NAME: MD. NAYEEM MOLLA

Subject: **Algorithm Design and Analysis**

Experiment V: Greedy Algorithm

1. Purpose

Apply the greedy algorithm to graph theory problem and implement it in programming.

2. Main requirements

(1) Write and debug the minimum spanning tree program with Prim's or Kruskal's algorithm.

(2) Write and debug the Dijkstra's algorithm.

3. Instrument and equipment

PC-compatible (language-free).

4. Algorithm's principles

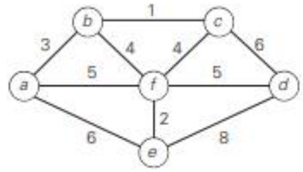
4.1 Minimum Spanning Tree

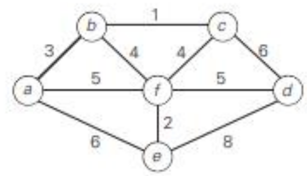
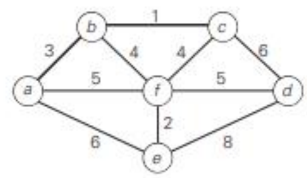
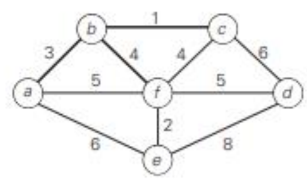
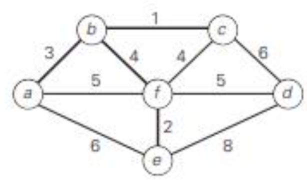
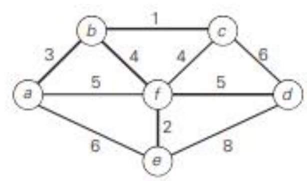
✧ **Pseudocode of Prim's Algorithm**

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
 find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
 such that v is in V_T and u is in $V - V_T$
 $V_T \leftarrow V_T \cup \{u^*\}$
 $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

✧ **Given example**



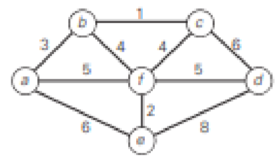
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b**(**a**, **3**) c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c**(**b**, **1**) d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f**(**b**, **4**)	
f(b, 4)	d(f, 5) **e**(**f**, **2**)	
e(f, 2)	**d**(**f**, **5**)	
d(f, 5)		

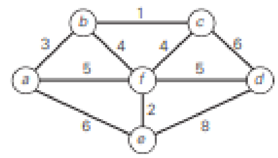
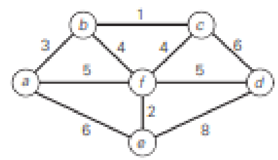
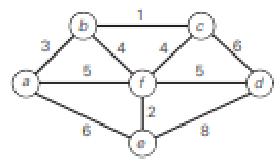
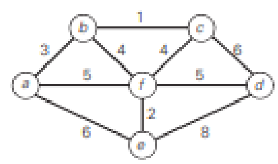
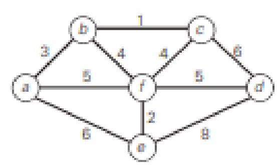
✧ **Pesudocode of Kruskal's Algorithm**

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

✧ **Given example**



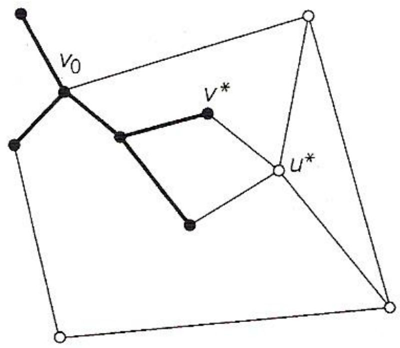
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

4.2 Single-Source Shortest-Path Problem

A known n node has a directional figure G= (V, E) and the edge's weight function c(e), in order to get the shortest path from a specified node v0 in G to all other nodes.

✧ **Algorithm Strategy**

Construct these shortest paths on a strip-by-line basis; assuming that the shortest path of i is constructed, the path to be constructed below should be the next shortest minimum length path. The greedy way to generate the shortest paths from v0 to all other nodes is to generate them in a non-descending order according to the path length.



✧ **Notes:**

Attach two markers to each vertex: the number d indicates the shortest path length to date, and the other marks the vertex's parent in the currently constructed tree.

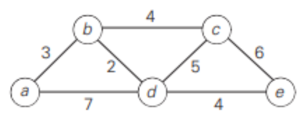
After assuring the vertex v* added to the tree, you have to do two more operations: move u* from the edge collection to the tree vertex aggregate; For the every edge vertex u left, If you are connected by an edge with a weight of w(u*,u) and u*, when du*+w(u*,u) < du, modify the marks update to u* and du*+w(u*,u).

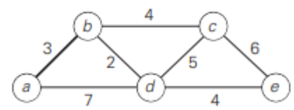
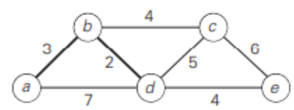
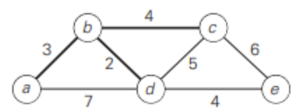
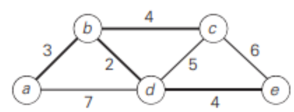
✧ **Pseudocode**

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
    Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \varnothing$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow DeleteMin(Q)$ //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - V_T$ that is adjacent to u^* **do**
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease(Q, u, d_u)

✧ **Given example**



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

Source Code:

```
# Python program for Kruskal's algorithm to find by Md nayeem MOlla 381
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph

from collections import defaultdict


# Class to represent a graph


class Graph:

    def __init__(self, vertices):
        self.V = vertices    # No. of vertices
        self.graph = []    # default dictionary

    # to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])
```

```
# A utility function to find set of an element i
# (uses path compression technique)
def find(self, parent, i):
        if parent[i] == i:
                return i
        return self.find(parent, parent[i])

# A function that does union of two sets of x and y
# (uses union by rank)
def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
                parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
                parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
                parent[yroot] = xroot
                rank[xroot] += 1

# The main function to construct MST using Kruskal's
# algorithm
def KruskalMST(self):

        result = []    # This will store the resultant MST

        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # Step 1: Sort all the edges in
        # non-decreasing order of their
        # weight. If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,
                                        key=lambda item: item[2])
```

```
        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge does't
            # cause cycle, include it in result
            # and increment the indexof result
            # for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
        # Else discard the edge

        minimumCost = 0
        print
        "Edges in the constructed MST"
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)


# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)
```

# Function call
g.KruskalMST()

INPUT:

```
# Python program for Kruskal's algorithm to find by Md nayeem MOlla 381
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph

from collections import defaultdict


# Class to represent a graph


class Graph:

    def __init__(self, vertices):
        self.V = vertices  # No. of vertices
        self.graph = []  # default dictionary

    # to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
```

main.py

```python
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's
    # algorithm
    def KruskalMST(self):

        result = []  # This will store the resultant MST

        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # Step 1: Sort all the edges in
        # non-decreasing order of their
        # weight. If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
```

main.py

ects\pythonProject5

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

            # If including this edge does't
            # cause cycle, include it in result
            # and increment the indexof result
            # for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
        # Else discard the edge

        minimumCost = 0
        print
        "Edges in the constructed MST"
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)


# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

# Function call
g.KruskalMST()
```

Output:

```
Run:   main
"C:\Users\NR nayeem\PycharmProjects\pythonProject5\venv\Scripts\python.exe" "C:/Users/NR nayeem/PycharmProjects/pythonProject5/main.py"
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19

Process finished with exit code 0
```