PROJECT 2

Graph Algorithms and Related Data Structures

Single-source shortest path algorithm , Minimum Spanning Tree (MST)

and Strongly Connected Components (SSCs)

ITCS 6114
Algorithms and Data Structures

Prof. Ahmed Dewan

April 2, 2023

Project Members

Prachi Shende 801306878

Yash Agrawal 801316596

UNIVERSITY OF NORTH CAROLINA AT CHARLOTTE

	INDEX
1.	Shortest Path Algorithm
2.	Dijikstra's Algorithm
3.	Data Structures used and Runtime of Dijiktra's Algorithm
4.	Minimum Spanning Tree and Algorithm Spanning Tree
5.	Kruskal's Algorithm
6.	Data Structures used and Runtime of Kruskal's Algorithm
7.	Finding Strongly Connected Graphs
8.	Source code used in project
9.	Output

1. SHORTEST PATH ALGORITHM

The Bellman-Ford algorithm, also celebrated as the single source shortest path algorithm (positive or negative for arbitrary weight), is used to calculate the shortest path between two points. The main distinction between this approach and the Dijkstra algorithm is that the Dijkstra algorithm may handle negative weight, but we can easily do it here.

Let G be a connected 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,e2..., *e*k-1 then length of *P*, denoted as *w*(*P*), is defined as (*P*)= Σi=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 (***s***, ***v***), d (s, v) = +∞ if no path exists.

Algorithm

INITIALIZE-SINGLE-SOURCE (G, s) for each vertex v ∈G, V

dod[v] = ∞ π[v] = NIL

d[s] = 0

Path Weight Calculation:

w(p)= Σw (vi-1, vi) where:

The weight of the shortest path from vertex u to vertex v can be calculated by, δ(u,v)=min{w(p): p is a path from u to v.

Edge Relaxation:

Edge relaxation is a technique that is used by the shortest path algorithms. It continually decreases the arbitrary cap of an actual quickest route for each vertex until the optimal solution equals the length of the shortest path.

In edge relaxation, we keep an attribute d[v] for each vertex v that is the reactive routing estimate. It is the maximum length of the shortest path from resource s to destination v.

Consider an edge e = (u,v) such that u is the vertex most recently added to the cloud v is not in the cloud

The relaxation of edge updates distance as follows: Relax (u, v, w)

if d[v] > d[u] + w(u, v)

d[v] = d[u] + w(u, v) π[v] = u

2. DIJIKSTRA'S ALGORITHM

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

This algorithm finds the shortest distance from a source vertex s in V given the graph G= (V,E) to every vertex in V, hence this problem is sometimes called the single-source shortest paths problem.

Assumption:

1) Graphs are connected.
2) Edges are undirected or directed.
3) Edge weights are non-negative.

Dijkstra's Algorithm Complexity

Time Complexity: O(E Log V)
where, E is the number of edges and V is the number of vertices.
Space Complexity: O(V)

- Initialization of all nodes with distance "infinite"; initialization of the starting node with 0
- Marking of the distance of the starting node as permanent, all other distances as temporarily.
- Setting of starting node as active.
- Calculation of the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges.
- 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.
- Setting of the node with the minimal temporary distance as active. Mark its distance as permanent.
- Repeating of steps 4 to 7 until there aren't any nodes left with a permanent distance, which neighbours still have temporary distances.
-

Algorithm

DIJKSTRA (G, w, s) INITIALIZE-SINGLE-SOURCE (G, s)

S=∅

Q = G.V

while Q ≠∅ do

u = EXTRACT-
MIN(Q)S = S ∪ {u}
for each vertex v ∈G.
Adj[u] doRELAX
(u, v, w)

3. DATA STRUCTURES USED AND RUNTIME FOR DIJIKSTRA'S ALGORITHM

1. Dijkstra's approach is used to discover the quickest route in both directional and undirected weighted graphs.
2. Array List, Adjacency List, Graphs, Hash Map, Priority Queue, and Tree Set are the data structures needed to implement Dijkstra's method for determining the shortest in directed and undirected graphs.
3. For this approach, we utilized default to construct an adjacent list to represent the graph.
4. Time Complexity of this implementation is O(V2). As the relaxation of vertex happens for the minimum picked vertex weight. So, this step will take O(V2) Runtime.
5. For the priority queue in Dijkstra's method, we employed a binary min-heap. This takes O (m log n) time, where m denotes the number of edges and n is the number of vertices.
6. Each EXTRACT-MIN operation takes O (log n) time whereas each RELAX operation takes O (log n) time for at most O (m) such operations.
7. Therefore, the total running time for Dijkstra's algorithm would be, O ((n + m) log n).
If all vertices v ∈G are reachable from the source s, then the running time would be O (m
log n)

4. MINIMUM SPANNING TREE ALGORITHM

1. A traversal of a graph g is a tree that has all the graph G's vertices linked by certain edges.

2. There can be more than one spanning tree in a graph. There are n-1 edges in the resultant spanning tree for a graph with n vertices.

3. MST (Minimum Spanning Tree): A minimal spanning tree is a single spanning tree that has less weighted edges than all other spanning trees in a graph G. (MST).

4. In a graph G, there exists a weight for each edge that connects the corresponding vertices.

5. An MST of a graph G is a subgraph which connects every other vertex in the graph with a total minimum weight of edges.

6. A minimal spanning tree may be computed using Prim's and Kruskal's algorithms. Kruskal's technique is used in this project to discover the shortest spanning tree and the actual cost (weight of the final tree edges).

There are two ways to find MST:

1. Kruskal’s Algorithm
2. Prim’s Algorithm

For this project, we have used Kruskal’s Algorithm.

5. KRUSKAL'S ALGORITHM

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first at the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
3. If the cycle is not formed, include this edge. Else, discard it.
4. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

Algorithm

MST-KRUSKAL (G, w) A=∅
for each vertex v ∈ G.V

doMAKE-SET (v)
//Sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight do

if FIND-SET (u) ≠ FIND-SET (v) then
    A = A ∪ {(u, v)}
    UNION (u, v)
return A

6. DATA STRUCTURES USED AND RUNTIME FOR KRUSKAL'S ALGORITHM

1. Array List, Graphs, Hash Map, Map, Navigable Set, and Tree Set are the data structures needed to implement Kruskal's Algorithm for determining the Minimum Spanning Tree (MST).
2. In Kruskal's Algorithm, edges of graph G are sorted into non-decreasing order by weight

3. Time taken for this sorting operation is, O (m log m) where, m represents edges.

4. The runtime is determined by how we do SET operations. The runtime is presented here on the premise that the disjoint-set-forest implementation is utilized because it is the quickest asymptotically.

5. The function MAKE-SET takes O(n) time to execute for n vertices.

6. The function FIND-SET takes O(m) time to execute form edges.

7. The function UNION takes O(n) time for n vertices.

8. As a result, the complete running time for executing Kruskal's method to determine the minimal spanning tree is O (m log m), where m is the number of edges.

9. If|E|<V2log|m|=O(logn)Thentherunningtimewouldbe O(mlogn)whichis asymptotically same as Prim's Algorithm.

7. STRONGLY CONNECTED GRAPHS

A directed graph is strongly connected if there is a path between all pairs of vertices.
That is, a path exists from the first vertex in the pair to the second, and another path exists from the second vertex to the first.
A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.
In this algorithm, DFS algorithm is used to get the finish time of each vertex.
DFS search produces a DFS tree/forest. Strongly Connected Components form subtrees of the DFS tree.

If we can find the head of such subtrees, we can store all the nodes in that subtree and that will be one SCC.
There is no back edge from one strongly connected component.
To find its strongly connected components, in linear time (that is, $\Theta(V + E)$)

***Algorithm*:**

STRONGLY CONNECTED GRAPH

Pick a vertex V in G Perform
DFS for V in G

    If a W not visited print no

Let G' be G with edges reversed

If a W not visited print no

    Else print yes

SOURCE CODE AND OUTPUT IN PROJECT

DIJIKSTRA'S ALGORITHM

```
class Dikstra:

   def __init__(self, directed=False):

       self.graph = defaultdict(list)

       self.directed = directed

   def addEdge(self, frm, to, weight):

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

       if self.directed is False:

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

       elif self.directed is True:

           self.graph[to] = self.graph[to]

   def find_min(self, distance, visited):

       minimum = float('inf')

       index = -1

       for v in self.graph.keys():

           if visited[v] is False and distance[v] < minimum:

               minimum = distance[v]

               index = v

       return index

   def dikstra(self, src):

       visited = {i: False for i in self.graph}

       distance = {i: float('inf') for i in self.graph}
```

```
        parent = {i: None for i in self.graph}

        distance[src] = 0

        for i in range(len(self.graph) - 1):

            u = self.find_min(distance, visited)

            visited[u] = True

            for v, w in self.graph[u]:

                if visited[v] is False and distance[u] + w < distance[v]:

                    distance[v] = distance[u] + w

                    parent[v] = u

        return parent, distance

    def printPath(self, parent, v):

        if parent[v] is None:

            return

        self.printPath(parent, parent[v])

        print(chr(v+65),end=" ")

    def printShortestpath(self, distance, parent, src):

        for i in self.graph.keys():

            if i == src:

                continue

            if i < 0:

                break

            if distance[i]==float("inf"):

                continue

                    print('Cost of shortest path from {} -> {} is \t{} and path is
\t{}'.format(chr(src+65), chr(i+65), distance[i], chr(src+65)), end=' ')

            self.printPath(parent, i)
```

```
            print()
```

OUTPUT FOR DIJIKSTRA'S (DIRECTED GRAPH)

**********Shortest Path Dikstra Algorithm**********

Shortest path for graph 0

Number of Vertices: 6

Number of Edges: 10

The Graph is DIRECTED graph

Source vertex: A

Cost of shortest path from A -> B is 1 and path is A B

Cost of shortest path from A -> C is 2 and path is A C

Cost of shortest path from A -> D is 3 and path is A C D

Cost of shortest path from A -> E is 3 and path is A B E

Cost of shortest path from A -> F is 6 and path is A C D F

Runtime for Dijkstras Algorithm: 1 Seconds

OUTPUT FOR DIJIKSTRA'S (UNDIRECTED GRAPH)

Shortest path for graph 5

Number of Vertices: 10

Number of Edges: 11

The Graph is UNDIRECTED graph

Source vertex: B

Cost of shortest path from B -> A is 3 and path is B A

Cost of shortest path from B -> D is 4 and path is B A D

Cost of shortest path from B -> E is 8 and path is B A E

Cost of shortest path from B -> C is 4 and path is B C

Cost of shortest path from B -> F is 14 and path is B A E F

Cost of shortest path from B -> G is 21 and path is B A E F G

Cost of shortest path from B -> H is 23 and path is B A E F H

Cost of shortest path from B -> I is 29 and path is B A E F G I

Cost of shortest path from B -> J is 34 and path is B A E F H J

Runtime for Dijkstras Algorithm: 1 Seconds

KRUSKAL'S ALGORITHM

```
class Kruskal:

  def __init__(self, vertices):

      self.V = vertices

      self.graph = []

  def addEdge(self, u, v, w):

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

  def find(self, parent, i):

      if i > len(parent):

          i = parent[len(parent)-1]

      if parent[i] == i:

          return i

      return self.find(parent, parent[i])

  def union(self, parent, rank, x, y):

      x_src = self.find(parent, x)

      y_src = self.find(parent, y)

      if rank[x_src] < rank[y_src]:

          parent[x_src] = y_src

      elif rank[x_src] > rank[y_src]:

          parent[y_src] = x_src

      else:
```

```
            parent[y_src] = x_src

            rank[x_src] += 1

    def KruskalMST(self):

        result = []

        e =i= 0

        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []

        rank = []

        for node in range(self.V):

            parent.append(node)

            rank.append(0)

        while e < self.V - 1:

            if i >= len(self.graph):

                break

            src, dest, weight = self.graph[i]

            i = i + 1

            x = self.find(parent, src)

            y = self.find(parent, dest)

            if x != y:

                e = e + 1

                result.append([src, dest, weight])

                self.union(parent, rank, x, y)

        print("Edge Selected \t Weight" )

        res=0

        for src, dest, weight in result:
```

```
        print("   ",chr(src + 65), "->", chr(dest + 65), '       ', weight)

        res+=weight

    print("The total Cost for Minimum Spanning Tree is", res)
```

OUTPUT FOR KRUSKAL'S ALGORITHM

********** Kruskal Algorithm **********

Minimum Spanning Tree for graph 8

Edge Selected Weight

C -> B 1

J -> I 1

G -> I 1

L -> Q 1

L -> R 1

A -> B 2

D -> E 2

I -> N 2

J -> O 2

J -> S 2

M -> L 2

A -> D 3

H -> N 3

N -> M 3

K -> E 4

C -> H 4

B -> F 6

K -> P 6

The total Cost for Minimum Spanning Tree is 46

Runtime for kruskal Algorithm: 1 Seconds

STRONGLY CONNECTED GRAPHS

```
class SCC_class(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)



   def from_dict(cls, edge_dict):

       return cls((k, v) for k, vs in edge_dict.items() for v in vs)



class StrongConnectedComponents(object):

   def strong_connect(self, head):

       lowlink, count, stack = self.lowlink, self.count, self.stack

       lowlink[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)

               lowlink[head] = min(lowlink[head], lowlink[tail])

           elif count[tail] < count[head]:

               if tail in self.stack:
```

```
                lowlink[head] = min(lowlink[head], count[tail])

    if lowlink[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.lowlink = 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
```

OUTPUT FOR STRONGLY CONNECTED GRAPHS

**********Strongly Connected Components**********

Strongly connected components for graph0

no.of vertices: 6

F E D C B A

Runtime for calculating SCC: 1 Seconds

Strongly connected components for graph1

no.of vertices: 11

K J I F H G E D C B A

Runtime for calculating SCC: 1 Seconds