Project 2: Graph Algorithms and Related Data Structures

Singles-source shortest path algorithm, Minimum Spanning Tree (MST), and Graph Fundamentals

- Teja Swaroop Sayya

Problem 1: Single-source Shortest Path Algorithm

Let G be the connected weighted graph. And Path P

- Then Length of path P is the sum of the weights of the edges in path P.
- If path P contains edges e0, e1, e2, e3, … , en-1 then length of P is denoted as w(P):

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

- The distance of vertex **v** from a vertex **s**(source) is the length of the shortest path between **v** and **s**
- Denoted as (s, v). d(s, v) = +♾️ if no path exist.

Edge Relaxation:
- It Repeatedly reduces the upper bound of an actual shortest path, for every vertex, till upper bound equals the shortest path length.
- For every vertex **v** , we maintain an attribute **d[v]** which is the shortest path estimate.It is an upper bound of a shortest path length from source s to v.
- Consider an edge e = (u, v) such that, **u** = vertex most recently added to the cloud, **v** = not in the cloud
- The relaxation edge updates distance as follows:

```
Relax (u, v, w):
If d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
        $\pi$[v] = u
```

Dijkstra's algorithm:

· It computes the shortest distances of all the vertices from a given start vertex s

Assumptions:

· graph is connected

· edges are directed/undirected

· edge weights are nonnegative, i.e. $w(e) \geq 0$

How Dijkstra's Algorithm works:

· Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.

· Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.

· For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.

· We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.

· Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.

· In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

Pseudo code of Dijkstra's Algorithm:

Dijkstra's algorithm is a kind of greedy algorithm.We grow a "cloud or set" of vertices, beginning with start vertex $\boldsymbol{s}$ and eventually covering all the vertices.

For each vertex $\boldsymbol{v}$ a label $\boldsymbol{d}(\boldsymbol{v})$: the distance of $\boldsymbol{v}$ from $\boldsymbol{s}$ in the subgraph consisting of the cloud and its adjacent vertices

At each step

· We add to the cloud the vertex $\boldsymbol{u}$ outside the cloud with the smallest distance label, $\boldsymbol{d}(\boldsymbol{u})$

· We update the labels of the vertices adjacent to u

```
Algorithm INITIALIZE-SINGLE-SOURCE (G, s)
    for each vertex v $\in$G.V
        d[v] = $\infty$
        $\pi$[v] = NIL
    d[s] = 0
```

```
Algorithm DIJKSTRA ( G, w, s)
    INITIALIZE-SINGLE-SOURCE (G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX (u, v, w)
```

Test on 8 input graphs(4 directed and 4 undirected) with sample input and expected output:

Sample input 1:

```
16 25 U
 A B 2
 A C 3
 A D 1
 B C 4
 B E 2
 C D 2
 C F 3
 D G 4
 E F 1
```

```
E H 2
F G 2
F I 3
G J 1
H I 1
H K 2
I J 2
I L 3
J M 1
K L 2
K N 3
L M 1
M O 2
N O 4
N P 3
O P 1
A
```

Expected Output:

Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological order)

1

Select an Input File

```
1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

5. Directed Graph-Input1

6. Directed Graph-Input2

7. Directed Graph-Input3.

8. Directed Graph-Input4.1

1

========== Dijkstra's Algorithm ===========

Number of Vertices: 16

Number of Edges: 25

Execution Time: 5427200 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

AA --> B Path cost: 2

AA --> C Path cost: 3

AA --> D Path cost: 1

AA --> B --> E Path cost: 4
AA --> B --> E --> F Path cost: 5

AA --> D --> G Path cost: 5

AA --> B --> E --> H Path cost: 6

AA --> B --> E --> H --> I Path cost: 7

AA --> D --> G --> J Path cost: 6

AA --> B --> E --> H --> K Path cost: 8

AA --> D --> G --> J --> M --> L Path cost: 8

AA --> D --> G --> J --> M Path cost: 7

AA --> B --> E --> H --> K --> N Path cost: 11

AA --> D --> G --> J --> M --> O Path cost: 9

AA --> D --> G --> J --> M --> O --> P Path cost: 10
```

Sample Input 2:

```
20 30 U
 A B 3
 A C 2
 A D 5
 B C 4
 B E 1
 C D 2
 C F 3
 D G 4
 E F 2
```

```
E H 3
F G 2
F I 4
G J 1
H I 2
H K 3
I J 4
I L 2
J M 1
K L 5
K N 3
L M 2
L O 4
M N 2
M P 3
N Q 1
O P 2
P Q 3
Q R 4
R S 2
S T 3
A
```

Expected output:

Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

```
3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

1

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

5. Directed Graph-Input1

6. Directed Graph-Input2

7. Directed Graph-Input3.

8. Directed Graph-Input4.

2

========== Dijkstra's Algorithm ===========

Number of Vertices: 20

Number of Edges: 30

Execution Time: 3548900 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 3
```

A --> C Path cost: 2

A --> C --> D Path cost: 4

A --> B --> E Path cost: 4

A --> C --> F Path cost: 5

A --> C --> F --> G Path cost: 7

A --> B --> E --> H Path cost: 7

A --> C --> F --> I Path cost: 9

A --> C --> F --> G --> J Path cost: 8

A --> B --> E --> H --> K Path cost: 10

A --> C --> F --> I --> L Path cost: 11

A --> C --> F --> G --> J --> M Path cost: 9

A --> C --> F --> G --> J --> M --> N Path cost: 11

A --> C --> F --> G --> J --> M --> P --> O Path cost: 14

A --> C --> F --> G --> J --> M --> P Path cost: 12

A --> C --> F --> G --> J --> M --> N --> Q Path cost: 12

A --> C --> F --> G --> J --> M --> N --> Q --> R Path cost: 16

A --> C --> F --> G --> J --> M --> N --> Q --> R --> S Path cost: 18

A --> C --> F --> G --> J --> M --> N --> Q --> R --> S --> T Path cost: 21

Sample Input3:

```
16 25 D
 A B 2
 A C 3
 A D 1
 B C 4
 B E 2
 C D 2
 C F 3
 D G 4
 E F 1
 E H 2
 F G 2
 F I 3
 G J 1
 H I 1
 H K 2
 I J 2
 I L 3
 J M 1
 K L 2
 K N 3
 L M 1
 M O 2
 N O 4
 N P 3
 O P 1
 A
```

Expected output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

1

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

5. Directed Graph-Input1

6. Directed Graph-Input2

7. Directed Graph-Input3.

8. Directed Graph-Input4.

5

========== Dijkstra's Algorithm ===========
Number of Vertices: 16

Number of Edges: 25

Execution Time: 3142900 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 2

A --> C Path cost: 3

A --> D Path cost: 1

A --> B --> E Path cost: 4

A --> B --> E --> F Path cost: 5

A --> D --> G Path cost: 5

A --> B --> E --> H Path cost: 6

A --> B --> E --> H --> I Path cost: 7

A --> D --> G --> J Path cost: 6

A --> B --> E --> H --> K Path cost: 8

A --> B --> E --> H --> I --> L Path cost: 10

A --> D --> G --> J --> M Path cost: 7

A --> B --> E --> H --> K --> N Path cost: 11

A --> D --> G --> J --> M --> O Path cost: 9
```

```
A --> D --> G --> J --> M --> O --> P Path cost: 10
```

Sample Input 4:

```
26 25 D
 A B 4
 A C 7
 A D 3
 B E 2
 B F 6
 C G 5
 C H 8
 D I 9
 D J 4
 E K 3
 E L 5
 F M 7
 F N 2
 G O 8
 G P 6
 H Q 4
 H R 5
 I S 2
 I T 9
 J U 6
 J V 7
 K W 4
 K X 8
 L Y 3
```

```
L Z 2
A
```

Expected output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

1

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

5. Directed Graph-Input1

6. Directed Graph-Input2

7. Directed Graph-Input3.

8. Directed Graph-Input4.

6
```

```
========== Dijkstra's Algorithm ===========

Number of Vertices: 26

Number of Edges: 25

Execution Time: 6334900 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 4

A --> C Path cost: 7

A --> D Path cost: 3

A --> B --> E Path cost: 6

A --> B --> F Path cost: 10

A --> C --> G Path cost: 12

A --> C --> H Path cost: 15

A --> D --> I Path cost: 12

A --> D --> J Path cost: 7

A --> B --> E --> K Path cost: 9

A --> B --> E --> L Path cost: 11

A --> B --> F --> M Path cost: 17
```

A --> B --> F --> N Path cost: 12

A --> C --> G --> O Path cost: 20

A --> C --> G --> P Path cost: 18

A --> C --> H --> Q Path cost: 19

A --> C --> H --> R Path cost: 20

A --> D --> I --> S Path cost: 14

A --> D --> I --> T Path cost: 21

A --> D --> J --> U Path cost: 13

A --> D --> J --> V Path cost: 14

A --> B --> E --> K --> W Path cost: 13

A --> B --> E --> K --> X Path cost: 17

A --> B --> E --> L --> Y Path cost: 14

A --> B --> E --> L --> Z Path cost: 13

Data Structures Used:

- ArrayList: Used to store lists of adjacent vertices for each vertex in the graph.
- HashMap: Used to represent the adjacency list, mapping each vertex to its list of adjacent vertices.
- PriorityQueue: Implemented as a binary min-heap to efficiently retrieve vertices with minimum distances during Dijkstra's algorithm.
- Arrays: Utilized to store distances from the source vertex to each vertex in the graph.

Analysis of Runtime:

- The main operations in Dijkstra's algorithm involve updating distances and extracting the vertex with the minimum distance.
- The PriorityQueue operations (offer and poll) take O(log n) time, where n is the number of vertices.
- Each vertex and edge are visited at most once, leading to O(m log n) time complexity, where m is the number of edges and n is the number of vertices.
- Overall, the runtime of Dijkstra's algorithm is O((n + m) log n) in the worst case scenario.

If all vertices are reachable from the source vertex, the runtime reduces to O(m log n).

RunTime of ShortestPathAlgorithm:

S.No	**Graph Type**	**Edges**	**Vertices**	**Average Runtime (Nanoseconds)**
1.	Undirected	16	25	5477200
2.	Undirected	20	30	3548900
3.	Undirected	16	25	3142900
4.	Undirected	16	25	6334900
5.	Directed	16	25	1543000
6.	Directed	26	25	2141900
7.	Directed	25	26	2476200
8.	Directed	15	25	1836800

Problem 2: Minimum Spanning Tree Algorithm (Kruskal)

Spanning Tree: A spanning tree of a given graph is a tree that contains all the vertices of thegraph G connected by some edges.

• A graph can contain more than one spanning tree. For a graph with n number of vertices,there exists n-1 edges in a resultant spanning tree.

Minimum Spanning Tree (MST): A single spanning tree which has the minimum weighted edges than all other spanning trees of a graph G is known as a minimum spanning tree (MST).

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

• A MST of a graph G is a subgraph which connects every other vertex in the graph with atotal minimum weight of edges.

• There are two ways to find MST.

1. Kruskal’s Algorithm

2. Prim’s Algorithm

We will be using Kruskal’s Algorithm now.

Kruskal’s Algorithm:

Steps to create Minimum Spanning Tree (MST) using Kruskal’s Algorithm:

1. It starts with a forest where each vertex is a tree (i.e., a single node tree).

2. It finds a safe edge to add to the growing forest by finding the safe edge (u,v) of all theedges that connect any two trees in the forest which has the least weight.

• Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forestan edge of least possible weight.

• At the end of the algorithm:

We are left with one cloud (i.e., one tree) that encompasses the MST.

Psedo code for kruskal’s algorithm:

```
Algorithm MST-KRUSKAL (G, w)
  A = ∅
  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, taken in nondecreasing order by weight
              if FIND-SET (u) ≠ FIND-SET (v)
                   A = A ∪ {(u, v)}
                   UNION (u, v)
  return A
```

Test on 4 input(Undirected graphs with sample input and expected output:

Sample Input 1:

16 25 U

A B 2

A C 3

A D 1

B C 4

B E 2

C D 2

C F 3

D G 4

E F 1

E H 2

F G 2

F I 3

G J 1

H I 1

H K 2

I J 2

I L 3

J M 1

K L 2

K N 3

L M 1

M O 2

N O 4

N P 3

O P 1

A

Expected Output:

Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological order)

2

Select an Input File

1. Undirected Graph-Input1

```
2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

1

============   Kruskal's Algorithm ============


Vertices Count: 16

Edges Count: 25

Execution Time: 10001300 nanoseconds


Minimum Spanning Tree:


A --> D Cost: 1

E --> F Cost: 1

G --> J Cost: 1

H --> I Cost: 1

J --> M Cost: 1

L --> M Cost: 1

O --> P Cost: 1
```

```
A --> B Cost: 2

B --> E Cost: 2

C --> D Cost: 2

E --> H Cost: 2

F --> G Cost: 2

H --> K Cost: 2

M --> O Cost: 2

K --> N Cost: 3


Total Cost of MST: 24
```

Sample Input 2:

```
20 30 U
 A B 3
 A C 2
 A D 5
 B C 4
 B E 1
 C D 2
 C F 3
 D G 4
 E F 2
 E H 3
```

```
F G 2
F I 4
G J 1
H I 2
H K 3
I J 4
I L 2
J M 1
K L 5
K N 3
L M 2
L O 4
M N 2
M P 3
N Q 1
O P 2
P Q 3
Q R 4
R S 2
S T 3
A
```

Expected Output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)
```

```
3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

2

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

2

============   Kruskal's Algorithm ============


Vertices Count: 20

Edges Count: 30

Execution Time: 1387700 nanoseconds


Minimum Spanning Tree:


B --> E Cost: 1

G --> J Cost: 1

J --> M Cost: 1
```

N --> Q Cost: 1

A --> C Cost: 2

C --> D Cost: 2

E --> F Cost: 2

F --> G Cost: 2

H --> I Cost: 2

I --> L Cost: 2

L --> M Cost: 2

M --> N Cost: 2

O --> P Cost: 2

R --> S Cost: 2

A --> B Cost: 3

H --> K Cost: 3

M --> P Cost: 3

S --> T Cost: 3

Q --> R Cost: 4

Total Cost of MST: 40

Sample Input 3:

```
16 25 U
A B 2
A C 3
A D 1
B C 4
B E 2
C D 2
C F 3
D G 4
E F 1
E H 2
F G 2
F I 3
G J 1
H I 1
```

H K 2

I J 2

I L 3

J M 1

K L 2

K N 3

L M 1

M O 2

N O 4

N P 3

O P 1

A

Expected Output:

Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

```
2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

2

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

3

============   Kruskal's Algorithm ============

Vertices Count: 16

Edges Count: 25

Execution Time: 1548000 nanoseconds


Minimum Spanning Tree:


A --> D Cost: 1

E --> F Cost: 1

G --> J Cost: 1
```

```
H --> I Cost: 1

J --> M Cost: 1

L --> M Cost: 1

O --> P Cost: 1

A --> B Cost: 2

B --> E Cost: 2

C --> D Cost: 2

E --> H Cost: 2

F --> G Cost: 2

H --> K Cost: 2

M --> O Cost: 2

K --> N Cost: 3


Total Cost of MST: 24
```

Sample Input 4:

```
16 25 U
 A B 2
 A C 3
 A D 1
 B C 4
```

```
B E 2
C D 2
C F 3
D G 4
E F 1
E H 2
F G 2
F I 3
G J 1
H I 1
H K 2
I J 2
I L 3
J M 1
K L 2
K N 3
L M 1
M O 2
N O 4
N P 3
O P 1
A
```

Expected Output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)
```

```
3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

2

Select an Input File

1. Undirected Graph-Input1

2. Undirected Graph-Input2

3. Undirected Graph-Input3.

4. Undirected Graph-Input4.

4

============   Kruskal's Algorithm ============


Vertices Count: 16

Edges Count: 25

Execution Time: 1398800 nanoseconds


Minimum Spanning Tree:


A --> D Cost: 1

E --> F Cost: 1

G --> J Cost: 1
```

H --> I Cost: 1

J --> M Cost: 1

L --> M Cost: 1

O --> P Cost: 1

A --> B Cost: 2

B --> E Cost: 2

C --> D Cost: 2

E --> H Cost: 2

F --> G Cost: 2

H --> K Cost: 2

M --> O Cost: 2

K --> N Cost: 3

Total Cost of MST: 24

Data Structures Used:

- Array List: Used for storing edges and vertices.
- Graphs: Represented by adjacency lists to store the graph structure.
- Hash Map: Utilized for mapping vertex indices to their corresponding characters.

- Map: Used for mapping indices to vertices and vice versa.
- Navigable Set: Employed for sorting the edges into non-decreasing order by weight.
- Tree Set: Utilized for sorting edges based on their weights during the algorithm execution.
- Disjoint Set (Union-Find Data Structure): Implemented using custom classes (Link and Path) for tracking disjoint sets of vertices and determining if adding an edge creates a cycle in the spanning tree.

Runtime Analysis:

In Kruskal's Algorithm, edges of graph G are sorted into non-decreasing order by weightw. Time taken for this sorting operation is,O (m log m) where, m represents edges.

• The runtime depends on the way we perform SET operations. Here, the runtime is givenbased on the assumption that disjoint-set-forest implementation is used as it is the

asymptotically fastest implementation known.

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

• The function FIND-SET takes O(m) time to execute for m edges.

• The function UNION takes O(n) time for n vertices.• Therefore, the total running time to execute kruskal's algorithm for finding minimum spanning tree would be O (m log m), where m is number of edges.

• If |E| < V2

log |m| = O (log n)

Then the running time would be O(m log n) which is asymptotically same as Prim's

Algorithm.

RunTime of MinimumSpanningAlgorithm:

S.No	**Graph Type**	**Edges**	**Vertices**	**Average Runtime (Nanoseconds)**
1.	Undirected	16	25	10001300
2.	Undirected	20	30	1387700
3.	Undirected	16	25	1548000
4.	Undirected	16	25	1398800

Problem 3: DFS - Topological Sorting and Cycles

A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

If the graph contains a cycle, then no linear ordering is possible.

We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges gfrom left to right. T

Algorithm for Topological Sorting using DFS:

Here's a step-by-step algorithm for topological sorting using Depth First Search (DFS):

· Create a graph with n vertices and m-directed edges.

Initialize a stack and a visited array of size n.

· For each unvisited vertex in the graph, do the following:

· Call the DFS function with the vertex as the parameter.

· In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.

· Once all the neighbors have been visited, push the vertex onto the stack.

· After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.

· The resulting list is the topologically sorted order of the graph.

Test on four input graphs with sample input and expected output:

Sample Input 1:

```
16 25 D
 A B 2
 A C 3
 A D 1
 B C 4
 B E 2
 C D 2
 C F 3
 D G 4
 E F 1
 E H 2
 F G 2
 F I 3
 G J 1
 H I 1
 H K 2
 I J 2
 I L 3
 J M 1
 K L 2
 K N 3
 L M 1
 M O 2
```

```
N O 4
N P 3
O P 1
A
```

Expected Output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

3

Select an Input File

5. DAG-Input1

6. DAG-Input2

7. Cyclic Graph-Input3.

8. Cyclic Graph-Input4.

5

=============  Topological Sorting  ============


The graph is acyclic. Performing topological sorting...

Topological sorting sequence:
```

```
A B E H K N C F I L D G J M O P
```

Sample Input 2:

```
26 25 D
 A B 4
 A C 7
 A D 3
 B E 2
 B F 6
 C G 5
 C H 8
 D I 9
 D J 4
 E K 3
 E L 5
 F M 7
 F N 2
 G O 8
 G P 6
 H Q 4
 H R 5
 I S 2
 I T 9
 J U 6
 J V 7
 K W 4
 K X 8
 L Y 3
 L Z 2
 A
```

Expected Output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

3

Select an Input File

5. DAG-Input1

6. DAG-Input2

7. Cyclic Graph-Input3.

8. Cyclic Graph-Input4.

6

=============  Topological Sorting  ============


The graph is acyclic. Performing topological sorting...

Topological sorting sequence:

A D J V U I T S C H R Q G P O B F N M E L Z Y K X W
```

Sample Input 3:

```
25 26 D
 A B 1
 B C 1
 C D 1
 D E 1
 E F 1
 F G 1
 G H 1
 H I 1
 I J 1
 J K 1
 K L 1
 L M 1
 M N 1
 N O 1
 O A 1
 A P 1
 P Q 1
 Q R 1
 R S 1
 S T 1
 T U 1
 U V 1
 V W 1
 W X 1
 X Y 1
 Y A 1
 A
```

Expected Output:

```
Select an Algorithm:

1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological order)

3

Select an Input File

5. DAG-Input1

6. DAG-Input2

7. Cyclic Graph-Input3.

8. Cyclic Graph-Input4.

7

=============  Topological Sorting  ============


The graph contains cycles.

Cycles along with their lengths:

Cycle 1: A -> B -> C -> D -> E -> F -> G -> H -> I -> J -> K -> L -> M -> N -> O (Length: 15)

Cycle 2: A -> P -> Q -> R -> S -> T -> U -> V -> W -> X -> Y (Length: 11)
```

Sample Input 4:

```
15 25 D
 A B
 A C
 B D
 B E
 C F
 C G
 D H
 E I
 F J
 G K
 H L
 I M
 J N
 K O
 L A
 M B
 N C
 O D
 E F
 G H
 I J
 K L
 M N
 N O
 E O
```

```
A
```

Expected Output:

```
Select an Algorithm:
```

```
1. Dijkstra's Algorithm (Calculates Shortest Path)

2. Kruskal's Algorithm (Minimum Spanning Tree)

3. Topological Sorting (if cyclic, print cycles; if not, print topological
order)

3

Select an Input File

5. DAG-Input1

6. DAG-Input2

7. Cyclic Graph-Input3.

8. Cyclic Graph-Input4.

8

=============  Topological Sorting  ============


The graph contains cycles.

Cycles along with their lengths:

Cycle 1: A -> B -> D -> H -> L (Length: 5)

Cycle 2: B -> E -> I -> M (Length: 4)

Cycle 3: N -> C -> F -> J (Length: 4)
```

RunTime of TopologicalSortAlgorithm:

S.No	Graph Type	Vertices	Edges	Average Runtime (Nanoseconds)
1.	Directed Acyclic	16	25	622099.9639481306
2.	Directed Acyclic	26	25	745900.0335074961
3.	Directed Cyclic	25	26	1936399.9599590898
4.	Directed Cyclic	15	25	1207099.9946445227

Source Code:

Main.java

```
package Algorithms;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        ShortestPathAlgorithm dijkstra = new ShortestPathAlgorithm();
        MinimumSpanningTree kruskal = new MinimumSpanningTree();
        TopologicalSort topoSort = new TopologicalSort();

        Scanner scanner = new Scanner(System.in);
        int selectedFile;

        System.out.println("Choose an Algorithm:\n" +
                "1. Single-source Shortest Path (Dijkstra's Algorithm)\n" +
                "2. Minimum Spanning Tree (Kruskal's Algorithm)\n" +
                "3. Topological Sorting (Cycle detection and topological order)");

        int choice = Integer.parseInt(scanner.nextLine());
        switch (choice) {
            case 1:
                System.out.println("Select an Input File:\n" +
                        "1. Undirected Graph - Input1\n" +
                        "2. Undirected Graph - Input2\n" +
                        "3. Undirected Graph - Input3\n" +
                        "4. Undirected Graph - Input4\n" +
                        "5. Directed Graph - Input1\n" +
                        "6. Directed Graph - Input2\n" +
                        "7. Directed Graph - Input3\n" +
                        "8. Directed Graph - Input4");
                selectedFile = Integer.parseInt(scanner.nextLine());
            String dijkstraFilePath = getFilePath(selectedFile);
            System.out.println("========== Dijkstra's Algorithm
===========");
            dijkstra.runAlgorithm(dijkstraFilePath);
            break;
         case 2:
            System.out.println("Select an Input File:\n" +
                  "1. Undirected Graph - Input1\n" +
                  "2. Undirected Graph - Input2\n" +
                  "3. Undirected Graph - Input3\n" +
                  "4. Undirected Graph - Input4");
            selectedFile = Integer.parseInt(scanner.nextLine());
            String kruskalFilePath = getFilePath(selectedFile);
            System.out.println("========== Kruskal's Algorithm
==========");
            kruskal.executeMST(kruskalFilePath);
            break;
         case 3:
            System.out.println("Select an Input File:\n" +
                  "5. DAG - Input1\n" +
                  "6. DAG - Input2\n" +
                  "7. Cyclic Graph - Input3\n" +
                  "8. Cyclic Graph - Input4");
            selectedFile = Integer.parseInt(scanner.nextLine());
            String topoFilePath = getFilePath(selectedFile);
            System.out.println("========== Topological Sorting
==========");
            topoSort.performTopologicalSort(topoFilePath);
            break;
         default:
            System.out.println("Invalid input. Please choose a valid option.");
            break;
      }
      scanner.close();
   }
```

```
    static String getFilePath(int selectedFile) {
        String filePath;
        switch (selectedFile) {
            case 1:
                filePath = "./inputfiles/UndirectedGraph_Input1.txt";
                break;
            case 2:
                filePath = "./inputfiles/UndirectedGraph_Input2.txt";
                break;
            case 3:
                filePath = "./inputfiles/UndirectedGraph_Input3.txt";
                break;
            case 4:
                filePath = "./inputfiles/UndirectedGraph_Input4.txt";
                break;
            case 5:
                filePath = "./inputfiles/DirectedGraph_Input1.txt";
                break;
            case 6:
                filePath = "./inputfiles/DirectedGraph_Input2.txt";
                break;
            case 7:
                filePath = "./inputfiles/DirectedGraph_Input3.txt";
                break;
            case 8:
                filePath = "./inputfiles/DirectedGraph_Input4.txt";
                break;
            default:
                System.out.println("Invalid Input. Defaulting to Undirected Graph
- Input1.");
                filePath = "./inputfiles/UndirectedGraph_Input1.txt";
                break;
        }
        return filePath;
    }
}
```

—---

Dijkstra’s Algorithm:

ShortestPathAlgorithm.java

```
package Algorithms;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class ShortestPathAlgorithm {
    private int startNode;
    private int numVertices;
    private int numEdges;
    private Map<Integer, List<int[]>> adjacencyMap;
    private int[] minDistances;
    private int[] previousNode;  // Tracks the previous node for path
reconstruction
    private boolean[] visitedNodes;
    private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public void runAlgorithm(String filePath) {
        try {
            loadGraph(filePath);
            long startTime = System.nanoTime();
            findShortestPaths();
            long endTime = System.nanoTime();
            displayResults(startTime, endTime);
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
            e.printStackTrace();
        }
    }

    private void loadGraph(String filePath) throws FileNotFoundException {
        File file = new File(filePath);
        Scanner scanner = new Scanner(file);
        numVertices = scanner.nextInt();
        numEdges = scanner.nextInt();
        char graphType = scanner.next().charAt(0);

        adjacencyMap = new HashMap<>();
        minDistances = new int[numVertices];
        previousNode = new int[numVertices];
        visitedNodes = new boolean[numVertices];

        for (int i = 0; i < numEdges; i++) {
            String fromNode = scanner.next();
            String toNode = scanner.next();
            int weight = scanner.nextInt();

            int fromIndex = alphabet.indexOf(fromNode.charAt(0));
            int toIndex = alphabet.indexOf(toNode.charAt(0));
            adjacencyMap.computeIfAbsent(fromIndex, k -> new ArrayList<>())
                    .add(new int[]{toIndex, weight});
            if (graphType == 'U') {
                adjacencyMap.computeIfAbsent(toIndex, k -> new ArrayList<>())
                        .add(new int[]{fromIndex, weight});
            }
        }

        String startNodeStr = scanner.next();
        startNode = alphabet.indexOf(startNodeStr.charAt(0));
        Arrays.fill(minDistances, Integer.MAX_VALUE);
        Arrays.fill(previousNode, -1);
        scanner.close();
    }

    private void findShortestPaths() {
        PriorityQueue<int[]> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        minDistances[startNode] = 0;
        priorityQueue.offer(new int[]{startNode, 0});

        while (!priorityQueue.isEmpty()) {
            int[] current = priorityQueue.poll();
            int currentNode = current[0];
            int currentDist = current[1];
            if (visitedNodes[currentNode]) continue;

            visitedNodes[currentNode] = true;

            for (int[] neighbor : adjacencyMap.getOrDefault(currentNode,
Collections.emptyList())) {
                int neighborNode = neighbor[0];
                int edgeWeight = neighbor[1];
                int newDist = currentDist + edgeWeight;

                if (newDist < minDistances[neighborNode]) {
                    minDistances[neighborNode] = newDist;
                    previousNode[neighborNode] = currentNode;
                    priorityQueue.offer(new int[]{neighborNode, newDist});
                }
            }
        }
    }

    private void displayResults(long startTime, long endTime) {
        System.out.println("Number of Vertices: " + numVertices);
        System.out.println("Number of Edges: " + numEdges);
        System.out.println("Execution Time: " + (endTime - startTime) + "
nanoseconds");
        System.out.println("Shortest Path Tree from source vertex " +
alphabet.charAt(startNode) + ":");
        System.out.println("Path from " + alphabet.charAt(startNode) + " to
other vertices:");

        for (int i = 0; i < numVertices; i++) {
            if (i != startNode) {
                if (minDistances[i] == Integer.MAX_VALUE) {
                    System.out.println(alphabet.charAt(startNode) + " --> " +
alphabet.charAt(i) + " Path cost: Unreachable");
                } else {
                    String path = buildPath(i);
                    System.out.println(alphabet.charAt(startNode) + path + " Path
cost: " + minDistances[i]);
                }
            }
        }
    }

    private String buildPath(int destination) {
        StringBuilder path = new StringBuilder();
        for (int at = destination; at != -1; at = previousNode[at]) {
            if (at != destination) {
                path.insert(0, " --> ");
            }
            path.insert(0, alphabet.charAt(at));
        }
        return path.toString();
    }
}
```

Kruskals Algoriithm:

MinimumSpanningTree.java

```
package Algorithms;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;

public class MinimumSpanningTree {
    private int numVertices, numEdges;
    private int mstEdgeCount;
    private Edge[] edgeList, mstEdgeList;
    private String nodeLabels = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    class Subset {
        int parent, rank;
    }

    class Edge implements Comparable<Edge> {
        int src, dest, cost;

        public int compareTo(Edge otherEdge) {
            return this.cost - otherEdge.cost;
        }
    }

    public void executeMST(String filePath) {
        try {
            readGraph(filePath);
            long startTime = System.nanoTime();
            applyKruskalMST();
            long endTime = System.nanoTime();
            printMSTResult(startTime, endTime);
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
            e.printStackTrace();
        }
    }

    private void readGraph(String filePath) throws FileNotFoundException {
        File file = new File(filePath);
        Scanner scanner = new Scanner(file);
        numVertices = scanner.nextInt();
        numEdges = scanner.nextInt();
        System.out.println("Number of Vertices: " + numVertices);
        System.out.println("Number of Edges: " + numEdges);
        scanner.next().charAt(0);

        edgeList = new Edge[numEdges];
        for (int i = 0; i < numEdges; ++i) {
            edgeList[i] = new Edge();
        }

        for (int i = 0; i < numEdges; i++) {
            int source = nodeLabels.indexOf(scanner.next().charAt(0));
            int destination = nodeLabels.indexOf(scanner.next().charAt(0));
            int weight = scanner.nextInt();
            edgeList[i].src = source;
            edgeList[i].dest = destination;
            edgeList[i].cost = weight;
        }
        scanner.close();
    }

    void applyKruskalMST() {
        mstEdgeList = new Edge[numVertices];
        mstEdgeCount = 0;
```

```
        int i;

        for (i = 0; i < numVertices; ++i) {
            mstEdgeList[i] = new Edge();
        }

        Arrays.sort(edgeList);
        Subset[] subsets = new Subset[numVertices];

        for (i = 0; i < numVertices; ++i) {
            subsets[i] = new Subset();
        }

        for (int v = 0; v < numVertices; ++v) {
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }

        i = 0;

        while (mstEdgeCount < numVertices - 1) {
            Edge nextEdge = edgeList[i++];
            int rootSrc = find(subsets, nextEdge.src);
            int rootDest = find(subsets, nextEdge.dest);

            if (rootSrc != rootDest) {
                mstEdgeList[mstEdgeCount++] = nextEdge;
                union(subsets, rootSrc, rootDest);
            }
        }
    }

    void union(Subset subsets[], int root1, int root2) {
        int root1Parent = find(subsets, root1);
        int root2Parent = find(subsets, root2);
        if (subsets[root1Parent].rank > subsets[root2Parent].rank) {
            subsets[root2Parent].parent = root1Parent;
        } else if (subsets[root1Parent].rank < subsets[root2Parent].rank) {
            subsets[root1Parent].parent = root2Parent;
        } else {
            subsets[root2Parent].parent = root1Parent;
            subsets[root1Parent].rank++;
        }
    }

    int find(Subset subsets[], int i) {
        if (subsets[i].parent != i) {
            subsets[i].parent = find(subsets, subsets[i].parent);
        }
        return subsets[i].parent;
    }

    public void printMSTResult(long startTime, long endTime) {
        System.out.println("Execution Time: " + (endTime - startTime) + "
nanoseconds");
        int totalMSTCost = 0;
        System.out.println("\nMinimum Spanning Tree:\n");

        for (int i = 0; i < mstEdgeCount; ++i) {
            System.out.println(nodeLabels.charAt(mstEdgeList[i].src) + " --> " +
                    nodeLabels.charAt(mstEdgeList[i].dest) + " Cost: " +
mstEdgeList[i].cost);
            totalMSTCost += mstEdgeList[i].cost;
        }

        System.out.println("\nTotal Cost of MST: " + totalMSTCost);
    }
}
```

Topological sorting:

```
package Algorithms;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class TopologicalSort {
    private Map<Character, List<Character>> adjList;
    private Set<Character> visitedNodes;
    private Set<Character> activePath;
    private List<List<Character>> detectedCycles;

    public void performTopologicalSort(String filePath) {
        try {
            adjList = new HashMap<>();
            visitedNodes = new HashSet<>();
            activePath = new HashSet<>();
            detectedCycles = new ArrayList<>();

            // Read the graph from the file
            Scanner scanner = new Scanner(new File(filePath));

            int numVertices = scanner.nextInt();
            int numEdges = scanner.nextInt();
            System.out.println("Vertices Count: " + numVertices);
            System.out.println("Edges Count: " + numEdges);
            char graphType = scanner.next().charAt(0);

            scanner.nextLine(); // Skip to the next line

            // Construct the graph
            for (int i = 0; i < numEdges; i++) {
```

```
            String line = scanner.nextLine();
            char startNode = line.charAt(0);
            char endNode = line.charAt(2);

            adjList.computeIfAbsent(startNode, k -> new
ArrayList<>()).add(endNode);
            if (graphType == 'U') {
                adjList.computeIfAbsent(endNode, k -> new
ArrayList<>()).add(startNode);
            }
        }
        scanner.close();

        // Check for cycles using DFS
        for (char node : adjList.keySet()) {
            if (!visitedNodes.contains(node)) {
                detectCycles(node, new HashSet<>(), new ArrayList<>());
            }
        }

        if (detectedCycles.isEmpty()) {
            System.out.println("The graph is acyclic. Proceeding with
topological sort...");
            long startTime = System.nanoTime();
            List<Character> topoOrder = getTopologicalOrder();
            long endTime = System.nanoTime();
            System.out.println("Execution Time: " + (endTime - startTime) + "
nanoseconds");
            System.out.println("Topological sorting order:");
            for (char node : topoOrder) {
                System.out.print(node + " ");
            }
            System.out.println();
        } else {
            System.out.println("The graph contains cycles.");
```

```
                System.out.println("Detected cycles with their lengths:");
                int cycleCount = 1;
                for (List<Character> cycle : detectedCycles) {
                    System.out.print("Cycle " + cycleCount + ": ");
                    for (int i = 0; i < cycle.size(); i++) {
                        System.out.print(cycle.get(i));
                        if (i < cycle.size() - 1) {
                            System.out.print(" -> ");
                        }
                    }
                    System.out.println(" (Length: " + cycle.size() + ")");
                    cycleCount++;
                }
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + filePath);
            e.printStackTrace();
        }
    }

    private void detectCycles(char node, Set<Character> pathNodes,
List<Character> pathTrail) {
        if (activePath.contains(node)) {
            List<Character> cycle = new
ArrayList<>(pathTrail.subList(pathTrail.indexOf(node), pathTrail.size()));
            detectedCycles.add(cycle);
            return;
        }
        if (!visitedNodes.contains(node)) {
            visitedNodes.add(node);
            activePath.add(node);
            pathTrail.add(node);

            List<Character> neighbors = adjList.getOrDefault(node, new
ArrayList<>());
```

```
            for (char neighbor : neighbors) {
                detectCycles(neighbor, pathNodes, pathTrail);
            }

            activePath.remove(node);
            pathTrail.remove(pathTrail.size() - 1);
        }
    }

    private List<Character> getTopologicalOrder() {
        List<Character> topoOrder = new ArrayList<>();
        Set<Character> visited = new HashSet<>();
        Set<Character> inProcess = new HashSet<>();

        for (char node : adjList.keySet()) {
            if (!visited.contains(node)) {
                dfsTopoSort(node, visited, inProcess, topoOrder);
            }
        }

        Collections.reverse(topoOrder);
        return topoOrder;
    }

    private void dfsTopoSort(char node, Set<Character> visited,
Set<Character> inProcess, List<Character> topoOrder) {
        visited.add(node);
        inProcess.add(node);

        List<Character> neighbors = adjList.getOrDefault(node, new
ArrayList<>());
        for (char neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                dfsTopoSort(neighbor, visited, inProcess, topoOrder);
            }
        }

        inProcess.remove(node);
        topoOrder.add(node);
    }
}
```