**PROJECT 2**

**GRAPH ALGORITHMS AND RELATED DATA STRUCTURES**

Single-source shortest path algorithm, Minimum Spanning Tree (MST), and Topological Sorting Algorithm (using DFS)

**ALGORITHMS AND DATA STRUCTURES**

**ITCS 6114 – FALL 2024**



**Submitted by**

**Navya Vejalla**

**801393891**

[nvejalla@charlotte.edu](mailto:nvejalla@charlotte.edu)

**Problem 1: Single-source Shortest Path Algorithm**

* 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 𝑒0, 𝑒1, e2…, 𝑒k-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.

**Edge Relaxation:**

* Edge relaxation is a technique that is used by the shortest path algorithms. It repeatedly
* reduces the upper bound of an actual shortest path, for every vertex, till the upper bound
* equals the shortest path length.
* In edge relaxation, For each 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 is the most recently added vertex 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

**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) ≥ 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 𝒔 and eventually covering all the vertices.

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

At each step

* We add to the cloud the vertex 𝒖 outside the cloud with the smallest distance label, 𝒅(𝒖)
* We update the labels of the vertices adjacent to u

**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] **do**

RELAX (u, v, w)

**Algorithm** INITIALIZE-SINGLE-SOURCE (G, s)

**for** each vertex v ∈ G.V **do**

d[v] = ∞

*π*[v] = NIL

d[s] = 0

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

**Sample input 1:**

16 28 U

A B 3

A C 4

A D 2

B C 5

B E 3

C D 3

C F 4

D G 5

E F 2

E H 3

F G 3

F I 4

G J 2

H I 2

H K 3

I J 3

I L 4

J M 2

K L 3

K N 4

L M 2

M O 3

N O 5

N P 4

O P 2

A P 4

B F 6

C G 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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

5. Directed Graph-1

6. Directed Graph-2

7. Directed Graph-3.

8. Directed Graph-4.

1

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

Number of Vertices: 16

Number of Edges: 28

Execution Time: 7860900 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 3

A --> C Path cost: 4

A --> D Path cost: 2

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

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

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

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

A --> B --> E --> H --> I Path cost: 11

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

A --> B --> E --> H --> K Path cost: 12

A --> P --> O --> M --> L Path cost: 11

A --> P --> O --> M Path cost: 9

A --> P --> N Path cost: 8

A --> P --> O Path cost: 6

A --> P Path cost: 4

**Sample Input 2:**

20 30 U

A B 5

A C 4

A D 7

B C 6

B E 3

C D 4

C F 5

D G 6

E F 4

E H 5

F G 4

F I 6

G J 3

H I 4

H K 5

I J 6

I L 4

J M 3

K L 7

K N 5

L M 4

L O 6

M N 4

M P 5

N Q 3

O P 4

P Q 5

Q R 6

R S 4

S T 5

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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

5. Directed Graph-1

6. Directed Graph-2

7. Directed Graph-3.

8. Directed Graph-4.

2

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

Number of Vertices: 20

Number of Edges: 30

Execution Time: 5869100 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 5

A --> C Path cost: 4

A --> D Path cost: 7

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

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

A --> D --> G Path cost: 13

A --> B --> E --> H Path cost: 13

A --> C --> F --> I Path cost: 15

A --> D --> G --> J Path cost: 16

A --> B --> E --> H --> K Path cost: 18

A --> C --> F --> I --> L Path cost: 19

A --> D --> G --> J --> M Path cost: 19

A --> B --> E --> H --> K --> N Path cost: 23

A --> C --> F --> I --> L --> O Path cost: 25

A --> D --> G --> J --> M --> P Path cost: 24

A --> B --> E --> H --> K --> N --> Q Path cost: 26

A --> B --> E --> H --> K --> N --> Q --> R Path cost: 32

A --> B --> E --> H --> K --> N --> Q --> R --> S Path cost: 36

A --> B --> E --> H --> K --> N --> Q --> R --> S --> T Path cost: 41

**Sample Input 3:**

26 30 D

A B 6

A C 9

A D 5

B E 4

B F 8

C G 7

C H 10

D I 11

D J 6

E K 5

E L 7

F M 9

F N 4

G O 10

G P 8

H Q 6

H R 7

I S 4

I T 11

J U 8

J V 9

K W 6

K X 10

L Y 5

L Z 4

M O 5

N P 6

O Q 7

P R 8

Q S 5

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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

5. Directed Graph-1

6. Directed Graph-2

7. Directed Graph-3.

8. Directed Graph-4.

6

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

Number of Vertices: 26

Number of Edges: 30

Execution Time: 5646300 nanoseconds

Shortest Path Tree from source vertex A:

Path from A to other vertices:

A --> B Path cost: 6

A --> C Path cost: 9

A --> D Path cost: 5

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

A --> B --> F Path cost: 14

A --> C --> G Path cost: 16

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

A --> D --> I Path cost: 16

A --> D --> J Path cost: 11

A --> B --> E --> K Path cost: 15

A --> B --> E --> L Path cost: 17

A --> B --> F --> M Path cost: 23

A --> B --> F --> N Path cost: 18

A --> C --> G --> O Path cost: 26

A --> C --> G --> P Path cost: 24

A --> C --> H --> Q Path cost: 25

A --> C --> H --> R Path cost: 26

A --> D --> I --> S Path cost: 20

A --> D --> I --> T Path cost: 27

A --> D --> J --> U Path cost: 19

A --> D --> J --> V Path cost: 20

A --> B --> E --> K --> W Path cost: 21

A --> B --> E --> K --> X Path cost: 25

A --> B --> E --> L --> Y Path cost: 22

A --> B --> E --> L --> Z Path cost: 21

**Sample Input 4:**

25 29 D

A B 4

B C 4

C D 4

D E 4

D G 4

E F 4

F G 4

G H 4

H I 4

I J 4

J K 4

K L 4

L M 4

M N 4

N O 4

O A 4

O Q 4

A P 4

P Q 4

Q R 4

R S 4

S T 4

T U 4

U V 4

V W 4

W X 4

W A 4

X Y 4

Y A 4

W

**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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

5. Directed Graph-1

6. Directed Graph-2

7. Directed Graph-3.

8. Directed Graph-4.

7

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

Number of Vertices: 25

Number of Edges: 29

Execution Time: 6180800 nanoseconds

Shortest Path Tree from source vertex W:

Path from W to other vertices:

W --> A Path cost: 4

W --> A --> B Path cost: 8

W --> A --> B --> C Path cost: 12

W --> A --> B --> C --> D Path cost: 16

W --> A --> B --> C --> D --> E Path cost: 20

W --> A --> B --> C --> D --> E --> F Path cost: 24

W --> A --> B --> C --> D --> G Path cost: 20

W --> A --> B --> C --> D --> G --> H Path cost: 24

W --> A --> B --> C --> D --> G --> H --> I Path cost: 28

W --> A --> B --> C --> D --> G --> H --> I --> J Path cost: 32

W --> A --> B --> C --> D --> G --> H --> I --> J --> K Path cost: 36

W --> A --> B --> C --> D --> G --> H --> I --> J --> K --> L Path cost: 40

W --> A --> B --> C --> D --> G --> H --> I --> J --> K --> L --> M Path cost: 44

W --> A --> B --> C --> D --> G --> H --> I --> J --> K --> L --> M --> N Path cost: 48

W --> A --> B --> C --> D --> G --> H --> I --> J --> K --> L --> M --> N --> O Path cost: 52

W --> A --> P Path cost: 8

W --> A --> P --> Q Path cost: 12

W --> A --> P --> Q --> R Path cost: 16

W --> A --> P --> Q --> R --> S Path cost: 20

W --> A --> P --> Q --> R --> S --> T Path cost: 24

W --> A --> P --> Q --> R --> S --> T --> U Path cost: 28

W --> A --> P --> Q --> R --> S --> T --> U --> V Path cost: 32

W --> X Path cost: 4

W --> X --> Y Path cost: 8

**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 DijkstraShortestPath:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Graph Type** | **Edges** | **Vertices** | **Average Runtime (Nanoseconds)** |
| 1. | Undirected | 16 | 28 | 6663400 |
| 2. | Undirected | 20 | 30 | 7859400 |
| 3. | Undirected | 18 | 28 | 5978300 |
| 4. | Undirected | 24 | 30 | 5956300 |
| 5. | Directed | 16 | 28 | 5901900 |
| 6. | Directed | 26 | 30 | 5646300 |
| 7. | Directed | 25 | 29 | 5681000 |
| 8. | Directed | 18 | 28 | 5050900 |

**Problem 2: Minimum Spanning Tree Algorithm**

**Spanning Tree**: A spanning tree of a given graph is a tree that contains all the vertices of the graph 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 resultant spanning tree.

Minimum Spanning Tree (MST): A single spanning tree which has the minimum weighted edges than all other spanning trees of 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 a total minimum weight of edges.

• There are two ways to find MST.

1. Kruskal’s Algorithm

2. Prim’s Algorithm

Note: Kruskal’s Algorithm is used in this project.

**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 the edges 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 forest an 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.

**Pseudocode for Kruskal’s algorithm:**

**Algorithm** MST-KRUSKAL (G, w)

A = ∅

**for** each vertex v ∈ G.V **do**

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 **do**

**if** FIND-SET (u) ≠ FIND-SET (v) **then**

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 28 U

A B 3

A C 4

A D 2

B C 5

B E 3

C D 3

C F 4

D G 5

E F 2

E H 3

F G 3

F I 4

G J 2

H I 2

H K 3

I J 3

I L 4

J M 2

K L 3

K N 4

L M 2

M O 3

N O 5

N P 4

O P 2

A P 4

B F 6

C G 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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

1

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

Vertices Count: 16

Edges Count: 28

Execution Time: 1527500 nanoseconds

Minimum Spanning Tree:

A --> D Cost: 2

E --> F Cost: 2

G --> J Cost: 2

H --> I Cost: 2

J --> M Cost: 2

L --> M Cost: 2

O --> P Cost: 2

A --> B Cost: 3

B --> E Cost: 3

C --> D Cost: 3

E --> H Cost: 3

F --> G Cost: 3

H --> K Cost: 3

M --> O Cost: 3

K --> N Cost: 4

Total Cost of MST: 39

**Sample Input 2:**

20 30 U

A B 5

A C 4

A D 7

B C 6

B E 3

C D 4

C F 5

D G 6

E F 4

E H 5

F G 4

F I 6

G J 3

H I 4

H K 5

I J 6

I L 4

J M 3

K L 7

K N 5

L M 4

L O 6

M N 4

M P 5

N Q 3

O P 4

P Q 5

Q R 6

R S 4

S T 5

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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

2

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

Vertices Count: 20

Edges Count: 30

Execution Time: 1431000 nanoseconds

Minimum Spanning Tree:

B --> E Cost: 3

G --> J Cost: 3

J --> M Cost: 3

N --> Q Cost: 3

A --> C Cost: 4

C --> D Cost: 4

E --> F Cost: 4

F --> G Cost: 4

H --> I Cost: 4

I --> L Cost: 4

L --> M Cost: 4

M --> N Cost: 4

O --> P Cost: 4

R --> S Cost: 4

A --> B Cost: 5

H --> K Cost: 5

M --> P Cost: 5

S --> T Cost: 5

Q --> R Cost: 6

Total Cost of MST: 78

**Sample Input 3:**

18 28 U

A B 2

A C 3

B C 4

B E 2

B E 2

C D 6

C I 2

D G 4

D J 3

E F 1

E H 2

E H 2

F G 2

F I 3

G J 1

H I 1

H K 2

I L 3

J M 5

K L 2

L M 1

M O 2

N P 3

O P 1

Q E 4

R G 5

K P 4

L N 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)

2

Select an Input File

1. Undirected Graph-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

3

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

Vertices Count: 18

Edges Count: 28

Execution Time: 2774300 nanoseconds

Minimum Spanning Tree:

E --> F Cost: 1

G --> J Cost: 1

H --> I Cost: 1

L --> M Cost: 1

O --> P Cost: 1

A --> B Cost: 2

B --> E Cost: 2

C --> I Cost: 2

E --> H Cost: 2

F --> G Cost: 2

H --> K Cost: 2

K --> L Cost: 2

M --> O Cost: 2

L --> N Cost: 2

D --> J Cost: 3

Q --> E Cost: 4

R --> G Cost: 5

Total Cost of MST: 35

**Sample Input 4:**

24 30 U

A B 3

A U 4

B C 5

B E 9

C D 3

C F 4

D G 5

E F 2

E H 3

F G 3

F I 4

G J 2

H I 2

H K 8

I J 3

I L 4

J M 2

K L 3

L M 2

M O 3

N O 5

N P 4

O P 2

Q E 6

R G 7

S I 5

T L 6

V W 3

W X 3

U V 4

R

**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-1

2. Undirected Graph-2

3. Undirected Graph-3.

4. Undirected Graph-4.

4

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

Vertices Count: 24

Edges Count: 30

Execution Time: 1280600 nanoseconds

Minimum Spanning Tree:

E --> F Cost: 2

G --> J Cost: 2

H --> I Cost: 2

J --> M Cost: 2

L --> M Cost: 2

O --> P Cost: 2

A --> B Cost: 3

C --> D Cost: 3

E --> H Cost: 3

F --> G Cost: 3

K --> L Cost: 3

M --> O Cost: 3

V --> W Cost: 3

W --> X Cost: 3

A --> U Cost: 4

C --> F Cost: 4

N --> P Cost: 4

U --> V Cost: 4

B --> C Cost: 5

S --> I Cost: 5

Q --> E Cost: 6

T --> L Cost: 6

R --> G Cost: 7

Total Cost of MST: 81

**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 weight w. 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 given based 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 | 28 | 1527500 |
| 2. | Undirected | 20 | 30 | 1431000 |
| 3. | Undirected | 18 | 28 | 2418000 |
| 4. | Undirected | 24 | 30 | 1280600 |

**Problem 3: Topological Sorting Algorithm**

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 g from 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 28 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 F 5

B H 4

C K 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)

3

Select an Input File

5. DAG-1

6. DAG-2

7. Cyclic Graph-3.

8. Cyclic Graph-4.

5

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

Vertices Count: 16

Edges Count: 28

The graph is acyclic. Performing topological sorting...

Execution Time: 329700 nanoseconds

Topological sorting sequence:

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

**Sample Input 2:**

26 30 D

A B 6

A C 9

A D 5

B E 4

B F 8

C G 7

C H 10

D I 11

D J 6

E K 5

E L 7

F M 9

F N 4

G O 10

G P 8

H Q 6

H R 7

I S 4

I T 11

J U 8

J V 9

K W 6

K X 10

L Y 5

L Z 4

M O 5

N P 6

O Q 7

P R 8

Q S 5

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-1

6. DAG-2

7. Cyclic Graph-3.

8. Cyclic Graph-4.

6

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

Vertices Count: 26

Edges Count: 30

The graph is acyclic. Performing topological sorting...

Execution Time: 302400 nanoseconds

Topological sorting sequence:

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

**Sample Input 3:**

25 29 D

A B 4

B C 4

C D 4

D E 4

D G 4

E F 4

F G 4

G H 4

H I 4

I J 4

J K 4

K L 4

L M 4

M N 4

N O 4

O A 4

O Q 4

A P 4

P Q 4

Q R 4

R S 4

S T 4

T U 4

U V 4

V W 4

W X 4

W A 4

X Y 4

Y A 4

W

**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-1

6. DAG-2

7. Cyclic Graph-3.

8. Cyclic Graph-4.

7

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

Vertices Count: 25

Edges Count: 29

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 -> B -> C -> D -> E -> F -> G -> H -> I -> J -> K -> L -> M -> N -> O -> Q -> R -> S -> T -> U -> V -> W -> X -> Y (Length: 24)

Cycle 3: A -> B -> C -> D -> E -> F -> G -> H -> I -> J -> K -> L -> M -> N -> O -> Q -> R -> S -> T -> U -> V -> W (Length: 22)

Execution Time: 4354800 nanoseconds

**Sample Input 4:**

18 28 D

A B 3

A C 2

B D 4

B E 1

C F 5

C G 3

D H 2

E I 6

F J 4

G K 1

H L 3

I M 2

J N 5

K O 4

L A 1

M B 6

N C 3

O D 2

E F 5

G H 1

I J 4

P E 2

Q K 3

R N 4

A P 2

B Q 5

C R 6

M O 3

N

**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-1

6. DAG-2

7. Cyclic Graph-3.

8. Cyclic Graph-4.

8

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

Vertices Count: 18

Edges Count: 28

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: J -> N -> C -> F (Length: 4)

Cycle 4: N -> C -> R (Length: 3)

Execution Time: 2112300 nanoseconds

**Runtime of TopologicalSortAlgorithm:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Graph Type** | **Vertices** | **Edges** | **Average Runtime (Nanoseconds)** |
| 1. | Directed Acyclic | 16 | 28 | 329700 |
| 2. | Directed Acyclic | 26 | 30 | 302400 |
| 3. | Directed Cyclic | 25 | 29 | 4354800 |
| 4. | Directed Cyclic | 18 | 28 | 2112300 |

**Source Code:**

**Main.java class**

package Algorithms;

import java.util.\*;

public class Main {

public static void main(String[] args) {

DijkstraShortestPath algorithm = new DijkstraShortestPath();

MinimumSpanningTree algorithms = new MinimumSpanningTree();

TopologicalSort topologicalSort = new TopologicalSort();

Scanner sc = new Scanner(System.in);

int ipFile;

System.out.println("Select an Algorithm:\n" +

"1. Dijkstra's Algorithm (Calculates Shortest Path)\n" +

"2. Kruskal's Algorithm (Minimum Spanning Tree)\n" +

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

int ch = Integer.parseInt(sc.nextLine());

switch (ch) {

case 1:

// Dijkstra's Algorithm

System.out.println("Select an Input File\n" +

"1. Undirected Graph-1\n" +

"2. Undirected Graph-2\n" +

"3. Undirected Graph-3.\n" +

"4. Undirected Graph-4.\n"+

"5. Directed Graph-1\n" +

"6. Directed Graph-2\n" +

"7. Directed Graph-3.\n" +

"8. Directed Graph-4.");

ipFile = Integer.parseInt(sc.nextLine());

String filePath = getFilePath(ipFile);

System.out.println("========== Dijkstra's Algorithm ===========");

algorithm.executeDijkstra(filePath);

break;

case 2:

// Kruskal's Algorithm

System.out.println("Select an Input File\n" +

"1. Undirected Graph-1\n" +

"2. Undirected Graph-2\n" +

"3. Undirected Graph-3.\n" +

"4. Undirected Graph-4.");

ipFile = Integer.parseInt(sc.nextLine());

String filePaths = getFilePath(ipFile);

System.out.println("========== Kruskal's Algorithm ==========");

algorithms.executeAlgorithm(filePaths);

break;

case 3:

// Topological Sorting

System.out.println("Select an Input File\n" +

"5. DAG-1\n" +

"6. DAG-2\n" +

"7. Cyclic Graph-3.\n" +

"8. Cyclic Graph-4.");

ipFile = Integer.parseInt(sc.nextLine());

String filePathTopo = getFilePath(ipFile);

System.out.println("========== Topological Sorting ==========");

topologicalSort.executeTopologicalSorting(filePathTopo);

break;

default:

System.out.println("Invalid Input!");

break;

}

sc.close();

}

static String getFilePath(int ipfile) {

String fPath;

switch (ipfile) {

case 1:

fPath = "./ipfiles/Undirected\_Graph\_1.txt";

break;

case 2:

fPath = "./ipfiles/Undirected\_Graph\_2.txt";

break;

case 3:

fPath = "./ipfiles/Undirected\_Graph\_3.txt";

break;

case 4:

fPath = "./ipfiles/Undirected\_Graph\_4.txt";

break;

case 5:

fPath = "./ipfiles/Directed\_Graph\_1.txt";

break;

case 6:

fPath = "./ipfiles/Directed\_Graph\_2.txt";

break;

case 7:

fPath = "./ipfiles/Directed\_Graph\_3.txt";

break;

case 8:

fPath = "./ipfiles/Directed\_Graph\_4.txt";

break;

default:

System.out.println("Invalid Input!\n");

fPath = "ipFiles/Undirected\_Graph\_1.txt";

break;

}

return fPath;

}

}

**Dijkstra’s Algorithm:**

DijkstraShortestPath.java class

package Algorithms;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.\*;

/\*\*

\* Implementation of Dijkstra's Shortest Path algorithm.

\*/

public class DijkstraShortestPath {

private int startingVertex;

private int numVertices;

private int numEdges;

private Map<Integer, List<Edge>> adjacencyList;

private int[] shortestDistances;

private boolean[] visitedNodes;

private String alphabetLabels = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

/\*\*

\* Executes Dijkstra's Shortest Path algorithm.

\*

\* filePath Path to the input file containing the graph data.

\*/

public void executeDijkstra(String filePath) {

try {

readGraphFile(filePath);

long startTime = System.nanoTime();

calculateShortestPaths();

long endTime = System.nanoTime();

printResults(startTime, endTime);

} catch (FileNotFoundException e) {

System.out.println("File not found.");

e.printStackTrace();

}

}

/\*\*

\* Reads the graph from the input file.

\*

\* filePath Path to the input file.

\* throws FileNotFoundException

\*/

private void readGraphFile(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);

adjacencyList = new HashMap<>();

shortestDistances = new int[numVertices];

visitedNodes = new boolean[numVertices];

for (int i = 0; i < numEdges; i++) {

String sourceNode = scanner.next();

String destinationNode = scanner.next();

int edgeWeight = scanner.nextInt();

int sourceIndex = alphabetLabels.indexOf(sourceNode.charAt(0));

int destinationIndex = alphabetLabels.indexOf(destinationNode.charAt(0));

adjacencyList.computeIfAbsent(sourceIndex, k -> new ArrayList<>())

.add(new Edge(destinationIndex, edgeWeight));

if (graphType == 'U') {

adjacencyList.computeIfAbsent(destinationIndex, k -> new ArrayList<>()).add(new Edge(sourceIndex, edgeWeight));

}

}

startingVertex = alphabetLabels.indexOf(scanner.next().charAt(0));

scanner.close();

}

/\*\*

\* Calculates shortest paths using Dijkstra's algorithm.

\*/

private void calculateShortestPaths() {

PriorityQueue<Node> minHeap = new PriorityQueue<>(Comparator.comparingInt(Node::getDistance));

Arrays.fill(shortestDistances, Integer.MAX\_VALUE);

shortestDistances[startingVertex] = 0;

minHeap.offer(new Node(startingVertex, 0));

// Loop until priority queue is empty

while (!minHeap.isEmpty()) {

Node currentNode = minHeap.poll();

int vertex = currentNode.getVertex();

int distance = currentNode.getDistance();

if (visitedNodes[vertex]) continue;

visitedNodes[vertex] = true;

// Iterate over neighbors of current vertex

for (Edge neighbor : adjacencyList.getOrDefault(vertex, Collections.emptyList())) {

int nextVertex = neighbor.getDestination();

int weight = neighbor.getWeight();

int newDistance = distance + weight; // Calculate new distance to neighbor

// Update shortest distance if new distance is shorter

if (newDistance < shortestDistances[nextVertex]) {

shortestDistances[nextVertex] = newDistance;

minHeap.offer(new Node(nextVertex, newDistance));

}

}

}

}

/\*\*

\* Prints results.

\*

\* startTime Start time of algorithm execution.

\* endTime End time of algorithm execution.

\*/

private void printResults(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 " + alphabetLabels.charAt(startingVertex) + ":");

System.out.println("Path from " + alphabetLabels.charAt(startingVertex) + " to other vertices:");

for (int i = 0; i < numVertices; i++) {

if (i != startingVertex) {

String path = reconstructPath(i);

System.out.println(alphabetLabels.charAt(startingVertex) + path + " Path cost: " + shortestDistances[i]);

}

}

}

/\*\*

\* Reconstructs the shortest path from starting vertex to destination vertex.

\*

\* destination Destination vertex.

\* return Shortest path as a string.

\*/

private String reconstructPath(int destination) {

StringBuilder path = new StringBuilder();

int currentVertex = destination;

while (currentVertex != startingVertex) {

path.insert(0, " --> " + alphabetLabels.charAt(currentVertex));

currentVertex = findPreviousVertex(currentVertex);

}

return path.toString();

}

/\*\*

\* Finds the previous vertex in the shortest path.

\*

\* vertex Current vertex.

\* return Previous vertex in the shortest path.

\*/

private int findPreviousVertex(int vertex) {

for (int i = 0; i < numVertices; i++) {

int v = i;

if (adjacencyList.getOrDefault(v, Collections.emptyList()).stream()

.anyMatch(edge -> edge.getDestination() == vertex && shortestDistances[v] + edge.getWeight() == shortestDistances[vertex])) {

return i;

}

}

return -1;

}

/\*\*

\* Node class representing a vertex and its distance.

\*/

private static class Node {

private int vertex;

private int distance;

public Node(int vertex, int distance) {

this.vertex = vertex;

this.distance = distance;

}

public int getVertex() {

return vertex;

}

public int getDistance() {

return distance;

}

}

/\*\*

\* Edge class representing a weighted edge between two vertices.

\*/

private static class Edge {

private int destination;

private int weight;

public Edge(int destination, int weight) {

this.destination = destination;

this.weight = weight;

}

public int getDestination() {

return destination;

}

public int getWeight() {

return weight;

}

}

}

**Kruskal’s Algorithm:**

MinimumSpanningTree.java class

package Algorithms;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Arrays;

import java.util.Scanner;

/\*\*

\* This class implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a graph.

\*/

public class MinimumSpanningTree {

private Scanner fileScanner;

private int numVertices, numEdges;

private int edgeCount;

private Edge[] edgeList, mstEdges;

private String vertexLabels = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

/\*\*

\* Represents a link between two vertices.

\*/

class Link {

int vertexA, vertexB;

}

/\*\*

\* Represents a weighted edge between two vertices.

\*/

class Edge implements Comparable<Edge> {

int startVertex, endVertex, weight;

@Override

public int compareTo(Edge otherEdge) {

return this.weight - otherEdge.weight;

}

}

/\*\*

\* Executes Kruskal's algorithm on the graph loaded from the specified file.

\*

\* filePath Path to the file containing the graph data.

\*/

public void executeAlgorithm(String filePath) {

try {

readGraphFromFile(filePath);

long startTime = System.*nanoTime*();

runKruskalAlgorithm();

long endTime = System.*nanoTime*();

displayResults(startTime, endTime);

} catch (FileNotFoundException e) {

System.*out*.println("File not found.");

e.printStackTrace();

}

}

/\*\*

\* Reads the graph data from the specified file.

\*

\* filePath Path to the file containing the graph data.

\* throws FileNotFoundException

\*/

private void readGraphFromFile(String filePath) throws FileNotFoundException {

File file = new File(filePath);

fileScanner = new Scanner(file);

numVertices = fileScanner.nextInt();

numEdges = fileScanner.nextInt();

System.*out*.println("Number of Vertices: " + numVertices);

System.*out*.println("Number of Edges: " + numEdges);

fileScanner.next().charAt(0); // Consume newline character

edgeList = new Edge[numEdges];

for (int i = 0; i < numEdges; ++i) {

edgeList[i] = new Edge();

}

for (int i = 0; i < numEdges; i++) {

int sourceVertex = vertexLabels.indexOf(fileScanner.next().charAt(0));

int destinationVertex = vertexLabels.indexOf(fileScanner.next().charAt(0));

int weight = fileScanner.nextInt();

edgeList[i].startVertex = sourceVertex;

edgeList[i].endVertex = destinationVertex;

edgeList[i].weight = weight;

}

fileScanner.close();

}

/\*\*

\* Runs Kruskal's algorithm to find the Minimum Spanning Tree.

\*/

void runKruskalAlgorithm() {

mstEdges = new Edge[numVertices];

edgeCount = 0;

int index;

for (index = 0; index < numVertices; ++index) {

mstEdges[index] = new Edge();

}

Arrays.*sort*(edgeList);

Link[] vertexLinks = new Link[numVertices];

for (index = 0; index < numVertices; ++index) {

vertexLinks[index] = new Link();

}

for (int i = 0; i < numVertices; ++i) {

vertexLinks[i].vertexA = i;

vertexLinks[i].vertexB = 0;

}

index = 0;

while (edgeCount < numVertices - 1) {

Edge nextEdge = edgeList[index++];

int vertexX = find(vertexLinks, nextEdge.startVertex);

int vertexY = find(vertexLinks, nextEdge.endVertex);

if (vertexX != vertexY) {

mstEdges[edgeCount++] = nextEdge;

union(vertexLinks, vertexX, vertexY);

}

}

}

/\*\*

\* Unites two vertices in the disjoint set.

\*

\* vertexLinks Array of vertex links.

\* vertexX First vertex.

\* vertexY Second vertex.

\*/

void union(Link[] vertexLinks, int vertexX, int vertexY) {

int value1 = find(vertexLinks, vertexX);

int value2 = find(vertexLinks, vertexY);

if (vertexLinks[value1].vertexB > vertexLinks[value2].vertexB)

vertexLinks[value2].vertexA = value1;

else if (vertexLinks[value1].vertexB < vertexLinks[value2].vertexB)

vertexLinks[value1].vertexA = value2;

else {

vertexLinks[value2].vertexA = value1;

vertexLinks[value1].vertexB++;

}

}

/\*\*

\* Finds the representative vertex of a set.

\*

\* vertexLinks Array of vertex links.

\* vertex Vertex to find.

\* return Representative vertex.

\*/

int find(Link[] vertexLinks, int vertex) {

if (vertexLinks[vertex].vertexA != vertex)

vertexLinks[vertex].vertexA = find(vertexLinks, vertexLinks[vertex].vertexA);

return vertexLinks[vertex].vertexA;

}

/\*\*

\* Displays the results of Kruskal's algorithm.

\*

\* startTime Start time of algorithm execution.

\* endTime End time of algorithm execution.

\*/

public void displayResults(long startTime, long endTime) {

System.out.println("Execution Time: " + (endTime - startTime) + " nanoseconds");

int totalCost = 0;

System.out.println("\nMinimum Spanning Tree: \n");

for (int i = 0; i < edgeCount; ++i) {

System.out.println(vertexLabels.charAt(mstEdges[i].startVertex) + " --> " +

vertexLabels.charAt(mstEdges[i].endVertex) + " Cost: " + mstEdges[i].weight);

totalCost += mstEdges[i].weight;

}

System.out.println("\nTotal Cost of MST: " + totalCost);

}

}

**Topological sorting:**

TopologicalSort.java

package Algorithms;

import java.io.\*;

import java.util.\*;

/\*\*

\* Implementation of Topological Sort algorithm.

\*/

public class TopologicalSort {

private Map<String, List<String>> adjacencyList;

private Set<String> exploredVertices;

private Set<String> recursiveCallStack;

private List<List<String>> detectedCycles;

/\*\*

\* Executes the topological sorting algorithm.

\*

\* filePath Path to the input file containing the graph data.

\*/

public void executeTopologicalSorting(String filePath) {

try {

adjacencyList = new HashMap<>();

exploredVertices = new HashSet<>();

recursiveCallStack = new HashSet<>();

detectedCycles = new ArrayList<>();

Scanner fileScanner = new Scanner(new File(filePath));

int numVertices = fileScanner.nextInt();

int numEdges = fileScanner.nextInt();

System.out.println("Number of Vertices: " + numVertices);

System.out.println("Number of Edges: " + numEdges);

String graphType = fileScanner.next();

fileScanner.nextLine();

// Build the graph

for (int i = 0; i < numEdges; i++) {

// Read source and destination vertices

String[] edgeLine = fileScanner.nextLine().split(" ");

String sourceVertex = edgeLine[0];

String destinationVertex = edgeLine[1];

// Add edge to adjacency list

adjacencyList.computeIfAbsent(sourceVertex, k -> new ArrayList<>()).add(destinationVertex);

}

fileScanner.close();

for (String vertex : adjacencyList.keySet()) {

if (!exploredVertices.contains(vertex)) {

isCyclic(vertex, new HashSet<>(), new ArrayList<>());

}

}

// Check if cycles were detected

if (detectedCycles.isEmpty()) {

// Graph is acyclic, perform topological sorting

System.out.println("The graph is acyclic. Performing topological sorting...");

long startTime = System.nanoTime();

List<String> sortedVertices = topologicalSort();

long endTime = System.nanoTime();

System.out.println("Execution Time: " + (endTime - startTime) + " nanoseconds");

System.out.println("Topological sorting sequence:");

for (String vertex : sortedVertices) {

System.out.print(vertex + " ");

}

System.out.println();

} else {

// Graph contains cycles

System.out.println("The graph contains cycles.");

System.out.println("Cycles along with their lengths:");

int cycleIndex = 1;

long startTime = System.nanoTime();

for (List<String> cycle : detectedCycles) {

System.out.print("Cycle " + cycleIndex + ": ");

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() + ")");

cycleIndex++;

}

long endTime = System.nanoTime();

}

} catch (FileNotFoundException e) {

System.out.println("File not found: " + filePath);

e.printStackTrace();

}

}

/\*\*

\* Checks if the graph contains cycles using DFS.

\*

\* vertex Current vertex.

\* visitedVertices Set of visited vertices.

\* currentPath Current path.

\*/

private void isCyclic(String vertex, Set<String> visitedVertices, List<String> currentPath) {

// Check if vertex is already in recursion stack

if (recursiveCallStack.contains(vertex)) {

// Cycle detected, add to detected cycles list

List<String> cycle = new ArrayList<>(currentPath.subList(currentPath.indexOf(vertex), currentPath.size()));

detectedCycles.add(cycle);

return;

}

// Mark vertex as visited and add to recursion stack

if (!exploredVertices.contains(vertex)) {

exploredVertices.add(vertex);

recursiveCallStack.add(vertex);

currentPath.add(vertex);

// Recur for neighboring vertices

List<String> neighbors = adjacencyList.getOrDefault(vertex, new ArrayList<>());

for (String neighbor : neighbors) {

isCyclic(neighbor, visitedVertices, currentPath);

}

// Remove vertex from recursion stack

recursiveCallStack.remove(vertex);

currentPath.remove(currentPath.size() - 1);

}

}

/\*\*

\* Performs topological sorting using DFS.

\*

\* return List of vertices in topological order.

\*/

private List<String> topologicalSort() {

List<String> sortedVertices = new ArrayList<>();

Set<String> visited = new HashSet<>();

Set<String> recursiveCallStack = new HashSet<>();

// Perform DFS traversal

for (String vertex : adjacencyList.keySet()) {

if (!visited.contains(vertex)) {

topologicalSortDFS(vertex, visited, recursiveCallStack, sortedVertices);

}

}

// Reverse the sorted list

Collections.reverse(sortedVertices);

return sortedVertices;

}

/\*\*

\* Recursive DFS function for topological sorting.

\*

\* vertex Current vertex.

\* visited Set of visited vertices.

\* recursiveCallStack Recursion stack.

\* sortedVertices List of sorted vertices.

\*/

private void topologicalSortDFS(String vertex, Set<String> visited, Set<String> recursiveCallStack, List<String> sortedVertices) {

// Mark vertex as visited and add to recursion stack

visited.add(vertex);

recursiveCallStack.add(vertex);

// Recur for neighboring vertices

List<String> neighbors = adjacencyList.getOrDefault(vertex, new ArrayList<>());

for (String neighbor : neighbors) {

if (!visited.contains(neighbor)) {

topologicalSortDFS(neighbor, visited, recursiveCallStack, sortedVertices);

}

}

// Remove vertex from recursion stack and add to sorted list

recursiveCallStack.remove(vertex);

sortedVertices.add(vertex);

}

}