Graph Algorithms and Related Data Structures

PROJECT REPORT

Submitted by:

Varad Deshpande 801243927

Problem 1: Single Source Shortest Path Algorithm

Problem Definition

Find the shortest path tree in both the directed and undirected graph from the given source vertex to all the vertices. Print the path and the path cost for the given source.

Dijkstra's Algorithm

It is used to calculate the shortest distances of all the vertices from a given starting vertex. In this algorithm, we grow a cloud of vertices starting from the source vertex, finally covering all the vertices. For each vertex 'v' we have a label d(v) which is the distance of v from the source vertex in the subgraph consisting of the cloud and its adjacent vertices.

At each step:

- We add a vertex 'v' inside the cloud having smallest value d(v)
- Update the labels of adjacent vertices to 'v' using relaxation function

Dijkstra's algorithm is considered to be a greedy algorithm as it always chooses the vertex with minimum d-value to add to the cloud.

Assumptions

Dijkstra's algorithm makes the following assumptions:

1. The graph is connected
2. The graph is undirected / directed
3. Edge weights are non-negative

Pseudo code

```
INITIALIZE-SINGLE-SOURCE(G, s)
        for each vertex v ∈ G.v
                v.d = ∞                        //Initialize d value to infinity
                v.π = NIL                      //Initialize parent to nil
        s.d                                    //Make the distance of source vertex 0

DIJKSTRA(G,w,s)
        INITIALIZE-SINGLE-SOURCE(G, s)
        S = ∅                                  //Initialize the cloud to empty
        Q = G.v                                //Add all vertice in priority queue
        while Q ≠ ∅
                u = EXTRACT-MIN(Q)             //Pop the vertex with min d-value
```

```
        S = S U {u}                    //Add the vertex to the cloud
        for each vertex v ∈ G.Adj[u]
                RELAX(u, v, w)         //Update the adjacent vertice of u using relax
                                         function

RELAX(u, v, w)
    if(v.d > u.d + w(u, v))            //If current d(v) is greater than d(u) plus the wt
        v.d = u.d + w(u, v)            //Change d(v) to be d(u) + edge wt
        v.π = u                        //Make u the parent of v
```

Runtime Analysis

- In the above pseudo code, the INITIALIZE-SINGLE-SOURCE runs for **O(m)** time.
- If the priority queue is constructed using min heap then the time required for heap construction will be **O(nlogn)**
- Each EXTRACT-MIN operation in the while loop takes **O(logn)** time.
- Each RELAX operation takes time **O(logn)** and there are atmost **O(m)** such operations.
- The total running time is therefore

  O(nlogn + mlogn) or **O((n+m)logn)**

  which is **O(mlogn)** if all vertices are reachable from the source.

Data Structures Used

- Graph data structure has been implemented using an **adjacency list**
- Each vertex will store Edge information like the destination vertex and edge weight in a list for that vertex.
- The graph is thus represented by a list of vertices. With each vertex having edge list as well as its own properties like d-value etc.
- The adjacency list implementation for graphs is space efficient and can be constructed in **O(n + m)** time.
- The query to get an edge can be performed in maximum **O(n)** time.
- The inbuilt priority queue collection in java has been used to represent a priority queue to store the vertice in increasing order of their d values.

Code

```
package algorithms;

import entity.Vertex;

import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Stack;

public class Dijkstra {

   public void findShortestPath(List<Vertex> graph, String startVertex) {
       /* Set the distance of the start vertex to 0 */
       Vertex start = graph.get(graph.indexOf(new Vertex(startVertex, 0, null,
null)));
       start.setdValue(0);
       graph.set(graph.indexOf(start), start);
       Comparator<Vertex> distanceSorter =
Comparator.comparing(Vertex::getdValue);
       /* Add the vertices in the priority queue ordered by the d-value of the
vertices */
       PriorityQueue<Vertex> pq = new PriorityQueue<>(distanceSorter);
       pq.addAll(graph);
       while(!pq.isEmpty()){
           /* Get the minimum value vertex from the priority queue */
           Vertex u = pq.poll();
           u.setVisited(true);
           u.getEdges().forEach(edge -> {
               /* Get the adjacent vertices of the current vertex and perform
the relaxation on these vertices */
               Vertex destinationVertex = edge.getDestination();
               if(!destinationVertex.getVisited()){
                   if(destinationVertex.getdValue() > u.getdValue() +
edge.getDistance()){
                       pq.remove(destinationVertex);
                       destinationVertex.setdValue(u.getdValue() +
edge.getDistance());
                       destinationVertex.setParent(u);
                       pq.offer(destinationVertex);
                   }
               }

           });
       }
       /* Output the shortest distance and path of each vertex from the source
vertex */
       graph.forEach(vertex -> {
           System.out.println("Vertex: " + vertex.getName());
```

```
            System.out.println("Shortest Distance: " + vertex.getdValue());
            Stack<String> stk = new Stack<>();
            stk.push(vertex.getName());
            Vertex parent = vertex.getParent();
            while(parent != null){
                stk.push(parent.getName());
                parent = parent.getParent();
            }
            System.out.printf("Shortest Path: ");
            while(stk.size() != 1){
                System.out.printf(stk.pop() + " --> ");
            }
            System.out.println(stk.pop());
            System.out.println();
        });
    }
}
```

Output

Case 1: Undirected Graph

9 14 U
A B 4
A H 8
B C 8
B H 11
C D 7
C F 4
C I 2
D E 9
D F 14
E F 10
F G 2
G I 6
G H 1
H I 7
A

Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
```

```
Enter your choice:
1
Calculating shortest paths from starting vertex: A
Vertex: A
Shortest Distance: 0
Shortest Path: A

Vertex: B
Shortest Distance: 4
Shortest Path: A --> B

Vertex: H
Shortest Distance: 8
Shortest Path: A --> H

Vertex: C
Shortest Distance: 12
Shortest Path: A --> B --> C

Vertex: D
Shortest Distance: 19
Shortest Path: A --> B --> C --> D

Vertex: F
Shortest Distance: 11
Shortest Path: A --> H --> G --> F

Vertex: I
Shortest Distance: 14
Shortest Path: A --> B --> C --> I

Vertex: E
Shortest Distance: 21
Shortest Path: A --> H --> G --> F --> E

Vertex: G
Shortest Distance: 9
Shortest Path: A --> H --> G


Process finished with exit code 0
```

Case 2: Undirected Graph

```
10 16 U
S A 4
S B 7
S C 7
B A 5
B D 6
B E 3
A D 2
C E 1
E H 10
D F 4
D G 7
D H 4
F G 2
F T 1
G T 10
H T 11
S
```

Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
Enter your choice:
1
Calculating shortest paths from starting vertex: S
Vertex: S
Shortest Distance: 0
Shortest Path: S

Vertex: A
Shortest Distance: 4
Shortest Path: S --> A

Vertex: B
Shortest Distance: 7
Shortest Path: S --> B
```

```
Vertex: C
Shortest Distance: 7
Shortest Path: S --> C

Vertex: D
Shortest Distance: 6
Shortest Path: S --> A --> D

Vertex: E
Shortest Distance: 8
Shortest Path: S --> C --> E

Vertex: H
Shortest Distance: 10
Shortest Path: S --> A --> D --> H

Vertex: F
Shortest Distance: 10
Shortest Path: S --> A --> D --> F

Vertex: G
Shortest Distance: 12
Shortest Path: S --> A --> D --> F --> G

Vertex: T
Shortest Distance: 11
Shortest Path: S --> A --> D --> F --> T


Process finished with exit code 0
```

Case 3: Directed Graph

12 21 D
A B 7
A H 5
A J 3
A F 1
B F 5
F E 11
E D 7
C D 5

```
F C 1
B E 5
G B 5
H G 3
J H 7
J I 1
I H 3
I C 4
C K 3
K L 3
C L 1
I L 1
J L 4
A
```

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
1
Calculating shortest paths from starting vertex: A
Vertex: A
Shortest Distance: 0
Shortest Path: A

Vertex: B
Shortest Distance: 7
Shortest Path: A --> B

Vertex: H
Shortest Distance: 5
Shortest Path: A --> H

Vertex: J
Shortest Distance: 3
Shortest Path: A --> J

Vertex: F
```

```
Shortest Distance: 1
Shortest Path: A --> F

Vertex: E
Shortest Distance: 12
Shortest Path: A --> F --> E

Vertex: D
Shortest Distance: 7
Shortest Path: A --> F --> C --> D

Vertex: C
Shortest Distance: 2
Shortest Path: A --> F --> C

Vertex: G
Shortest Distance: 8
Shortest Path: A --> H --> G

Vertex: I
Shortest Distance: 4
Shortest Path: A --> J --> I

Vertex: K
Shortest Distance: 5
Shortest Path: A --> F --> C --> K

Vertex: L
Shortest Distance: 3
Shortest Path: A --> F --> C --> L

Process finished with exit code 0
```

Case 4: Directed Graph

14 23 D
M Q 2
M R 2
M X 1
M P 10
N O 3
N Q 5

N U 4
O R 3
O S 6
O V 1
P O 2
P S 2
P Z 1
Q T 7
R U 12
R Y 3
S R 5
T N 3
U T 4
V W 9
V X 6
W Z 11
Y V 1
M

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
1
Calculating shortest paths from starting vertex: M
Vertex: M
Shortest Distance: 0
Shortest Path: M

Vertex: Q
Shortest Distance: 2
Shortest Path: M --> Q

Vertex: R
Shortest Distance: 2
Shortest Path: M --> R

Vertex: X
```

```
Shortest Distance: 1
Shortest Path: M --> X

Vertex: P
Shortest Distance: 10
Shortest Path: M --> P

Vertex: N
Shortest Distance: 12
Shortest Path: M --> Q --> T --> N

Vertex: O
Shortest Distance: 12
Shortest Path: M --> P --> O

Vertex: U
Shortest Distance: 14
Shortest Path: M --> R --> U

Vertex: S
Shortest Distance: 12
Shortest Path: M --> P --> S

Vertex: V
Shortest Distance: 6
Shortest Path: M --> R --> Y --> V

Vertex: Z
Shortest Distance: 11
Shortest Path: M --> P --> Z

Vertex: T
Shortest Distance: 9
Shortest Path: M --> Q --> T

Vertex: Y
Shortest Distance: 5
Shortest Path: M --> R --> Y

Vertex: W
Shortest Distance: 15
```

```
Shortest Path: M --> R --> Y --> V --> W

Process finished with exit code 0
```

Problem 2: Minimum Spanning Tree Algorithm

Problem Statement

For a given connected, undirected, weighted graph, find the spanning tree that minimizes the total weight. Print the edges of the tree and the total cost of the minimum spanning tree.

Prim's Algorithm

Minimum Spanning Tree

- Spanning tree is a subset of Graph G, which has all the vertices covered with a minimum possible number of edges.
- A **minimum spanning tree** is a spanning tree whose cumulative edge weights have the smallest value.
- A minimum spanning tree can be thought of as the least cost path that goes through the entire graph and touches every vertex.

Finding Minimum Spanning Tree

Minimum spanning tree can be found out using Prim's Algorithm. This algorithm falls under the greedy category. The algorithm operates like Dijkstra's for finding the shortest path in the graph.

- The tree starts at an arbitrary vertex and grows until it spans all the vertices of the graph.
- Initially the cloud is empty and at each step we add a vertex in the cloud till the cloud has all the vertices.
- At each step, we add a vertex v inside the cloud which has the smallest key label(d(v)).
- We update the label of the vertices adjacent to the selected vertex.

Assumptions

Prim's algorithm makes the following assumptions:

1. The graph is connected.
2. The graph is undirected.
3. The graph is weighted.

Pseudo code

```
PRIMS(G, w, r)
    for each u ∈ G.v                //Initialize each vertex with d(u) as infinity and no
        u.key = ∞                   //parent
        u.π = NIL
    r.key = 0                       //Make d-value of source as 0
    Q = G.v                         //Add the vertice to priority queue
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)          //Extract the vertex with min d-value
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u,v) < v.key  //If the adjacent vertex in queue and has
                v.π = u                  // key greater than the weight of edge
                v.key = w(u,v)           //adjust the key value and set parent
```

Runtime Analysis

- In the above pseudo code, the initialization step(first for loop) runs for **O(m)** time.
- If the priority queue is constructed using min heap then the time required for heap construction will be **O(nlogn)**
- Each EXTRACT-MIN operation in the while loop takes **O(logn)** time.
- Each operation to adjust the key takes time **O(logn)** and there are atmost **O(m)** such operations.
- The total running time is therefore

  O(nlogn + mlogn) or simply **O(mlogn)**

Data Structure Used

- Graph data structure has been implemented using an **adjacency list**
- Each vertex will store Edge information like the destination vertex and edge weight in a list for that vertex.
- The graph is thus represented by a list of vertices. With each vertex having an edge list as well as its own properties like d-value etc.
- The adjacency list implementation for graphs is space efficient and can be constructed in **O(n + m)** time.
- The query to get an edge can be performed in maximum **O(n)** time.
- The inbuilt priority queue collection in java has been used to represent a priority queue to store the vertice in increasing order of their d values.

Code

```
package algorithms;

import entity.Vertex;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

public class Prims {

    public void findMinimumSpanningTree(List<Vertex> graph, String startVertex)
{
        /* Set the distance of the start vertex to 0 */
         Vertex start = graph.get(graph.indexOf(new Vertex(startVertex, 0, null,
null)));
        start.setdValue(0);
        graph.set(graph.indexOf(start), start);
                                          Comparator<Vertex>     distanceSorter     =
Comparator.comparing(Vertex::getdValue);
         /* Add the vertices in the priority queue ordered by the d-value of the
vertices */
        PriorityQueue<Vertex> pq = new PriorityQueue<>(distanceSorter);
        pq.addAll(graph);
        while(!pq.isEmpty()){
            /* Get the minimum value vertex from the priority queue */
            Vertex u = pq.poll();
            u.setVisited(true);
            u.getEdges().forEach(edge -> {
                  /* Get the adjacent vertices of the current vertex and the keys
for these vertices */
                Vertex destinationVertex = edge.getDestination();
                if(!destinationVertex.getVisited()){
                    if(destinationVertex.getdValue() > edge.getDistance()){
                        pq.remove(destinationVertex);
                        destinationVertex.setdValue(edge.getDistance());
                        destinationVertex.setParent(u);
                        pq.offer(destinationVertex);
                    }
                }
            });
        }
        int totalCost = 0;
         /* Output the edges included and the total cost of the minimum spanning
tree */
        for(Vertex vertex : graph){
            if(vertex.getParent() != null){
                 System.out.println("Edge: " + vertex.getParent().getName() + " -
" + vertex.getName() + "\t\tCost: " + vertex.getdValue());
                totalCost += vertex.getdValue();
```

```
            }
        }
        System.out.println("\nTotal Cost of Spanning Tree: " + totalCost);
    }
}
```

Output

Case 1:

9 14 U
A B 4
A H 8
B C 8
B H 11
C D 7
C F 4
C I 2
D E 9
D F 14
E F 10
F G 2
G I 6
G H 1
H I 7
A

Code Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
Enter your choice:
2
Calculating minimum spanning tree from starting vertex: A
Edge: A - B         Cost: 4
Edge: A - H         Cost: 8
Edge: F - C         Cost: 4
Edge: C - D         Cost: 7
Edge: G - F         Cost: 2
Edge: C - I         Cost: 2
Edge: D - E         Cost: 9
```

```
Edge: H - G         Cost: 1

Total Cost of Spanning Tree: 37

Process finished with exit code 0
```

Case 2:

10 16 U
S A 4
S B 7
S C 7
B A 5
B D 6
B E 3
A D 2
C E 1
E H 10
D F 4
D G 7
D H 4
F G 2
F T 1
G T 10
H T 11
S

Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
Enter your choice:
2
Calculating minimum spanning tree from starting vertex: S
Edge: S - A         Cost: 4
Edge: A - B         Cost: 5
Edge: E - C         Cost: 1
Edge: A - D         Cost: 2
Edge: B - E         Cost: 3
Edge: D - H         Cost: 4
```

```
Edge: D - F         Cost: 4
Edge: F - G         Cost: 2
Edge: F - T         Cost: 1

Total Cost of Spanning Tree: 26
```

Case 3:

11 21 U
A B 5
A G 21
A E 12
A J 1
B J 20
B C 9
B G 18
C G 17
C D 16
C K 8
D G 11
D H 14
D F 7
E G 2
E F 6
E I 10
F H 4
F K 13
F J 19
G H 3
I A 15
G

Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's
algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
Enter your choice:
2
Calculating minimum spanning tree from starting vertex: G
Edge: E - A         Cost: 12
```

```
Edge: A - B         Cost: 5
Edge: G - E         Cost: 2
Edge: A - J         Cost: 1
Edge: B - C         Cost: 9
Edge: F - D         Cost: 7
Edge: C - K         Cost: 8
Edge: G - H         Cost: 3
Edge: H - F         Cost: 4
Edge: E - I         Cost: 10

Total Cost of Spanning Tree: 61

Process finished with exit code 0
```

Case 4:

```
9 16 U
A B 4
B C 11
B D 9
C A 8
D C 7
D E 2
D F 6
E B 8
E G 7
E H 4
F C 1
F E 5
G H 14
G I 9
H F 2
H I 10
A
```

Execution

```
The Graph provided is undirected Graph
Please select the action for undirected graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Calculate Minimum spanning tree using Prim's algorithm
```

```
Enter your choice:
2
Calculating minimum spanning tree from starting vertex: A
Edge: A - B          Cost: 4
Edge: A - C          Cost: 8
Edge: E - D          Cost: 2
Edge: H - E          Cost: 4
Edge: C - F          Cost: 1
Edge: E - G          Cost: 7
Edge: F - H          Cost: 2
Edge: G - I          Cost: 9

Total Cost of Spanning Tree: 37

Process finished with exit code 0
```

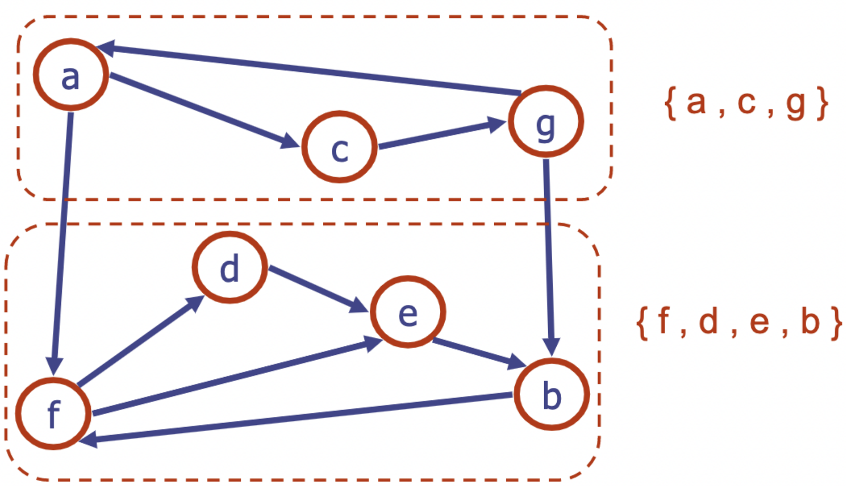
Problem 3: Finding Strongly Connected Components

Problem Statement

Given a directed graph with n vertices and m edges, find the strongly connected components in the graph.

Finding Strongly Connected Components

- A Strongly Connected Component(SCC) or Maximal subgraph is a subgraph where each vertex can reach all other vertices in the subgraph.



- The Depth First Search(DFS) algorithm can be modified further to find the strongly connected components in the digraph.
- We need to calculate the DFS of the original graph and its transpose to find out the strongly connected components.
- A transpose of a directed graph is another directed graph on the same set of vertices with direction of all the edges reversed.
- The DFS traversal of these two graphs helps us to find the strongly connected components.

Assumptions

We have the following assumptions while calculating the SCC algorithm:

1. The given graph is a directed graph.

Pseudo code

STRONGLY-CONNECTED-COMPONENTS(G)

1. Call DFS(G) to calculate the finish times u.f of each vertex u in the graph.
2. Compute G^T (Transpose)
3. Call DFS(G^T), but in the main loop of DFS, consider vertices in decreasing order of u.f (As computed on line 1)
4. Output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected component.

Runtime Analysis

- The above pseudo code does two things:
    - Calculates the DFS of a graph(Directed graph + Transpose)
    - Calculates the transpose of a graph
- The time required to calculate DFS of a graph is **O(n + m)**
- If we have an adjacency list representation of a graph, then its transpose can be calculated in **O(n + m)** time.
- Therefore the total time required to find the strongly connected components of a digraph is **O(n + m).**

Data Structures Used

- Graph data structure has been implemented using an **adjacency list**
- Each vertex will store Edge information like the destination vertex and edge weight in a list for that vertex.
- The graph is thus represented by a list of vertices. With each vertex having an edge list as well as its own properties like d-value etc.

- The adjacency list implementation for graphs is space efficient and can be constructed in **O(n + m)** time.
- The query to get an edge can be performed in maximum **O(n)** time.
- The inbuilt priority queue collection in java has been used to represent a priority queue to store the vertice in increasing order of their d values.

Code

```java
package algorithms;

import entity.Edge;
import entity.Vertex;
import java.util.List;

public class DFS {

   Integer time;

   public void dfs(List<Vertex> graph){
       time = 0;
       System.out.println("DFS: ");
       for (Vertex u: graph) {
           /* Call DFS-VISIT for each node that has not been visited */
           if(u.getColor().equals("WHITE")){
               dfsVisit(graph, u);
           }
       }
   }

   /* Function to calculate the dfs of the transpose graph */
   public void dfsTranspose(List<Vertex> graph){
       time = 0;
       int i =1;
       for (Vertex u: graph) {
           if(u.getColor().equals("WHITE")){
               System.out.println("\nComponent " + i + ": ");
               dfsVisit(graph, u);
               i++;
           }
       }
   }

   public void dfsVisit(List<Vertex> graph, Vertex u){
       /* Increase the time and set the start time of the vertex */
       time++;
       u.setD(time);
```

```java
        /* Set the color of the vertex to GRAY */
        u.setColor("GRAY");
        for (Edge e: u.getEdges()) {
            /* For each adjacent vertex of the current vertex perform DFS
VISIT if not already visited */
            Vertex v = e.getDestination();
            if(v.getColor().equals("WHITE")){
                v.setParent(u);
                dfsVisit(graph, v);
            }
        }
            /* Set the color of the vertex to be BLACK as all the adjacent
vertices have been handled */
        u.setColor("BLACK");
        System.out.printf(u.getName() + "\t");
        /* Set the finish time for the vertex */
        time++;
        u.setF(time);
    }
}

package algorithms;

import entity.Vertex;
import utils.Utility;
import java.util.List;

public class SCC {

    public void stronglyConnected(List<Vertex> graph){
        DFS dfs = new DFS();
        /* Calculate DFS of the graph */
        dfs.dfs(graph);
         /* Calculate the transpose and order the vertices in decreasing order of
finish time as of DFS */
        List<Vertex> transpose = Utility.getTranspose(graph);
         System.out.println("\n\nFollowing are the strongly connected components
in the graph");
        /* Call DFS on the transpose graph */
        dfs.dfsTranspose(transpose);
    }
}
```

Output

Case 1:

```
11 18 D
A B 2
B C 3
B E 1
C D 7
D C 1
D G 5
E B 6
E D 2
E F 2
F H 1
F G 4
G J 3
H I 7
H J 1
I F 4
J K 2
K H 1
K J 2
A
```

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
1.  Calculate  single  source  shortest  path  using  Dijkstra's
algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
2
Calculating strongly connected components for the given graph
DFS:
F    I    H    K    J    G    D    C    E    B    A

Following are the strongly connected components in the graph

Component 1:
A
Component 2:
```

```
E    B
Component 3:
D    C
Component 4:
J    K    H    I    F    G

Process finished with exit code 0
```

Case 2:

```
11 18 D
A B 5
A C 1
A J 3
B C 2
B K 4
C E 1
D J 2
D G 6
E A 3
F D 3
F I 7
G K 5
H F 1
I A 8
I E 1
I H 2
J G 5
K J 7
A
```

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
1. Calculate single source shortest path using Dijkstra's algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
```

```
2
Calculating strongly connected components for the given graph

DFS:
E    C    G    J    K    B    A    D    H    I    F

Following are the strongly connected components in the graph

Component 1:
I    H    F
Component 2:
D
Component 3:
B    C    E    A
Component 4:
J    G    K

Process finished with exit code 0
```

Case 3:

10 14 D
A C 3
A H 4
B A 5
B G 6
C D 7
D F 1
E A 2
E I 2
F J 3
G I 4
H F 1
H G 3
I H 5
J C 2
A

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
```

```
1.  Calculate  single  source  shortest  path  using  Dijkstra's
algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
2
Calculating strongly connected components for the given graph

DFS:
J    F    D    C    I    G    H    A    B    E

Following are the strongly connected components in the graph

Component 1:
E
Component 2:
B
Component 3:
A
Component 4:
G    I    H
Component 5:
D    F    J    C

Process finished with exit code 0
```

Case 4:

```
11 17 D
A B 4
B C 3
B D 1
B F 3
C A 2
C E 2
C H 3
D G 4
E D 5
E H 1
E K 5
F B 2
F D 1
G I 4
```

I D 4
J E 5
K J 6
A

Execution

```
The Graph provided is directed Graph
Please select the action for directed graph:
1.  Calculate  single  source  shortest  path  using  Dijkstra's
algorithm
2. Find Strongly Connected Components for the given Digraph
Enter your choice:
2
Calculating strongly connected components for the given graph
DFS:
I    G    D    H    J    K    E    C    F    B    A

Following are the strongly connected components in the graph

Component 1:
F    B    C    A
Component 2:
K    J    E
Component 3:
H
Component 4:
G    I    D

Process finished with exit code 0
```

Data Structure and Utility Code

Vertex Code

```java
package entity;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Vertex {
   String name;
   Integer dValue;
   List<Edge> edges;
   Vertex parent;
   Boolean visited;
   Integer d;
   Integer f;
   String color;

   public Vertex(String name, Integer dValue, List<Edge> edges, Vertex parent)
{
       this.name = name;
       this.dValue = dValue;
       this.edges = edges;
       this.parent = parent;
       this.visited = false;
       this.color = "WHITE";
   }

   public Vertex(Vertex another){
       this.name = another.getName();
       this.dValue = another.getdValue();
       this.edges = new ArrayList<Edge>();
       this.parent = null;
       this.visited = false;
       this.color = "WHITE";
       this.d = another.getD();
       this.f = another.getF();
   }



   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
```

```java
    }

    public Integer getdValue() {
        return dValue;
    }

    public void setdValue(Integer dValue) {
        this.dValue = dValue;
    }

    public List<Edge> getEdges() {
        return edges;
    }

    public void setEdges(List<Edge> edges) {
        this.edges = edges;
    }

    public Vertex getParent() {
        return parent;
    }

    public void setParent(Vertex parent) {
        this.parent = parent;
    }

    public Boolean getVisited() {
        return visited;
    }

    public void setVisited(Boolean visited) {
        this.visited = visited;
    }

    public Integer getD() {
        return d;
    }

    public void setD(Integer d) {
        this.d = d;
    }

    public Integer getF() {
        return f;
    }

    public void setF(Integer f) {
        this.f = f;
    }
```

```java
    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Vertex vertex = (Vertex) o;
        return Objects.equals(name, vertex.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return "Vertex{" +
                "name='" + name + '\'' +
                ", dValue=" + dValue +
                ", edges=" + edges +
                ", parent='" + (parent==null ? "NIL" : parent.getName()) + '\''
+
                ", visited=" + visited +
                '}';
    }
}
```

Edge Code

```java
package entity;

import java.util.Objects;

public class Edge {
    Vertex destination;
    Integer distance;

    public Edge(Vertex destination, Integer distance) {
        this.destination = destination;
        this.distance = distance;
    }
```

```
    public Vertex getDestination() {
        return destination;
    }

    public void setDestination(Vertex destination) {
        this.destination = destination;
    }

    public Integer getDistance() {
        return distance;
    }

    public void setDistance(Integer distance) {
        this.distance = distance;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Edge edge = (Edge) o;
        return Objects.equals(destination, edge.destination) &&
Objects.equals(distance, edge.distance);
    }

    @Override
    public int hashCode() {
        return Objects.hash(destination, distance);
    }

    @Override
    public String toString() {
        return "Edge{" +
                "destination=" + destination.getName() +
                ", distance=" + distance +
                '}';
    }
}
```

Create Transpose Utility

```
package utils;

import entity.Edge;
import entity.Vertex;

import java.util.ArrayList;
import java.util.Comparator;
```

```java
import java.util.List;

public class Utility {

    public static List<Vertex> getTranspose(List<Vertex> graph){
        List<Vertex> transpose = new ArrayList<>();
        graph.forEach(v -> {
            v.getEdges().forEach(edge -> {
                Vertex end = new Vertex(v);
                Vertex start = new Vertex(edge.getDestination());
                Integer distance = edge.getDistance();
                /* Reverse the edges of the graph */
                if(!transpose.contains(start)){
                    transpose.add(start);
                } else {
                    start = transpose.get(transpose.indexOf(start));
                }
                if(!transpose.contains(end)){
                    transpose.add(end);
                } else {
                    end = transpose.get(transpose.indexOf(end));
                }
                start.getEdges().add(new Edge(end, distance));
            });
        });
        /* Order the vertices in the decreasing order of finish time */
        Comparator<Vertex> distanceSorter =
Comparator.comparing(Vertex::getF).reversed();
        transpose.sort(distanceSorter);
        transpose.forEach(vertex -> {
            vertex.setD(0);
            vertex.setF(0);
        });
        return transpose;
    }
}
```

Conclusion

Thus we have studied and implemented algorithms to calculate:

1. Single source shortest path
2. Minimum spanning tree
3. Strongly connected components

We have also done a detailed runtime analysis of these algorithms based on the implementations and data structures used and stated the complexity of them respectively.