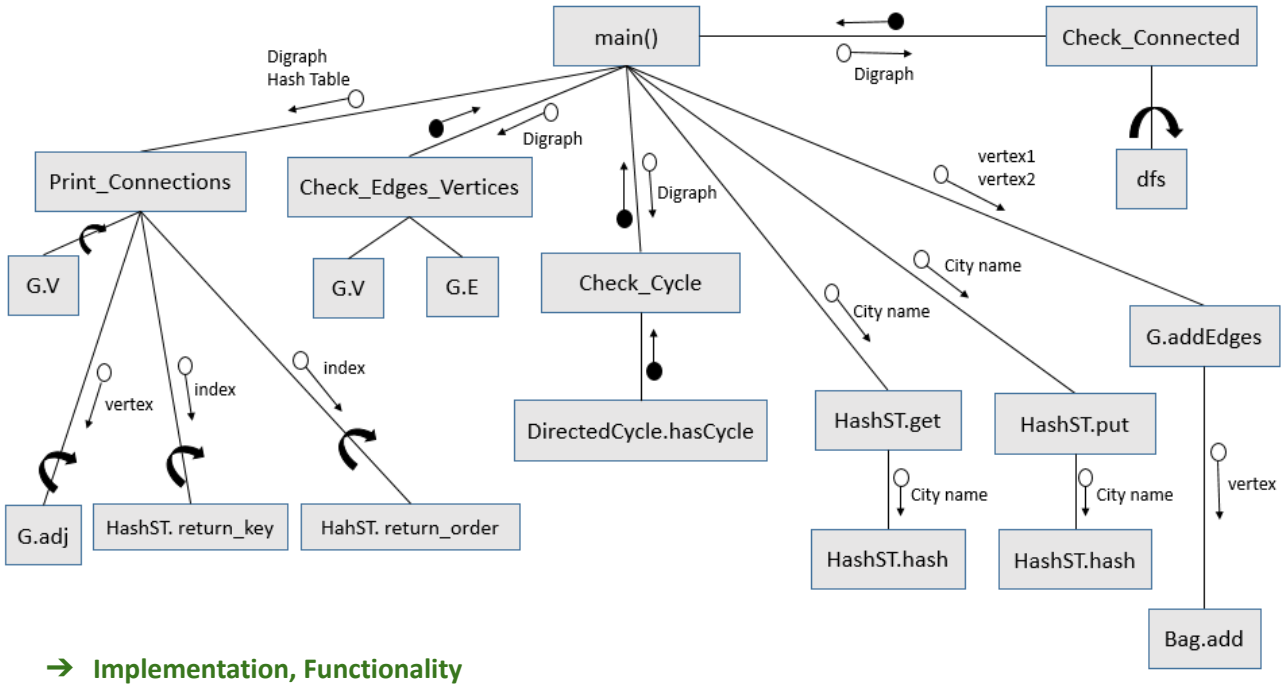
Report

Question 1

➔ Problem Statement and Code Design



➔ Implementation, Functionality

```
boolean Check_Edges_Vertices (Digraph G) {
   if (number of edges in G + 1 == number of vertices in G )
      {
        return true
      }
   else
        {
            return false
        }
}
```

```
boolean Check_Cycle (Digraph G) {

   initialize a DirectedCycle object
   call its hasCycle() method

   if (Digraph G hasCycle)
      {
            return false
      }
   else
        {
            return true
        }
}
```

```
boolean Check_Connected (Digraph G) {

 initialize a boolean array visited

 for ( int vertex = 0 ; vertex < number of vertices ; vertex++)
      {
        Depth_First_Search (G , vertex , visited);
      }

 for ( int vertex = 0 ; vertex < number of vertices ; vertex++)
      {
        if (vertex is not visited)
        {
           return false;
        }
      }
      return true;
}
```

Check_Edges_Vertices: this method returns "true" if the number of edges is one less than the number of vertices and returns "false" otherwise.

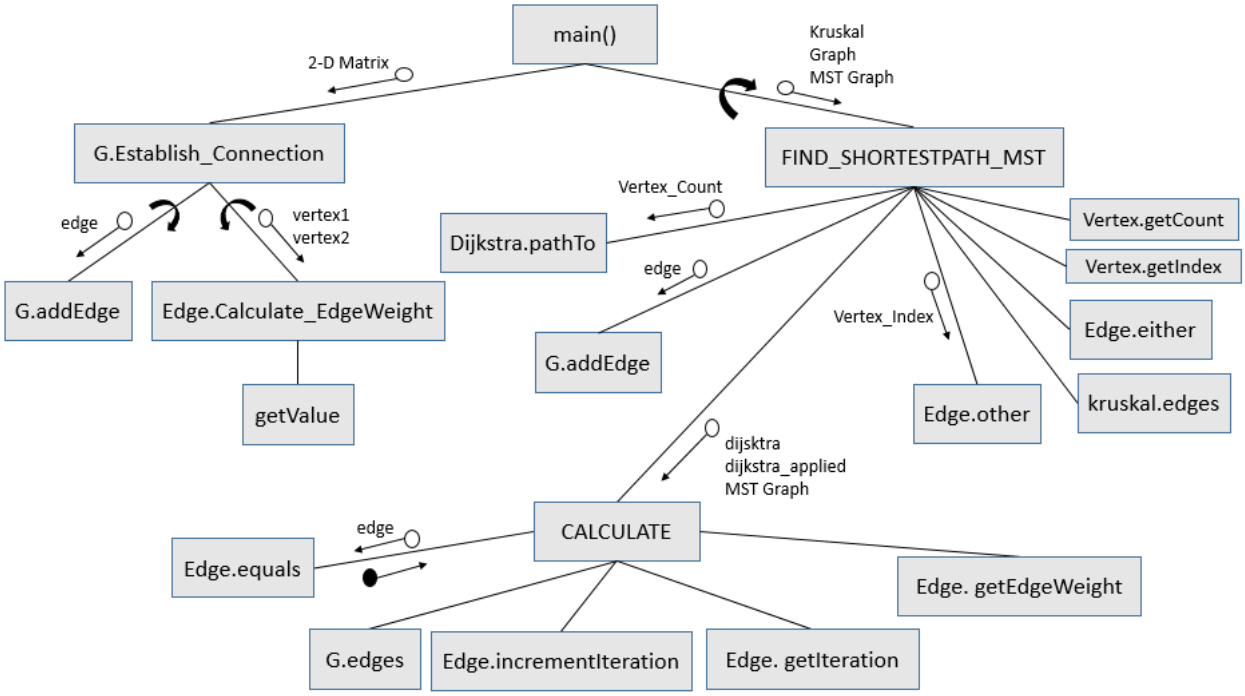
Check_Cycle: this method returns "true" if the graph is acyclic and returns "false" otherwise. This method calls the "hasCycle" method in the "DirectedCycle" class to check this condition.
Check_Connected: this method returns "true" if the graph is connected and returns "false" otherwise. The method checks the condition by performing a DFS starting from each vertex and then checking if all vertices were visited.
PrintConnections: It prints out the cities and their connections based on the order they were inserted into the hash table and according to the edges created between them in the Digraph.

➔ Testing

For testing this class I tried different graph inputs with different sizes (different number of edges and vertices) to validate that the methods work for different input sizes. I also tried graphs with different conditions. Specifically, I tried various graphs in which each one of them has only one of the conditions not satisfied to validate and I also tried one that has all three conditions satisfied in order to validate that each condition/method functions the right way.

Question 2

➔ Problem Statement and Code Design



➔ Implementation, Functionality

FIND_SHORTESTPATH_MST: this method calls the required methods like the Kruskal MST and the Dijkstra's Shortest Path methods and does the required operations for finding the needed path. It calls the **"CALCULATE"** method that calculates total path cost and prints it to the console.
CALCULATE: this method calculates the total cost of the trip. To do so: for each edge in the MST Graph, we add its cost to the total cost of the trip. But, before adding it we check if the edge was visited twice,

if it is so, we increment the number of iterations over the edge in order to to use it in the calculation. The method uses a special formula to find the total trip cost.

Establish_Connection: the method iterates through the grid or the 2- DMatrix row by row and then column by column. At each iteration we create an edge between the vertices in the matrix accordingly.

```
FIND_SHORTESTPATH_MST ((KruskalMST kruskal , Graph MST_Graph, Graph G) {

 initialize an array of Vertices

 for ( each edge in the kruskal MST)
     {
       add edge to the MST_Graph
       add each vertex in the edge to the array
     }

  set redundant vertices in the array to null
  add remaining vertices in the array to another array "vertices"

 if (one of the vertices in the "vertices" array is not a source/destination vertex)
   {
     apply Dijkstra's shortest path (from this vertex, to source/destination vertex )
   }

 Print (CALCULATE) // returns total trip cost
}
```

```
CALCUATE (Graph MST_Graph, DijkstraSP dijkstra, boolean dijkstra_applied {

 Trip_total_cost = 0

 for (each edge in the MST_Graph)
     {
       if (dijkstra_applied)
        {
          if edge if visited twice, increment the visited count of the edge by 1
        }
     }
 Trip_total_cost = Trip_total_cost + weight of edge + ceiling of visited count
                   of the edge

 return Trip_total_cost
}
```

➔ Testing

For testing this class I tried the code with different Matrices that have different numbers of columns and rows denoting different numbers of edges and vertices, which means that I tried bigger and smaller input matrices to check that the code is valid for different input sizes. I also tried to change the source and destination vertices in each matrix to observe whether the path chosen was always the most optimal path and to validate that the calculation of the path cost is done correctly and efficiently.

➔ Final Assessments (For All Questions)

The trouble points in completing this assignment were mostly trying to understand the question. For example, I found trouble trying to figure out how to print the cities in the specified order in Question 1 and I could hardly understand the objectives that we were asked to achieve in Question 2. The most challenging part was writing the algorithms that find the optimal path from the source to the destination vertex in Question 2 . However, what I liked about this assignment is that it provided me with hands-on-practice about the new topic of MST graphs. I learned from it the algorithms of finding the minimum spanning tree (Question 2) and using the hash structure as a symbol table as well as checking the conditions of a tree structure on a graph like cycles and connections (Question 1). I also learned the algorithmic thinking behind those methods which are a very important topic in Data Structures and Algorithms.