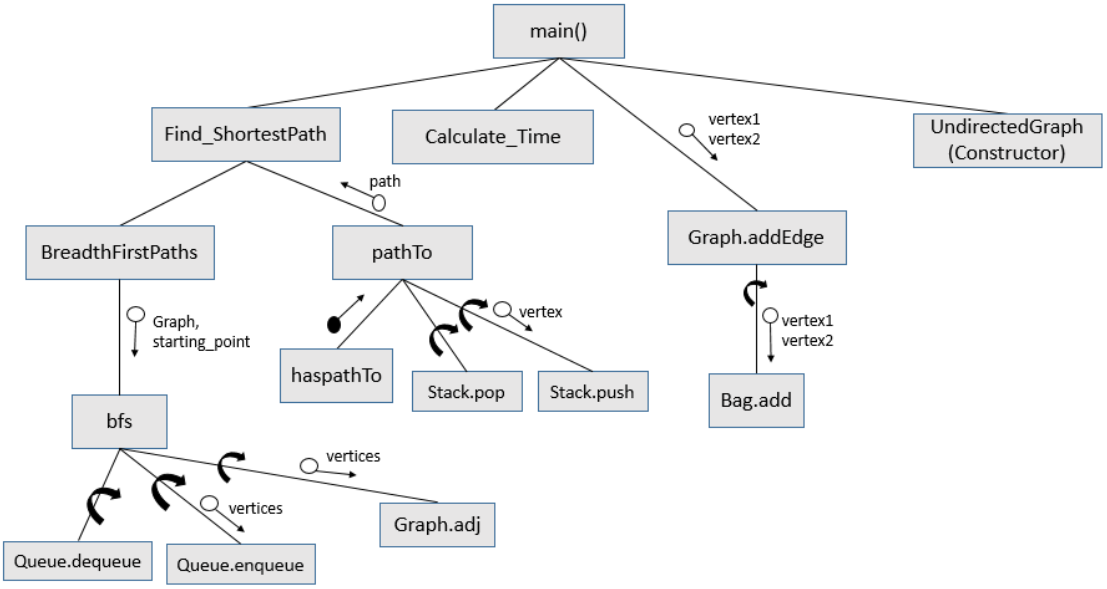
Report

Basme Zantout

Question 1

➔ Problem Statement and Code Design



➔ Implementation, Functionality

```
PathTo (int destinationVertex)
{
   if (there is no path to destinationVertex)
         { return null }

   else
         {
            initialize a Stack
            for ( int x = destinationVertex and for every
                  vertex in edge to x and while x not equal
                  sourceVertex)
               {
                     push  x to stack
               }

            push sourceVertex to stack
            print out number of vertices in stack

            return the stack
}
```

```
bsf (int sourceVertex)
{
   Initialize a Queue
   Mark sourceVertex as visited
   Enqueue the sourceVertex

    While (queue is not empty)
       {
          int v = dequeue first element in queue
           for each (vertex adjacent to v)
                {
                   mark it as visited
                   store the vertex in edge to it
                    enqueue it to queue
                }
        }
}
```

```
Calculate_Time()
{
   int z = 0

   if ( Route_Time  <  (ChangeState*2) )
     {
        z = (ChangeState*2) - Route_Time
     }

   time = (Route_Time * (Number_of_Visited_Cities -  1))
            + (z * (Number_of_Visited_Cities - 2))

    print out time
}
```

```
Find_ShortestPath()
{
   Initialize a  BreadthFirstPaths  object with
   sourceVertex and given Graph

   for each (vertex x in the shortest pathTo the
              from sourceVertex to destinationVertex)
      {
         print out x
      }
}
```

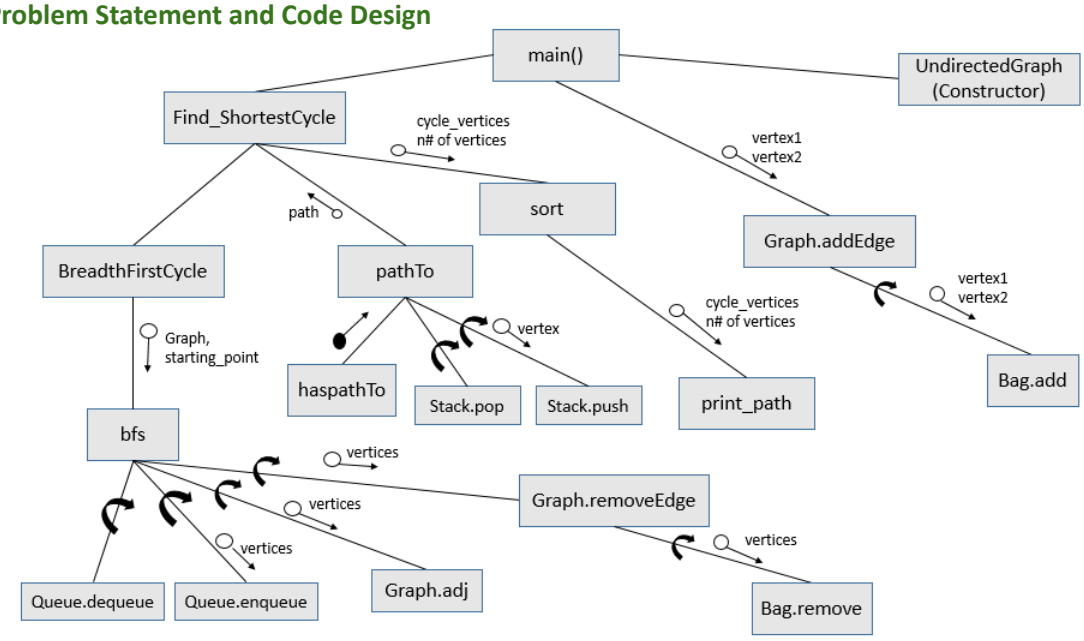
bsf: this method uses the Breadth First Search algorithm to traverse over each vertex in a graph (finds a path in a graph) starting from a source vertex *(Note: it is the same one used in Question 2)*.
pathTo: finds the shortest path from the source vertex to a destination vertex given as parameter.
SortestPath: finds the shortest path from the starting to the destination airport. It calls the "pathTo" method in the "BreadthFirstPaths" class which finds the shortest path from the starting (source) airport to the given destination airport and prints out the vertices in the path.
CalculateTime: The method does the calculation as follows: if the time for traveling from one city to another is greater than 2 times the time required by airports to change their states then the final time taken for the whole route is the "Route_Time" multiplied by the number of routes or edges. Otherwise, if the time for traveling from one city to another is less than 2 times the time required by airports to change their states, then the final time is the calculated the same as before but this time we add the difference between the "Route_Time" and "ChangeState" at each city/vertex visited.

➔ Testing

For testing this class I tried different graph inputs with different sizes (different number of edges and vertices) to validate that the methods I wrote like the *BreathFirstPaths()* and *pathTo()* methods correctly find the shortest path from a vertex to another. I also tried to alter the Route and loading/running times to check that the *Calculate_Time()* method correctly calculates the flight time in different graphs and different airline routes.

Question 2

➔ Problem Statement and Code Design



➔ Implementation, Functionality

ShortestCycle: finds the shortest path from the starting vertex to the included vertex and then finds *another* shortest path from the included vertex back to the starting vertex. It also stores the vertices in the path in an array and calls a method that sorts the vertices in ascending order.

PathTo: it first checks if there is a possible path to the included vertex. It returns the source and included vertices if no such path exists *(Note: we didn't return null because there might be only one single path between source and included vertex)*. Otherwise, if there exists a path, then the method finds the shortest path and deletes the vertices included in the path to avoid repeating the path twice in a cycle.

```
Find_ShortestCycle()
{       pathVertices [0] = sourceVertex
        pathVertices [1] = IncludedVertex
        pathVertices_count = 2

        Initialize a  BreadthFirstCyle object for GOING path
        with sourceVertex and given Graph

        for each (vertex x in the shortest pathTo the
                  from sourceVertex to IncludedVertex)
            {
              if (x not equal pathVertices [0]  && pathVertices [1])
                 {
                   add x to pathVertices [count]
                    count++
                 }
        Initialize a  BreadthFirstCyle object for RETURNING path
        with sourceVertex and given Graph

        for each (vertex x in the shortest pathTo the
                  from IncludedVertex to sourceVertex)
            {
              if (x not equal pathVertices [0]  && pathVertices [1])
                 {
                   add x to pathVertices [count]
                    count++
                 }
        sort pathVertices
}
```

```
PathTo (int IncludedVertex)
{
   Initialize a Stack

   if (there is no path to destinationVertex)
    {
     push sourceVertex & IncludedVertex to Stack
     return the Stack
    }
 else
   {
    for ( int x = IncludedVertex and for every vertex in
          edge to x and while x not equal sourceVertex)
      {
            push  x to stack
            if (x not equal IncludedVertex)
           { removeEdge between x and edgeTo[x]}
      }

     push sourceVertex to stack
     if (number of vertices in stack == 2)
     { removeEdge between sourceVertex and IncludedVertex }

            return the stack
   } }
```

➔ Testing

For testing this class I tried the code with different numbers of edges and vertices to validate that the methods I wrote like the *BreathFirstCycle()* and *pathTo()* methods correctly find the shortest cycle which includes a specific vertex without repeating the same path twice. I also considered the case when only one path is available in the tour guide. Additionally, I tried bigger and smaller input graphs to check that the sorting and other methods are valid for different input sizes.

➔ 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 calculate the airline time based on the route taken and it was hard to figure out that vertices in Question 2 were written in ascending order and that they had to be a cycle with no repeated vertices. On the other hand, the most challenging part was writing the algorithms that find the shortest path/cycle like the BreadthFirstSearch algorithm. However, what I liked about this assignment is that it provided me with hands-on-practice about the new topic of undirected graphs. I learned from it the algorithms of finding the shortest path/cycle in a graph and I learned the algorithmic thinking behind those methods which are a very important topic in Data Structures and Algorithms.