
        print('{} \t\t{} \t {}'.format('Vertex', 'Path Cost', 'Path'))

        for key in self.graph.keys():
            if key == source:
                continue
            if  distance[key]==float("inf"):
                continue
            print('{} -> {} \t\t {} \t\t {}'.format(chr(source+65), chr(key+65), distance[key], chr(source+65)), end=' ')
            tmp = []
            self.path(root, key)
            print()

    def dijkstra(self, source):
        visited = {i: False for i in self.graph}
        distance = {i: float('inf') for i in self.graph}
        root = {i: None for i in self.graph}

        distance[source] = 0

        # find shortest path for all vertices
        for i in range(len(self.graph) - 1):
            start = self.find_min_dist(distance, visited)
            visited[start] = True
            for vertex, w in self.graph[start]:

                if visited[vertex] is False and distance[start] + w < distance[vertex]:
                    distance[vertex] = distance[start] + w
                    root[vertex] = start
        return root, distance
```

```python
if __name__ == '__main__':
    directed = False
    n_inputs = 0

    while n_inputs < 4:
        filepath = "graph" + str(n_inputs) + ".txt"
        inputFile = open(filepath,'r')
        n_line = 0
        source = sys.maxsize
        print('\033[1m'+"Single source Shortest paths for"+'\033[0m'+" {} is: ".format(filepath))
        print("=======================================================================")

        for line in inputFile.readlines():
            tmp = line.split()
            if n_line == 0:
                print("Number of Vertices provided: {}".format(int(tmp[0])))
                print("Number of Edges provided: {}".format(int(tmp[1])))
                d_ud = tmp[2]
                if d_ud == "D":
                    directed = True
                    print("The Graph is: DIRECTED")
                else:
                    directed = False
                    print("The Graph is: UN DIRECTED")
                G = Graph(directed)
            elif len(tmp) == 1:
                source = ord(tmp[0])-65
            else:
                G.addEdge(ord(tmp[0])-65, ord(tmp[1])-65, int(tmp[2]))

            n_line += 1

        print("Start or Source provided: {}".format(chr(source+65)))
        st = time.time()
        root, distance = G.dijkstra(source)
```

```python
        #print(parent, dist)
        print("-----------------------------------------")
        print("Dijkstra's Algorithm results:")
        print("-----------------------------------------")
        G.solution(distance, root, source)
        n_inputs += 1
        print("-----------------------------------------------------------------------")
        print('\033[1m'+"This input's execution time is :" +'\033[0m', (time.time() - st)*1000,"\n")
        print("-----------------------------------------------------------------------\n")
```

The Results for the above graphs are: -

```
Single source Shortest paths for graph0.txt is:
=======================================================================
Number of Vertices provided: 9
Number of Edges provided: 18
The Graph is: UN DIRECTED
Start or Source provided: A
-----------------------------------------
Dijkstra's Algorithm results:
-----------------------------------------
Vertex          Path Cost        Path
A -> C           9               A C
A -> D           13              A D
A -> B           22              A B
A -> H           56              A B H
A -> F           54              A C F
A -> E           46              A D E
A -> I           53              A D I
A -> G           69              A D E G
```

This input's execution time is : 0.9992122650146484

Single source Shortest paths for graph1.txt is:

==

Number of Vertices provided: 10
Number of Edges provided: 14
The Graph is: UN DIRECTED
Start or Source provided: A

Dijkstra's Algorithm results:

Vertex	Path Cost	Path
A -> B	3	A B
A -> C	1	A C
A -> D	5	A C D
A -> E	7	A C D E
A -> F	9	A C D E F
A -> G	13	A C D E F G
A -> H	10	A C D E H
A -> I	13	A C D E F J I
A -> J	11	A C D E F J

This input's execution time is : 0.9989738464355469

Single source Shortest paths for graph2.txt is:

==

Number of Vertices provided: 10
Number of Edges provided: 20
The Graph is: DIRECTED
Start or Source provided: A

Dijkstra's Algorithm results:

Vertex	Path Cost	Path
A -> B	4	A B
A -> I	15	A B I
A -> D	13	A B D
A -> C	20	A B D C
A -> E	15	A B D E
A -> F	19	A B D F
A -> G	22	A B D E G
A -> H	19	A B D E H
A -> J	3	A J

This input's execution time is : 0.9922981262207031

```
Single source Shortest paths for graph3.txt is:
========================================================================
Number of Vertices provided: 10
Number of Edges provided: 20
The Graph is: DIRECTED
Start or Source provided: A
------------------------------------------
Dijkstra's Algorithm results:
------------------------------------------
Vertex          Path Cost       Path
A -> B           4              A B
A -> C           15             A B C
A -> D           13             A B D
A -> E           15             A B D E
A -> F           19             A B D F
A -> G           22             A B D E G
A -> H           19             A B D E H
A -> I           29             A B D E H I
A -> J           3              A J
------------------------------------------------------------------------
This input's execution time is : 1.9979476928710938

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

Graph Directed Graph's and Un-Directed Graph's Runtimes:

Graph0: 0.9992122650146484
Graph1: 0.9989738464355469
Graph2: 0.9922981262207031
Graph3: 1.9979476928710938

Runtimes in the above table are in Micro seconds

PROBLEM 2:

Minimum Spanning Tree Algorithm

```
class Graph:

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


    def addEdge(self, root, vertex, cost):
        self.graph.append([root, vertex, cost])

    def get_rootnode(self, root_node, v):
        if root_node[v] == v:
            return v
        return self.get_rootnode(root_node, root_node[v])

    def union_of_sets(self, root_node, rank, x, y):
        root_x = self.get_rootnode(root_node, x)
        root_y = self.get_rootnode(root_node, y)

        if rank[root_x] < rank[root_y]:
            root_node[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            root_node[root_y] = root_x

        else:
            root_node[root_y] = root_x
            rank[root_x] += 1
```

```
    def Kruskal(self):
        result = []
        idx_e = 0
        idx_r = 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.vertices):
            parent.append(node)
            rank.append(0)
        while idx_e < self.vertices - 1:

            root, vertex, cost = self.graph[idx_r]
            idx_r += 1
            s1 = self.get_rootnode(parent, root)
            s2 = self.get_rootnode(parent, vertex)
            if s1 != s2:
                idx_e += 1
                result.append([root, vertex, cost])
                self.union_of_sets(parent, rank, s1, s2)

        print('\033[1m' + "Edge Selected \t\t Weight" + '\033[0m')
        print()
        total_cost=0
        for x, y, z in result:
            print(chr(x + 65), "->", chr(y + 65), "\t\t\t  ", z)
            total_cost += z
        print("*****************************************************************************************")
        print('\033[1m'+"The total Cost for Minimum Spanning Tree is", total_cost,'\033[0m')
```

```python
if __name__ == '__main__':
    n_inputs = 0
    print()
    while (n_inputs < 4):
        print("=====================================================================================")
        filepath = "graph" + str(n_inputs) + ".txt"
        inputFile = open(filepath, "r")
        print('\033[1m'+"The Minimum Spanning Tree using Kruskal's Algorithm for"+'\033[0m'+" {} :" .format(filepath))
        print("=====================================================================================")

        n_line = 0
        for line in inputFile.readlines():
            tmp = line.split()
            if n_line == 0:
                no_of_vertices = int(tmp[0])
                graph = Graph(no_of_vertices)
            elif len(tmp) == 1:
                pass
            else:
                graph.addEdge(ord(tmp[0]) - 65, ord(tmp[1]) - 65, int(tmp[2]))
            n_line = n_line + 1

        print("---------------------------------------")
        print("Kruskal's Algorithm Results")
        print("---------------------------------------")
        st = time.time()
        graph.Kruskal()
        n_inputs += 1
        print("------------------------------------------------------------------------------------")
        print('\033[1m'+"This input's execution time is :" +'\033[0m', (time.time() - st)*1000)
        print("----------------------------------------------------------------------------------\n")
```

The Results for the above graphs are: -

```
======================================================================
The Minimum Spanning Tree using Kruskal's Algorithm for graph0.txt :
======================================================================
---------------------------
Kruskal's Algorithm Results
---------------------------
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
-------------------------------------------------------
The total Cost for Minimum Spanning Tree is 148
-------------------------------------------------------
This input's execution time is : 1.992940902709961
-------------------------------------------------------
```

```
==========================================================================
The Minimum Spanning Tree using Kruskal's Algorithm for graph1.txt :
==========================================================================
----------------------------
Kruskal's Algorithm Results
----------------------------
Edge Selected          Weight

A -> C                   1
F -> H                   1
B -> C                   2
D -> E                   2
E -> F                   2
I -> J                   2
J -> F                   2
B -> D                   3
G -> H                   3
----------------------------------------------------------
The total Cost for Minimum Spanning Tree is 18
----------------------------------------------------------
This input's execution time is : 1.0082721710205078
----------------------------------------------------------

==========================================================================
The Minimum Spanning Tree using Kruskal's Algorithm for graph2.txt :
==========================================================================
----------------------------
Kruskal's Algorithm Results
----------------------------
Edge Selected          Weight

F -> C                   1
I -> J                   1
D -> E                   2
H -> F                   2
H -> J                   2
A -> J                   3
A -> B                   4
E -> H                   4
E -> G                   7
----------------------------------------------------------
The total Cost for Minimum Spanning Tree is 26
----------------------------------------------------------
This input's execution time is : 0.0
----------------------------------------------------------

==========================================================================
The Minimum Spanning Tree using Kruskal's Algorithm for graph3.txt :
==========================================================================
----------------------------
Kruskal's Algorithm Results
----------------------------
Edge Selected          Weight
```

```
F -> C                  1
I -> J                  1
D -> E                  2
H -> F                  2
H -> J                  2
A -> J                  3
A -> B                  4
E -> H                  4
E -> G                  7
------------------------------------------------------
The total Cost for Minimum Spanning Tree is 26
------------------------------------------------------
This input's execution time is :  0.0
------------------------------------------------------
```

PROBLEM 3:

Finding Strongly Connected Components

```python
from collections import defaultdict
import sys
import time


class Graph:

    def __init__(self,vertices):
        self.V = vertices
        self.graph = defaultdict(list)
        self.scc_count = 0

    # adds edge to the graph
    def addEdge(self,root,vertex):
        self.graph[root].append(vertex)

    # helper function for DFS
    def helper_DFS(self,vertex,visited, st):
        visited[vertex]= True
        #print(vertex, end=" ")
        st.append(vertex)

        for i in self.graph[vertex]:
            if visited[i]==False:
                self.helper_DFS(i,visited, st)
                self.scc_count -= 1
        self.scc_count += 1
        return st, self.scc_count
```

```python
    # helper function that marks current node as visited and traverse all the adjacent nodes to current node
    def helper(self, vertex, visited, stack):
        visited[vertex]= True

        for node in self.graph[vertex]:
            if visited[node]==False:
                self.helper(node, visited, stack)
        stack = stack.append(vertex)


    # Function that returns transposed graph
    def get_transpose(self):
        gr = Graph(self.V)

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


    # Function that finds strongly connected components in a directed graph
    def strongly_connected_components(self):

        stack = []
        # first DFS all visited as False
        visited =[False]*(self.V)
        # fill vertices for first DFS
        for i in range(self.V):
            if visited[i]==False:
                self.helper(i, visited, stack)

        grph = self.get_transpose()
```

```python
        # second DFS all visited as False
        visited =[False]*(self.V)

        # Process all vertices in order of Stack
        while stack:
            i = stack.pop()
            if visited[i]==False:
                tmp = []
                scc,sccCount = grph.helper_DFS(i, visited,tmp)
                print("Strongly Connected Components ",sccCount," = ", scc)
                #print("\n")


if __name__ == '__main__':
    n_inputs = 0
    print()
    while (n_inputs < 4):
        print("=========================================================================================")
        filepath = "graph" + str(n_inputs) + "_SCC.txt"
        inputFile = open(filepath, "r")
        print('\033[1m'+"Strongly Connected Components in the given "+'\033[0m'+" {} :" .format(filepath))
        print("=========================================================================================")

        n_line = 0
        for line in inputFile.readlines():
            tmp = line.split()
            if n_line == 0:
                no_of_vertices = int(tmp[0])
                graph = Graph(no_of_vertices)
            elif len(tmp) == 1:
                pass
            else:
                #graph.add_edge(temp[0],temp[1])
                graph.addEdge(ord(tmp[0]) - 65, ord(tmp[1]) - 65)
            n_line = n_line + 1
```

```
        print("*************************************")
        print("SCC Algorithm Results")
        print("*************************************")
        st = time.time()
        graph.strongly_connected_components()
        n_inputs += 1
        print("------------------------------------------------------------------------------------")
        print('\033[1m'+"This input's execution time is :" +'\033[0m', (time.time() - st)*1000)
        print("------------------------------------------------------------------------------------\n")
```

The Results for the above graphs are: -

```
==============================================================
Strongly Connected Components in the given  graph0_SCC.txt :
==============================================================
--------------------------
SCC Algorithm Results
--------------------------
Strongly Connected Components  1  =  [0, 2, 1, 4, 3, 5, 7, 6]
Strongly Connected Components  2  =  [9]
Strongly Connected Components  3  =  [8]
-------------------------------------------------------------
This input's execution time is : 0.0
-------------------------------------------------------------

==============================================================
Strongly Connected Components in the given  graph1_SCC.txt :
==============================================================
--------------------------
SCC Algorithm Results
--------------------------
Strongly Connected Components  1  =  [0, 2, 3, 1, 4, 5, 7]
Strongly Connected Components  2  =  [6]
Strongly Connected Components  3  =  [8]
Strongly Connected Components  4  =  [9]
-------------------------------------------------------------
This input's execution time is : 0.0
-------------------------------------------------------------

==============================================================
Strongly Connected Components in the given  graph2_SCC.txt :
==============================================================
--------------------------
SCC Algorithm Results
--------------------------
Strongly Connected Components  1  =  [6]
Strongly Connected Components  2  =  [0, 2, 1, 4, 3, 5, 7]
Strongly Connected Components  3  =  [8]
Strongly Connected Components  4  =  [9]
-------------------------------------------------------------
This input's execution time is : 0.0
-------------------------------------------------------------

==============================================================
Strongly Connected Components in the given  graph3_SCC.txt :
==============================================================
```

```
--------------------------
SCC Algorithm Results
--------------------------
Strongly Connected Components  1  =  [0, 2, 1, 4, 3, 5, 7, 6]
Strongly Connected Components  2  =  [8]
Strongly Connected Components  3  =  [9]
-------------------------------------------------------------
```
This input's execution time is : 0.0

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