# ITCS – 6114: Project 2

**Submitted By: Ankur Huralikoppi**

Student ID: 800782775
ahuralik@uncc.edu

# Algorithms & Data Structures: ITCS 6114

*Project 2*

**Introduction:**

This report provides the details of implementation, algorithm analysis and outputs of the input files provided for:

1. Verifying Graph Connectivity
2. Establishing the minimal spanning tree(MST)
3. Finding the Shortest Paths to all other vertices from '1'.

**Approach:**

The below algorithms were chosen to implement the above mentioned problems:

1. Breadth First Search
2. Prims MST Algorithm with Priority Queue.
3. Dijkstra Shortest Path Algorithm with Priority Queue.

Implementation details:

The major classes and their functionalities are given below:

1. The Graph Class
   The Graph class holds members for:
   - Vertices
   - Edges
   - Adjacency Matrices for the graph, MST and the shortest Paths.
   - A list of vertices which mark start of disconnected components.
   - A flag that is true if the graph is connected.

   The graph class also holds methods to do the following:
   - Read and setup the graph details from the input file
   - Run BFS for connectivity
   - Print adjacency Matrices
   - Setters & Getters for the above mentioned data members.
   - To set/reset vertex colors, get the edge weight given two vertices as input.

2. The PrimsMinimalSpanningTree class
   The PrimsMinimalSpanningTree class hold members for:
   - The graph that the Prims algorithm will run on.
   - A priority queue which is used for the algorithm

   The PrimsMinimalSpanningTree has method for Finding out the minimal spanning tree.

3. The DijkstraShortestPath Class
   The DijkstraShortestPath class holds members for:
   - The graph that the Prims algorithm will run on.
   - A priority queue which is used for the algorithm

   The DijkstraShortestPath has a method for finding out the shortest paths for the graph.

4. The PriorityQueueElement Class
   The PriorityQueueElement class has members for:
   - The vertex
   - The cost
   - The parent

   The PriorityQueueElement has only setters and getters.
   **NOTE: The class structure resembles the structure of the priority queue studied in the class.**

5. The PriorityQueue Class
   The PriorityQueue class is a wrapper around the java.util.PriorityQueue class. The priority queue class has the a wrapped method around the base java.util.PriorityQueue method for the priority queue operations, with additional methods for updation of elements when the algorithm determines there's a better route to reach a particular vertex.

   According to the Oracle documentation of the java.util.PriorityQueue class, the operations and their complexities are:

   | | |
   |---|---|
   | 1. Insert() | O(log(V)) |
   | 2. Delete() | O(log(V)) |
   | 3. IsEmpty() | constant time[because it does a return size==0; |
   | 4. Contains() | θ(V) |

   **SOURCE: http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html**

6. The PriorityQueueElementComparator Class
   The PriorityQueueElementComparator class implements the java.util.Comparator<T> interface which determines the relative ordering of the elements in the PriorityQueue. The PriorityQueueElement with the least cost is the highest priority element at any given point of time in the queue.

Finding Graph Connectivity:

Breadth First Search (BFS) was used to establish the connectivity of the graph. Given a graph, the algorithm begins by marking every vertex as "Unvisited". Starting from vertex '1', the algorithm runs BFS on it. The BFS algorithm uses a Queue to maintain details about the connectivity among vertices and if a vertex is reachable, it is marked "Reachable". The pseudo code of the algorithm and the complexities occurring is highlighted below:

```
checkConnectivity(G){
  foreach v ϵ G.V do{                    //This loop results in a linear complexity against the vertices. i.e. ϑ(V)
    if (v is unvisited) do {
        componentStartVertex.add(G, v);  //a graph component starts with this vertex.
        doBFS();
    }
  }
}

doBFS(G, v){
  Queue q = φ;
  q.enqueue(v);
  while (q != φ) do {                    //This loop runs in a linear complexity against the edges. i.e. ϑ(E)
    u = q.dequeue();
```

```
      foreach x ϵ Adjacent[u] do {
         if (x is unvisited) do
            q.enqueue(x);
      u.setVisited();
      }
   }
}
```

The implementation of the above had additional complexities associated with the queue insertion, deletion and getting the list of adjacent vertices. The BFS starts with running through all the vertices, and calling the doBFS( ) method that will mark the vertices, those are reachable from the vertex passed to the method, as "Visited". If there are more than one disconnected components, the method checkConnectivity() will have vertices with "Unvisited" status when the calls to doBFS return.

Implementation Details:

For the implementation, the method is called directly on the Graph class object that is created after reading the file.

Complexity Analysis:

From the above algorithm, the line 'for v ϵ G.V' has a linear complexity of θ(V) since it runs through all the vertices once. The line while (q != ϕ) has a complexity linear to the edges since the adjacency about a vertex is related to the number of edges projected out or in from the vertex.

Given the above algorithm, the BFS algorithm boils down to **θ(V + E)**

Finding the Minimal Spanning Tree:

The below is the algorithm used for MST using Prims Algorithm:

```
for(Integer i : G.ComponentStartVertices){
   pq.enqueue(i, -1, 0.0f);            //has a complexity of O(lg V)

   while(!pq.isEmpty()){               //this while loop runs at the max 'V' times, i.e. O(V)
      vertex = pq.dequeue();           //the delete operation has a complexity of O(lg V)
      foreach(adjacentVertex : Adjacent(vertex))){ //Has a complexity of O(E), since this runs linearly on adjacent edges.
            if(!pq.checkAndUpdate(adjacentVertex, vertex, w(vertex, adjacentVertex))){
               if(!g.isVertexProcessed(adjacentVertex)) //The checkAndUpdate method has a complexity of O(V+lgV)
                  pq.enqueue(adjacentVertex, vertex, w(vertex, adjacentVertex));
               }
            }

            if(vertex.getParent() != -1){
               g.setMinimalSpanningTreeAdjMatrixValue(vertex.Parent, vertex, vertex.Cost);
            }
            g.setVertexProcessed(vertex);
      }
}
```

Algorithm Analysis:

The algorithm has to run for every component of the graph. Since the Graph class already has a list that has the start vertices of all the disconnected components, the algorithm begins by running the Prim's algorithm for every sub-graph. The algorithm finds out a MST for each of the components.

The priority queue has been organized as a min-heap. The priority queue contains the least reachable cost and the parent through which the vertex has to be reached in the MST at any given point of time. The algorithm initially pushes the start vertex into the queue, and then starts growing the MST by choosing the smallest possible cost available from the reachable set of vertices from the current vertex that is deleted.

Complexity:

Since a Min-heap is used as a priority queue, the insert & delete operations pose a complexity of O(lg V). The enqueue operation in the second line of the algorithm would cost O(lg V). The while(!pq.isEmpty()) runs 'V' times since the queue will begin with having one vertex, and eventually have all the vertices. Thus, this loop runs 'v' times and has a complexity of O(V). The first line inside this loop is a delete operation which causes a complexity of O(lg V). The next line, for(adjacentVertex: Adjacent(vertex)) causes a complexity of O(E) since adjacency is dependent linearly on edges. Since the implementation uses Adjacency matrix, it takes an additional complexity of O(V) to get the adjacent vertices. The next three lines check if an adjacent vertex is already present, if it does, the algorithm checks if the weight from the current vertex to the adjacent vertex is less than the cost present in the queue. If it's lower than that in the queue, the algorithm updates that entry and heapfies the heap. This method causes a complexity of O(V + 2Lg V). 'V' complexity since the algorithm linearly looks in the queue to find if the vertex is already present. And if the element is found, it's deleted and a new entry is added. If the element is not present in the queue, then the vertex is added. This operation again costs O(lg V). Considering all these complexities and operations which runs inside loops, the overall complexity can be reduced as:

- ➔ Time for first insert + (while Loop * queue dequeue * complexity to get adjacent vertices) + adjacency Loop * ( pq exchange + pq insert))
- ➔ lg v + {(V * lg V * V) + E *(V + 2lg V))} [omit 2, since it's a constant]
- ➔ lg V + {V^2.lg V + EV + Elg V} [omit 2, since it's a constant]
- ➔ lg V + V^2.lg V + EV + Elg V [we can omit lg V since lg V ϵ O(EVlg V)]
- ➔ **V^2.lg V + EV + Elg V**

**The complexity of the MST using Priority queue is reduced to a quadratic complexity in V as O(V^2.Lg V + EV + ELg V)**

Finding Shortest Path from '1' to rest of the vertices:

Dijkstra's algorithm with Priority Queue was used to find the shortest paths from '1' to rest of the vertices. The below algorithm was used for the same:

```
pq.enqueue(i, -1, 0.0f);            //has a complexity of O(lg V)

  while(!pq.isEmpty()){             //this while loop runs at the max 'V' times, i.e. O(V)
    vertex = pq.dequeue();          //the delete operation has a complexity of O(lg V)
    foreach(adjacentVertex : Adjacent(vertex))){ //Has a complexity of O(E), since this runs linearly on adjacent edges.
        if(!pq.checkAndUpdate(adjacentVertex, vertex, (vertex.Cost + w(vertex, adjacentVertex)))){
          if(!g.isVertexProcessed(adjacentVertex)) //The checkAndUpdate method has a complexity of O(V+lgV)
            pq.enqueue(adjacentVertex, vertex, (vertex.Cost + w(vertex, adjacentVertex)));
          }
        }

        if(vertex.getParent() != -1){
          g.setMinimalSpanningTreeAdjMatrixValue(vertex.Parent, vertex, vertex.Cost);
        }
        g.setVertexProcessed(vertex);
    }
```

Algorithm Analysis:
The Dijkstra's algorithm uses the priority queue similar to the way as in the Prim's algorithm. The only difference is the cost has to be cumulatively calculated since the evaluation is from vertex '1' to the rest of the vertices. The parts of algorithm that differs from Prim's algorithm are highlighted in bold in the above algorithm.

Complexity:
The complexity of Dijkstra's algorithm is the same as Prims algorithm for this implementation since there is no change in the logic or the sequence of operations, but only the cost that is saved in the Priority Queue changes. Thus, the complexity of Dijkstra's algorithm remains **O(V^2.Lg V + EV + ELg V).**

**Findings:**

- Although a Priority Queue is used for implementation, the complexity turns out to be quadratic. This is because of the method '*checkAndUpdate'*. This method is used since the operation Decrease-Min is not provided in the Priority Queue implementation of java. The *checkAndUpdate* method linearly searches the queue to see if the passed vertex is present or not, if it does, then it replaces it. For the exchange, we had to do a remove and insert because, there is no provision to do a '*Heapify*' option. This caused additional complexity. And since this operation is called inside a while loop which runs 'V' times, the complexity was raised to a quadratic term in V.
- In all the algorithms, the Adjacent[v] is implemented as a linear search in the adjacency matrix in the row corresponding to 'v'. This has added an additional complexity of 'V' since the linear search needs to check 'V' number of entries in the matrix.

**Improvements:**

- If an adjacency list was built along with the adjacency matrix, all the operations on getting the adjacent vertices would reduce to constant time operations. But, it would require an additional space of θ(VE). But, this implementation would help reduce the algorithm complexity by a factor of 'V'!
- If a min-heap which supported the Decrease-Min and Exchange operations in an optimal way was used, the complexity of ( V + 2lg V) would be reduced only to lg V. The java.util.PriorityQueue doesn't support these operations because the nature of the heap, i.e. Min or Max heap, is solely decided by the Comparator that is passed to it. It's not determined while writing the interface as to which type of a heap will be created using this interface.

Unsorted Array V/s Sorted Array V/s Heaps for algorithms:

- While using unsorted arrays, the insertion takes a constant time and doesn't add anything to the overall complexity. But the delete operations have a linear complexity since the element with the highest priority has to be linearly searched.
- If a sorted array is used, then the insertion would take a linear time since we'd have to search for the right position of the element in the sorted array. Deletion would take a constant time in this case since the array is sorted and we know where the highest priority element sits.
- Heaps are more tricky data structures to be used. Although insert, delete and exchange operations are in lg V complexities, the running time of the algorithm for very dense graphs reaches to quadratic terms in V against an expected time of (V + E)lg V. Heaps don't support a proper search operation. Searching in heap is required when we want to check if an adjacent vertex is already present in the heap, and if it does, then exchange it. For the search, we'll have to do a linear search for it. We can have an extra flag in the list of vertices which hold a 'true' value if the vertex is present in the queue and 'false' if it doesn't. We can set/reset this flag when we do the enqueue/dequeue operations. With this setup, the check for the vertex present in the list is a constant time operation. By this, we can avoid running through the whole list and then determining the vertex is absent.

  There are specialized data structures for heaps which have a very optimal complexity for operations line decrease-min etc, like Fibonacci Heaps.

  A combination of Fibonacci Heaps and graph represented as an adjacency list is expected to give the most optimal running time.

Outputs:

**NOTE: Nature of the adjacency matrix for Dijkstra's algorithm is as below:**
**If an entry [i][j] has a value 'x', it means that the cost from the vertex 1 to vertex 'j' is 'x' and the parent of 'j' is 'i**'.

1. Output of input 1

---

Adjacency Matrix for the input graph:

---

```
0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0  20.0 0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  3.0  5.0
7.0  0.0  0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  7.0  0.0  0.0  10.0 5.0  0.0  15.0 0.0  7.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  10.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  5.0  20.0 0.0  0.0
0.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0  0.0  6.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  0.0  5.0  0.0    0.0  10.0 0.0    0.0  6.0  0.0  0.0  0.0  0.0
0.0  0.0  3.0  15.0 0.0  0.0  2.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
20.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0  12.0 0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  7.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0    0.0  0.0  50.0   0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0  0.0    10.0 0.0  100.0  12.0 0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  10.0   0.0  0.0  0.0    0.0  0.0  33.0 0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  10.0 0.0  12.0 0.0  0.0    0.0  0.0  8.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  50.0 100.0  0.0  8.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  12.0   0.0  0.0  0.0    0.0  0.0  0.0  5.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0  0.0  0.0    33.0 0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  20.0 0.0  0.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    5.0  0.0  0.0  0.0  0.0  0.0
3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
5.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0  0.0  0.0    0.0  0.0  0.0  0.0  0.0  0.0
```

---

Evaluating Graph Connectivity using BFS.

---

Graph is connected.

---

Evaluating Minimal Spanning Tree with Prim's Algorithm.

---

Evaluating sub component with vertex '1'

---

Edge from 1 to 19 is in the min span tree with cost: 3.0
Edge from 1 to 20 is in the min span tree with cost: 5.0
Edge from 1 to 2 is in the min span tree with cost: 7.0
Edge from 2 to 4 is in the min span tree with cost: 7.0
Edge from 4 to 6 is in the min span tree with cost: 5.0
Edge from 6 to 11 is in the min span tree with cost: 6.0
Edge from 4 to 10 is in the min span tree with cost: 7.0
Edge from 10 to 7 is in the min span tree with cost: 5.0
Edge from 7 to 8 is in the min span tree with cost: 2.0
Edge from 8 to 3 is in the min span tree with cost: 3.0
Edge from 7 to 16 is in the min span tree with cost: 6.0
Edge from 7 to 13 is in the min span tree with cost: 10.0
Edge from 13 to 14 is in the min span tree with cost: 8.0

Edge from 4 to 5 is in the min span tree with cost: 10.0
Edge from 5 to 17 is in the min span tree with cost: 5.0
Edge from 11 to 12 is in the min span tree with cost: 10.0
Edge from 11 to 15 is in the min span tree with cost: 12.0
Edge from 15 to 18 is in the min span tree with cost: 5.0
Edge from 13 to 9 is in the min span tree with cost: 12.0

Adjacency Matrix for the Minimal Spanning Forest:

---

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 7.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.0 | 5.0 |
| 7.0 | 0.0 | 0.0 | 7.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 7.0 | 0.0 | 0.0 | 10.0 | 5.0 | 0.0 | 0.0 | 0.0 | 7.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 6.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2.0 | 0.0 | 5.0 | 0.0 | 0.0 | 10.0 | 0.0 | 0.0 | 6.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 | 2.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 12.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 7.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 6.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | 0.0 | 12.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | 12.0 | 0.0 | 0.0 | 0.0 | 0.0 | 8.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 8.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 12.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 6.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

---

Evaluating Shortest Paths from Vertex 1 to the rest using Dijkstra's Algorithm.

---

Path from 1 to 19 is via 1 with cost: 3.0
Path from 1 to 20 is via 1 with cost: 5.0
Path from 1 to 2 is via 1 with cost: 7.0
Path from 1 to 4 is via 2 with cost: 14.0
Path from 1 to 6 is via 4 with cost: 19.0
Path from 1 to 9 is via 1 with cost: 20.0
Path from 1 to 10 is via 4 with cost: 21.0
Path from 1 to 5 is via 4 with cost: 24.0
Path from 1 to 11 is via 6 with cost: 25.0
Path from 1 to 7 is via 10 with cost: 26.0
Path from 1 to 8 is via 7 with cost: 28.0
Path from 1 to 17 is via 5 with cost: 29.0
Path from 1 to 3 is via 8 with cost: 31.0
Path from 1 to 16 is via 7 with cost: 32.0
Path from 1 to 13 is via 9 with cost: 32.0
Path from 1 to 12 is via 11 with cost: 35.0
Path from 1 to 15 is via 11 with cost: 37.0
Path from 1 to 14 is via 13 with cost: 40.0
Path from 1 to 18 is via 15 with cost: 42.0

Adjacency matrix for Shortest paths.

```
---------------------------------------------------------
0.0  7.0  0.0  0.0  0.0  0.0  0.0  0.0  20.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0  5.0
7.0  0.0  0.0  14.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  31.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  14.0 0.0  0.0  24.0 19.0 0.0  0.0  0.0  21.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  24.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  29.0 0.0  0.0  0.0
0.0  0.0  0.0  19.0 0.0  0.0  0.0  0.0  0.0  0.0  25.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  28.0 0.0  26.0 0.0  0.0  0.0  0.0  0.0  32.0 0.0  0.0  0.0  0.0
0.0  0.0  31.0 0.0  0.0  0.0  28.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
20.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  32.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  21.0 0.0  0.0  26.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  25.0 0.0  0.0  0.0  0.0  0.0  35.0 0.0  0.0  37.0 0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  35.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  32.0 0.0  0.0  0.0  0.0  40.0 0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  40.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  37.0 0.0  0.0  0.0  0.0  0.0  0.0  42.0 0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  32.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  29.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  42.0 0.0  0.0  0.0  0.0  0.0
3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
5.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

2. Output of input 2

```
-------------------------------------------------------------------------------------------------
Adjacency Matrix for the input graph:
-------------------------------------------------------------------------------------------------
0.0    1.2    0.0    0.0    0.0    0.0    0.0    0.0    3.0    0.0
1.2    0.0    0.8    0.0    0.5    0.0    0.0    0.0    0.0    0.0
0.0    0.8    0.0    0.0    0.0    3.1    0.0    0.0    0.0    1.5
0.0    0.0    0.0    0.0    1.5    0.0    0.0    0.0    3.2    0.0
0.0    0.5    0.0    1.5    0.0    0.0    2.0    5.1    0.0    0.0
0.0    0.0    3.1    0.0    0.0    0.0    5.5    0.0    0.0    0.0
0.0    0.0    0.0    0.0    2.0    5.5    0.0    0.0    0.0    0.0
0.0    0.0    0.0    0.0    5.1    0.0    0.0    0.0    0.0    8.8
3.0    0.0    0.0    3.2    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    1.5    0.0    0.0    0.0    0.0    8.8    0.0    0.0

-------------------------------------------------------------------------------------------------
Evaluating Graph Connectivity using BFS.
-------------------------------------------------------------------------------------------------
Graph is connected.

-------------------------------------------------------------------------------------------------
Evaluating Minimal Spanning Tree with Prim's Algorithm.
-------------------------------------------------------------------------------------------------

Evaluating sub component with vertex '1'
---------------------------------------------------------
Edge from 1 to 2 is in the min span tree with cost: 1.2
Edge from 2 to 5 is in the min span tree with cost: 0.5
Edge from 2 to 3 is in the min span tree with cost: 0.8
Edge from 5 to 4 is in the min span tree with cost: 1.5
Edge from 3 to 10 is in the min span tree with cost: 1.5
```

```
Edge from 5 to 7 is in the min span tree with cost: 2.0
Edge from 1 to 9 is in the min span tree with cost: 3.0
Edge from 3 to 6 is in the min span tree with cost: 3.1
Edge from 5 to 8 is in the min span tree with cost: 5.1

Adjacency Matrix for the Minimal Spanning Forest:
-----------------------------------------------------------
0.0   1.2   0.0   0.0   0.0   0.0   0.0   0.0   3.0   0.0
1.2   0.0   0.8   0.0   0.5   0.0   0.0   0.0   0.0   0.0
0.0   0.8   0.0   0.0   0.0   3.1   0.0   0.0   0.0   1.5
0.0   0.0   0.0   0.0   1.5   0.0   0.0   0.0   0.0   0.0
0.0   0.5   0.0   1.5   0.0   0.0   2.0   5.1   0.0   0.0
0.0   0.0   3.1   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   2.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   5.1   0.0   0.0   0.0   0.0   0.0
3.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   1.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0

-------------------------------------------------------------------------------------------------------
--------
Evaluating Shortest Paths from Vertex 1 to the rest using Dijkstra's Algorithm.
-------------------------------------------------------------------------------------------------------
--------
Path from 1 to 2 is via 1 with cost: 1.2
Path from 1 to 5 is via 2 with cost: 1.7
Path from 1 to 3 is via 2 with cost: 2.0
Path from 1 to 9 is via 1 with cost: 3.0
Path from 1 to 4 is via 5 with cost: 3.2
Path from 1 to 10 is via 3 with cost: 3.5
Path from 1 to 7 is via 5 with cost: 3.7
Path from 1 to 6 is via 3 with cost: 5.1
Path from 1 to 8 is via 5 with cost: 6.8

Adjacency matrix for Shortest paths.
-----------------------------------------------------
0.0   1.2   0.0   0.0   0.0   0.0   0.0   0.0   3.0   0.0
1.2   0.0   2.0   0.0   1.7   0.0   0.0   0.0   0.0   0.0
0.0   2.0   0.0   0.0   0.0   5.1   0.0   0.0   0.0   3.5
0.0   0.0   0.0   0.0   3.2   0.0   0.0   0.0   0.0   0.0
0.0   1.7   0.0   3.2   0.0   0.0   3.7   6.8   0.0   0.0
0.0   0.0   5.1   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   3.7   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   6.8   0.0   0.0   0.0   0.0   0.0
3.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   3.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

3. Output for input 3

```
-----------------------------------------------------------------------------------------------
Adjacency Matrix for the input graph:
-----------------------------------------------------------------------------------------------
0.0    0.0    0.0    2.3    2.4    0.0    0.0    0.0    1.5    0.0
0.0    0.0    1.5    0.0    0.0    0.0    0.0    8.2    0.0    6.3
0.0    1.5    0.0    0.0    0.0    3.2    0.0    0.0    0.0    5.6
2.3    0.0    0.0    0.0    3.1    0.0    8.3    0.0    0.0    0.0
2.4    0.0    0.0    3.1    0.0    0.0    0.0    0.0    5.6    0.0
0.0    0.0    3.2    0.0    0.0    0.0    0.0    3.1    0.0    0.0
0.0    0.0    0.0    8.3    0.0    0.0    0.0    0.0    0.8    0.0
0.0    8.2    0.0    0.0    0.0    3.1    0.0    0.0    0.0    0.0
1.5    0.0    0.0    0.0    5.6    0.0    0.8    0.0    0.0    0.0
0.0    6.3    5.6    0.0    0.0    0.0    0.0    0.0    0.0    0.0

-----------------------------------------------------------------------------------------------
Evaluating Graph Connectivity using BFS.
-----------------------------------------------------------------------------------------------
Graph is not connected. Has 2 components.

-----------------------------------------------------------------------------------------------
Evaluating Minimal Spanning Tree with Prim's Algorithm.
-----------------------------------------------------------------------------------------------

Evaluating sub component with vertex '1'
-------------------------------------------------------------
Edge from 1 to 9 is in the min span tree with cost: 1.5
Edge from 9 to 7 is in the min span tree with cost: 0.8
Edge from 1 to 4 is in the min span tree with cost: 2.3
Edge from 1 to 5 is in the min span tree with cost: 2.4

Evaluating sub component with vertex '2'
-------------------------------------------------------------
Edge from 2 to 3 is in the min span tree with cost: 1.5
Edge from 3 to 6 is in the min span tree with cost: 3.2
Edge from 6 to 8 is in the min span tree with cost: 3.1
Edge from 3 to 10 is in the min span tree with cost: 5.6

Adjacency Matrix for the Minimal Spanning Forest:
-------------------------------------------------------------
0.0    0.0    0.0    2.3    2.4    0.0    0.0    0.0    1.5    0.0
0.0    0.0    1.5    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    1.5    0.0    0.0    0.0    3.2    0.0    0.0    0.0    5.6
2.3    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
2.4    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
0.0    0.0    3.2    0.0    0.0    0.0    0.0    3.1    0.0    0.0
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.8    0.0
0.0    0.0    0.0    0.0    0.0    3.1    0.0    0.0    0.0    0.0
1.5    0.0    0.0    0.0    0.0    0.0    0.8    0.0    0.0    0.0
0.0    0.0    5.6    0.0    0.0    0.0    0.0    0.0    0.0    0.0

-----------------------------------------------------------------------------------------------
Evaluating Shortest Paths from Vertex 1 to the rest using Dijkstra's Algorithm.
-----------------------------------------------------------------------------------------------
```

```
Graph is not connected. Not running Dijkstra's algorithm on it.
```

4. Output for intput 4

```
-----------------------------------------------------------------------------------------------------
Adjacency Matrix for the input graph:
-----------------------------------------------------------------------------------------------------
0.0   3.1   1.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
3.1   0.0   2.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
1.2   2.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   1.2   0.0   0.0   3.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   1.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   1.6   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.8   0.0   1.2   0.0   0.0   0.0   0.0   0.0   9.8
0.0   0.0   0.0   0.0   0.0   0.8   0.0   0.0   0.8   0.0   0.0   3.0   0.0   0.0   1.1
0.0   0.0   0.0   3.0   0.0   0.0   0.0   0.0   0.0   5.1   3.2   0.0   6.1   0.0   0.0
0.0   0.0   0.0   0.0   0.0   1.2   0.8   0.0   0.0   0.0   0.0   2.5   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   5.1   0.0   0.0   1.2   0.0   0.0   2.1   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   3.2   0.0   1.2   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   3.0   0.0   2.5   0.0   0.0   0.0   0.0   0.0   3.1
0.0   0.0   0.0   0.0   1.6   0.0   0.0   6.1   0.0   0.0   0.0   0.0   0.0   3.1   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   2.1   0.0   0.0   3.1   0.0   0.0
0.0   0.0   0.0   0.0   0.0   9.8   1.1   0.0   0.0   0.0   0.0   3.1   0.0   0.0   0.0

-----------------------------------------------------------------------------------------------------
Evaluating Graph Connectivity using BFS.
-----------------------------------------------------------------------------------------------------
Graph is not connected. Has 3 components.

-----------------------------------------------------------------------------------------------------
Evaluating Minimal Spanning Tree with Prim's Algorithm.
-----------------------------------------------------------------------------------------------------

Evaluating sub component with vertex '1'
-------------------------------------------------------
Edge from 1 to 3 is in the min span tree with cost: 1.2
Edge from 3 to 2 is in the min span tree with cost: 2.5

Evaluating sub component with vertex '4'
-------------------------------------------------------
Edge from 4 to 5 is in the min span tree with cost: 1.2
Edge from 5 to 13 is in the min span tree with cost: 1.6
Edge from 4 to 8 is in the min span tree with cost: 3.0
Edge from 13 to 14 is in the min span tree with cost: 3.1
Edge from 14 to 10 is in the min span tree with cost: 2.1
Edge from 10 to 11 is in the min span tree with cost: 1.2

Evaluating sub component with vertex '6'
-------------------------------------------------------
Edge from 6 to 7 is in the min span tree with cost: 0.8
Edge from 7 to 9 is in the min span tree with cost: 0.8
Edge from 7 to 15 is in the min span tree with cost: 1.1
Edge from 9 to 12 is in the min span tree with cost: 2.5
```

```
Adjacency Matrix for the Minimal Spanning Forest:
-------------------------------------------------------
0.0   0.0   1.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   2.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
1.2   2.5   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   1.2   0.0   0.0   3.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   1.2   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   1.6   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.8   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.8   0.0   0.0   0.8   0.0   0.0   0.0   0.0   0.0   1.1
0.0   0.0   0.0   3.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.8   0.0   0.0   0.0   0.0   2.5   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   1.2   0.0   0.0   2.1   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   1.2   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   2.5   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   1.6   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   3.1   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   2.1   0.0   0.0   3.1   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   1.1   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

-------------------------------------------------------------------------------------------------
Evaluating Shortest Paths from Vertex 1 to the rest using Dijkstra's Algorithm.
-------------------------------------------------------------------------------------------------
Graph is not connected. Not running Dijkstra's algorithm on it.
```

**Code Listing:**
**Main.java**

```
package edu.uncc.algos.project2.main;

import edu.uncc.algos.project2.graphstructures.Graph;
import edu.uncc.algos.project2.helpers.mst.PrimsMinimalSpanningTree;
import edu.uncc.algos.project2.helpers.shortestpath.DijkstraShortestPath;


public class Main {

	/**
	 * @param args
	 */
	public static void main(String[] args) {

		//Read the graph and print the adjacency matrix.
		Graph g = Graph.initGraphFromFile("input4.txt");
		System.out.println("-----------------------------------------------------------------------------------------------------------");
		System.out.println("Adjacency Matrix for the input graph: ");
		System.out.println("-----------------------------------------------------------------------------------------------------------");
		g.printAdjMatrix(Graph.ADJ_MAT);

		//check if the graph is connected.
		System.out.println("\n-----------------------------------------------------------------------------------------------------------");
		System.out.println("Evaluating Graph Connectivity using BFS.");
		System.out.println("-----------------------------------------------------------------------------------------------------------");
		g.checkConnectivity();

		//reset vertexColors for use in MST
		g.resetVertexColors();

		//Establish the MST using Prim's Algorithm.
		System.out.println("\n-----------------------------------------------------------------------------------------------------------");
		System.out.println("Evaluating Minimal Spanning Tree with Prim's Algorithm.");
		System.out.println("-----------------------------------------------------------------------------------------------------------");
		PrimsMinimalSpanningTree primSpanningTree = new PrimsMinimalSpanningTree(g);
		primSpanningTree.findMinimalSpanningTree();
		System.out.println("\nAdjacency Matrix for the Minimal Spanning Forest:");
		System.out.println("-----------------------------------------------------------");
		g.printAdjMatrix(Graph.MIN_SPAN_ADJ_MAT);

		//reset the vertex colors for use in the Dijkstra Shortest Path Algorithm
		g.resetVertexColors();

		//Find the shortest path from '1' as the source to all the other vertices.
		System.out.println("\n-----------------------------------------------------------------------------------------------------------");
		System.out.println("Evaluating Shortest Paths from Vertex 1 to the rest using Dijkstra's Algorithm.");
		System.out.println("-----------------------------------------------------------------------------------------------------------");
		if(g.isConnected()){
			DijkstraShortestPath shortestPath = new DijkstraShortestPath(g);
```

```
                        shortestPath.findShortestPaths();
                        System.out.println("\nAdjacency matrix for Shortest paths.");
                        System.out.println("-----------------------------------------------------");
                        g.printAdjMatrix(Graph.SHORTEST_PATH_MAT);
                } else {
                        System.out.println("Graph is not connected. Not running Dijkstra's algorithm on it.");
                }
        }
}
```

**Graph.java**

```
package edu.uncc.algos.project2.graphstructures;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

import edu.uncc.algos.project2.constants.VertexColor;

public class Graph {

        private final Integer VERTICES;
        private final Integer EDGES;

        private List<Vertex> vertices;
        private List<Edge> edges;

        private boolean isConnected;

        private List<Integer> componentStartVertex;

        private float adjMatrix[][];
        private float adjMatrixForMinimalSpanningTree[][];
        private float adjMatrixForShortestPaths[][];

        public static final int ADJ_MAT = 1;
        public static final int MIN_SPAN_ADJ_MAT = 2;
        public static final int SHORTEST_PATH_MAT = 3;

        public Graph(Integer v, Integer e){
                VERTICES = v;
                EDGES = e;

                adjMatrix = new float[VERTICES+1][VERTICES+1];
                adjMatrixForMinimalSpanningTree = new float[VERTICES+1][VERTICES+1];
                adjMatrixForShortestPaths = new float[VERTICES+1][VERTICES+1];
```

```
		vertices = new ArrayList<Vertex>(VERTICES+1);
		edges = new ArrayList<Edge>(EDGES+1);

		for(int i = 1 ; i < VERTICES + 1 ; i ++)
			vertices.add(new Vertex(i));

		componentStartVertex = new ArrayList<Integer>();
		setConnected(true);
	}

	public void setMinimalSpanningTreeAdjMatrixValue(Integer sourceVertex, Integer destinationVertex, float cost){
		adjMatrixForMinimalSpanningTree[sourceVertex][destinationVertex] =
				adjMatrixForMinimalSpanningTree[destinationVertex][sourceVertex] = cost;
	}

	public void setShortestPathAdjMatrixValue(Integer destVertex, Integer throughVertex, float cost){
		adjMatrixForShortestPaths[destVertex][throughVertex] =
				adjMatrixForShortestPaths[throughVertex][destVertex] = cost;

	}

	public List<Integer> getComponentStartVertices(){
		return componentStartVertex;
	}

	public void addEdge(Integer startVertex, Integer endVertex, float weight){
		adjMatrix[startVertex][endVertex] = adjMatrix[endVertex][startVertex] = weight;

		edges.add(new Edge(startVertex, endVertex, weight));
	}

	public void printAdjMatrix(int matrixType){

		if(matrixType == Graph.MIN_SPAN_ADJ_MAT){
			for(int i = 1; i < VERTICES + 1 ; i ++){
				for(int j = 1; j < VERTICES + 1; j ++)
					System.out.print(adjMatrixForMinimalSpanningTree[i][j]+"\t" );
				System.out.println();
			}
		} else if(matrixType == Graph.ADJ_MAT){
			for(int i = 1; i < VERTICES + 1 ; i ++){
				for(int j = 1; j < VERTICES + 1; j ++)
					System.out.print(adjMatrix[i][j]+"\t");
				System.out.println();
			}
		} else {
			for(int i = 1; i < VERTICES + 1 ; i ++){
				for(int j = 1; j < VERTICES + 1; j ++)
					System.out.print(adjMatrixForShortestPaths[i][j]+"\t" );
				System.out.println();
			}
		}
	}
```

```
	public static Graph initGraphFromFile(String inputFile){

		Graph g;

		FileInputStream fstream = null;
		DataInputStream in = null;
		BufferedReader br = null;

		try{
			fstream = new FileInputStream(inputFile);
			in = new DataInputStream(fstream);
			br = new BufferedReader(new InputStreamReader(in));
			String strLine;

			g = new Graph(Integer.parseInt(br.readLine().trim()), Integer.parseInt(br.readLine().trim()));

			while ((strLine = br.readLine()) != null)   {
				String [] str = strLine.split(",");
				g.addEdge(Integer.parseInt(str[0].trim()), Integer.parseInt(str[1].trim()),
Float.parseFloat(str[2].trim()));
			}

			return g;

		}catch (Exception e){
			e.printStackTrace();
		} finally {
			try {
				in.close();
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
		return null;
	}

	private void setVertexColor(Integer vertexId, int color){
		for(Vertex v : vertices)
			if(v.getId() == vertexId)
				v.setColor(color);
	}

	private int getVertexColor(Integer vertexId){
		for(Vertex v : vertices)
			if(v.getId() == vertexId)
				return v.getColor();

		return -1;
	}

	private void doBFS(Vertex v){

		Queue<Integer> queue = new LinkedList<Integer>();
		queue.add(v.getId());
```

```
		while(!queue.isEmpty()){
			Integer vertexId = queue.remove();
			for(Integer i : getAdjecentVertices(vertexId)){
				if(getVertexColor(i) == VertexColor.WHITE)
					queue.add(i);
			setVertexColor(vertexId, VertexColor.BLACK);
			}
		}
	}

	public List<Integer> getAdjecentVertices(int vertexId){
		List<Integer> adjVertices = new ArrayList<Integer>();

		for(int i = 1 ; i < VERTICES + 1 ; i ++)
			if(adjMatrix[vertexId][i] != 0.0)
				adjVertices.add(i);

		return adjVertices;
	}

	public void checkConnectivity() {

		for(Vertex v : vertices){
			if(v.getColor() == VertexColor.WHITE){
				componentStartVertex.add(v.getId());
				doBFS(v);
			}
		}

		if(componentStartVertex.size() > 1){
			System.out.println("Graph is not connected. Has " + componentStartVertex.size() + " components.");
			isConnected = false;
		} else {
			System.out.println("Graph is connected.");
			isConnected = true;
		}

	}

	public void resetVertexColors(){
		for(Vertex v : vertices)
			v.setColor(VertexColor.WHITE);
	}

	public boolean isConnected() {
		return isConnected;
	}

	public void setConnected(boolean isConnected){
		this.isConnected = isConnected;
	}

	public int getNoOfVertices(){
```

```
		return VERTICES;
	}

	public float getEdgeWeight(int sourceVertex, int destVertex) {
		return adjMatrix[sourceVertex][destVertex];
	}

	public void setVertexProcessed(int vertex) {
		vertices.get(vertex - 1).setColor(VertexColor.BLACK);
	}

	public boolean isVertexProcessed(Integer adjacentVertex) {
		if(vertices.get(adjacentVertex - 1).getColor() == VertexColor.BLACK)
			return true;

		return false;
	}
}
```

**PriorityQueue.java**

```
package edu.uncc.algos.project2.helpers;

import java.util.Comparator;

public class PriorityQueue {

	private java.util.PriorityQueue<PriorityQueueElement> pq;

	public PriorityQueue(int noOfVertices){
		pq = new java.util.PriorityQueue<PriorityQueueElement>(noOfVertices, new PrioQueueElementComparator());

	}

	public void insert(int vertex, int parent, float cost){
		pq.add(new PriorityQueueElement(vertex, parent, cost));
	}

	public PriorityQueueElement delete(){
		return pq.poll();
	}

	public void update(int vertex, int parent, float cost){
		PriorityQueueElement elementToUpdate = null;
		for(PriorityQueueElement pqe : pq){
			if(pqe.getVertex() == vertex){
				elementToUpdate = pqe;
				break;
			}
		}

		pq.remove(elementToUpdate);
		pq.add(new PriorityQueueElement(vertex, parent, cost));
	}
```

```
	public boolean checkAndUpdate(int vertex, int parent, float cost){
		boolean returnValue = false;
		PriorityQueueElement elementToUpdate = null;

		for(PriorityQueueElement pqe : pq){
			if(pqe.getVertex() == vertex){
				returnValue = true;
				if(pqe.getCost() > cost){
				elementToUpdate = pqe;
				break;
				}
			}
		}

		if(elementToUpdate != null){
			pq.remove(elementToUpdate);
			pq.add(new PriorityQueueElement(vertex, parent, cost));
		}

		return returnValue;
	}

	public boolean isEmpty(){
		return pq.isEmpty();
	}

	public float contains(int vertex){
		for(PriorityQueueElement pqe : pq)
			if(pqe.getVertex() == vertex)
				return pqe.getCost();
			return -1.0f;
	}

	public PriorityQueueElement getElementWithVertex(int vertex){
		for(PriorityQueueElement pqe : pq)
			if(pqe.getVertex() == vertex)
				return pqe;
		return null;
	}
}

class PrioQueueElementComparator implements Comparator<PriorityQueueElement>{
	@Override
	public int compare(PriorityQueueElement arg0, PriorityQueueElement arg1) {
		if(arg0.getCost() < arg1.getCost())
			return -1;
		else if(arg0.getCost() > arg1.getCost())
			return 1;
		return 0;
	}

}
```

PriorityQueueElement.java

```
package edu.uncc.algos.project2.helpers;

public class PriorityQueueElement {
		private int vertex;
		private int parent;
		private float cost;

		public PriorityQueueElement(int v, int p, float c){
			vertex = v;
			parent = p;
			cost = c;
		}

		public void setParent(int p){
			parent = p;
		}

		public void setCost(float c){
			cost = c;
		}

		public int getVertex(){
			return vertex;
		}

		public int getParent(){
			return parent;
		}

		public float getCost(){
			return cost;
		}
	}
```

**DijkstraShortestPath.java**

```
package edu.uncc.algos.project2.helpers.shortestpath;

import edu.uncc.algos.project2.graphstructures.Graph;
import edu.uncc.algos.project2.helpers.PriorityQueue;
import edu.uncc.algos.project2.helpers.PriorityQueueElement;

public class DijkstraShortestPath {

	private PriorityQueue pq;
	private Graph g;

	public DijkstraShortestPath(Graph g){
		this.g = g;
```

```
		pq = new PriorityQueue(g.getNoOfVertices());
	}

	public void findShortestPaths(){

		pq.insert(1, -1, 0.0f);

		while(!pq.isEmpty()){
			PriorityQueueElement vertex = pq.delete();

			for(Integer adjacentVertex : g.getAdjecentVertices(vertex.getVertex())){
				if(!pq.checkAndUpdate(adjacentVertex, vertex.getVertex(), vertex.getCost() +
g.getEdgeWeight(vertex.getVertex(), adjacentVertex))) {
					if(!g.isVertexProcessed(adjacentVertex))
						pq.insert(adjacentVertex, vertex.getVertex(), vertex.getCost() +
g.getEdgeWeight(vertex.getVertex(), adjacentVertex));
				}
			}

			if(vertex.getParent() != -1){
				g.setShortestPathAdjMatrixValue(vertex.getVertex(), vertex.getParent(), vertex.getCost());
				System.out.println("Path from 1 to " + vertex.getVertex()  + " is via " + vertex.getParent()+ " with
cost: " + vertex.getCost());
			}
			g.setVertexProcessed(vertex.getVertex());

		}
	}
}
```

**PrimsMinimalSpanningTree.java**

```
package edu.uncc.algos.project2.helpers.mst;

import edu.uncc.algos.project2.graphstructures.Graph;
import edu.uncc.algos.project2.helpers.PriorityQueue;
import edu.uncc.algos.project2.helpers.PriorityQueueElement;

public class PrimsMinimalSpanningTree {

	private Graph g;
	private PriorityQueue pq;

	public PrimsMinimalSpanningTree(Graph g){
		this.g = g;
		pq = new PriorityQueue(g.getNoOfVertices());
	}

	public void findMinimalSpanningTree(){

		for(Integer i : g.getComponentStartVertices()){
			System.out.println("\nEvaluating sub component with vertex '" + i + "'");
			System.out.println("-------------------------------------------------------------");
			pq.insert(i, -1, 0.0f);
```

```
                        while(!pq.isEmpty()){
                                PriorityQueueElement vertex = pq.delete();

                                for(Integer adjacentVertex : g.getAdjecentVertices(vertex.getVertex())){
                                        if(!pq.checkAndUpdate(adjacentVertex, vertex.getVertex(),
g.getEdgeWeight(vertex.getVertex(), adjacentVertex))) {
                                                if(!g.isVertexProcessed(adjacentVertex))
                                                        pq.insert(adjacentVertex, vertex.getVertex(),
g.getEdgeWeight(vertex.getVertex(), adjacentVertex));
                                        }
                                }

                                if(vertex.getParent() != -1){
                                        g.setMinimalSpanningTreeAdjMatrixValue(vertex.getParent(), vertex.getVertex(),
vertex.getCost());
                                        System.out.println("Edge from " + vertex.getParent() + " to " + vertex.getVertex() + " is
in the min span tree with cost: " + vertex.getCost());
                                }
                                g.setVertexProcessed(vertex.getVertex());
                        }
                }
        }
}
```

**VertexColor.java**

```
/**
 *
 */
package edu.uncc.algos.project2.constants;

/**
 * @author Ankur
 *
 */
public interface VertexColor {

        int WHITE = 1;
        int GRAY  = 2;
        int BLACK = 3;

}
```