

# 6385.0W1 Algorithmic Aspects of Telecommunication Networks

## Project 3

Name: Adarsh Raghupati

NetID: axh190002

## Contents

Problem statement: .....	2
Algorithm Description: .....	2
Algorithm 1: Simple Heuristics.....	2
Deletion:.....	2
Algorithm 2: Simple Construction Heuristic: .....	3
Graphical representation and analysis of the result:.....	4
Deletion heuristic algorithm: .....	4
Construction heuristic algorithm: .....	4
Experimental run examples: .....	5
Deletion heuristic algorithm: .....	5
Construction heuristic algorithm: .....	6
ReadMe: .....	7
Technologies/Tools used: .....	7
How to run the program: .....	7
Appendix: .....	8
Source Code: .....	8
<b>Reference</b> .....	14

## Problem statement:

1. Given the location of  $n$  nodes in the plane by their coordinates. Create a network topology (represented by an undirected graph), such that it has the following properties:
2. It contains all the given nodes.
3. The degree of each vertex in the graph is at least 3, that is, each node is connected to at least 3 other nodes.
4. The diameter of the graph is at most 4. By this we mean that any node can be reached from any other node in at most 4 hops. That is, the diameter in our case refers to the hop-distance, not to any kind of geometric distance. Note that this diameter bound implies that the graph must be connected.
5. The total cost of the network topology is as low as possible, where the cost is measured by the total geometric length of all links. This means, you have compute how long each link is geometrically (that is, how far apart are its end-nodes), and then sum it up for all links that exist in the network. This sum represents the total cost that we would like to minimize, under the constraints described above in items 1,2,3.

## Algorithm Description:

### Algorithm 1: Simple Heuristics

#### Deletion:

The algorithm starts with all the edges in the graph and delete the worst edge at each iteration if its deletion results in a network that contains a feasible solution. The procedure stops when no edges in the solution can be deleted. Typically, for minimization problems, the edges are selected for deletion by decreasing order of costs.

#### Steps:

- Generate  $n$  nodes with random  $x,y$  coordinates
- Make the graph as complete graph by adding edges between each pair of vertices
- Compute the distance of each edge
- Sort the edges according to their distance
- Delete the edges in the decreasing order of distance
- If the deletion of the edge violates the degree constraint or the diameter constraint, then do not remove the edge from the graph
- Continue the deletion process till all edges are processed
- Calculate the cost by summing the distance of remaining edges
- Repeat the above steps for different values of  $n$

### Algorithm 2: Simple Construction Heuristic:

The most popular approach is to attempt to find a solution by starting with the problem graph, without any edges, and to construct the solution by adding edges one at a time. Usually this is done by selecting the best edge, according to a specified rule, and adding it to the solution only if it does not violate feasibility.

Steps:

- Generate n nodes with random x,y coordinates
- Create n vertices without any edges
- Create all possible edges and compute their distances
- Sort the edges according to their distance
- Add the edges to the vertices in increasing order of their distance
- Stop adding the edges If the network satisfies the condition of vertex degree and diameter constraint
- Calculate the cost by summing the distance of considered edges
- Repeat the above steps for different values of n

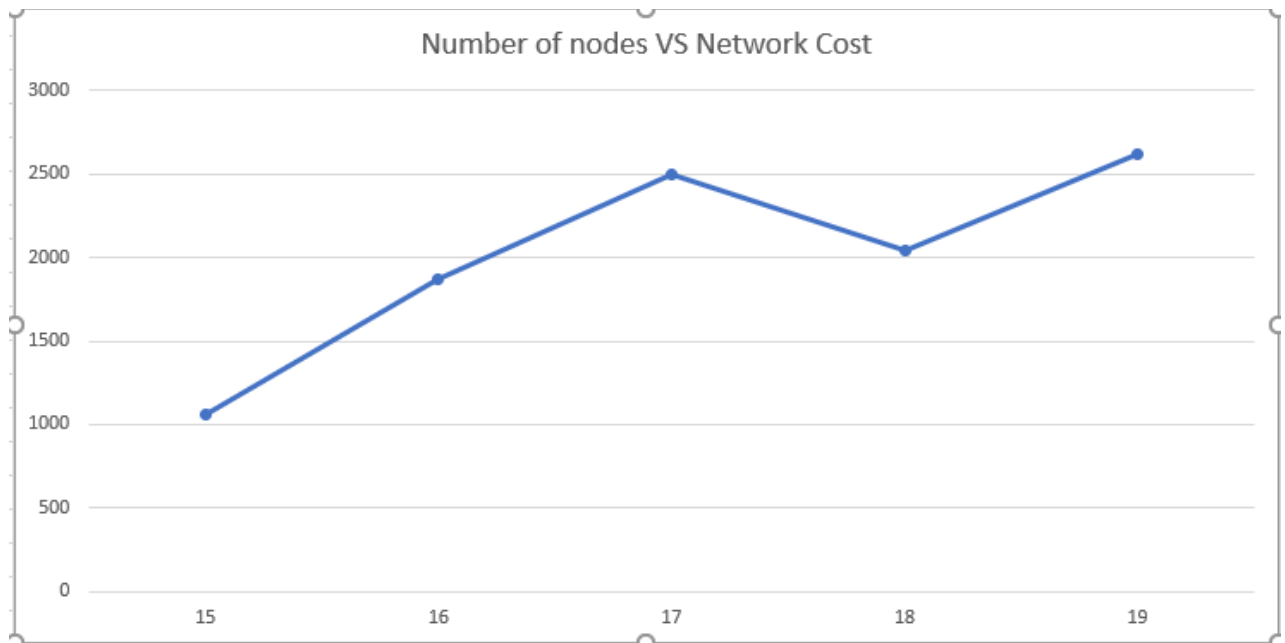
To check the diameter constraint for the graph, breadth first search algorithm with hop count is used.

Breadth First Search:

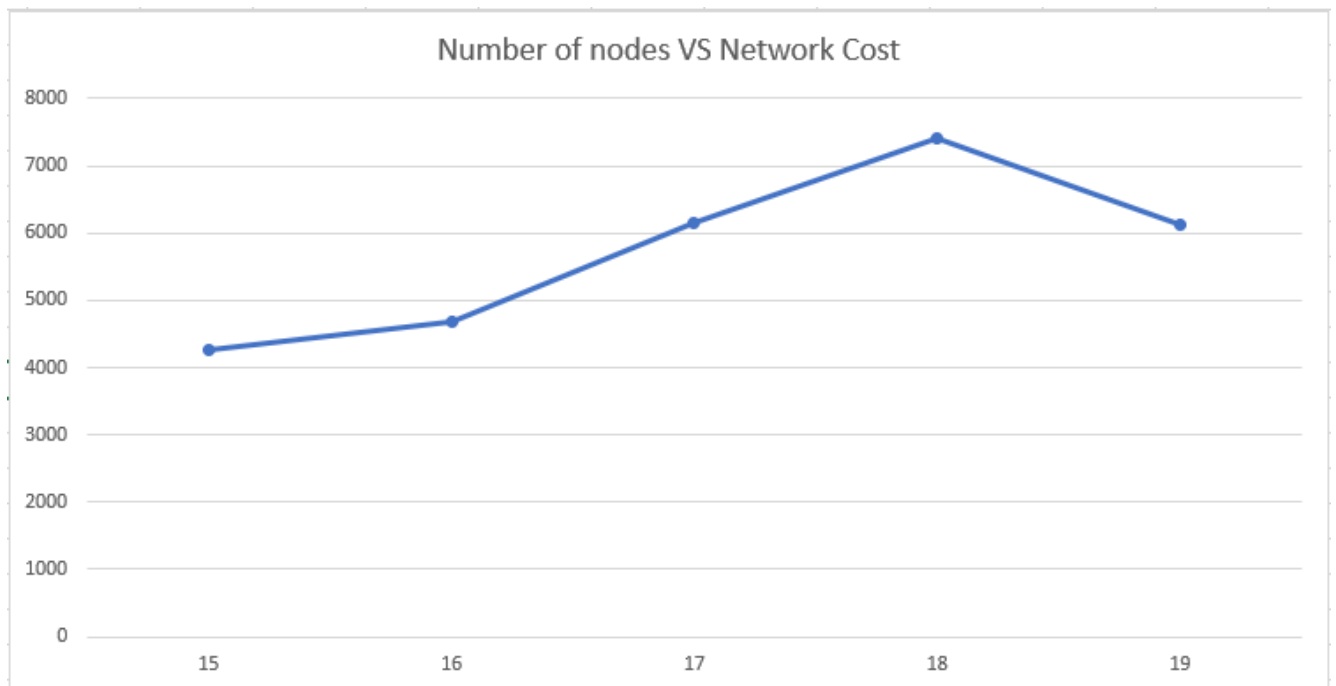
```
Add vertex 1 to queue
Visited.add(vertex1)
hopCount=1
while(queue is not empty and hopCount<4){
  Vertex u = queue.poll()
  For neighboring vertex of u{
    If( neighbor is not visited)
      Add to queue
  }
  hopCount++
}
```

Graphical representation and analysis of the result:

Deletion heuristic algorithm:



Construction heuristic algorithm:



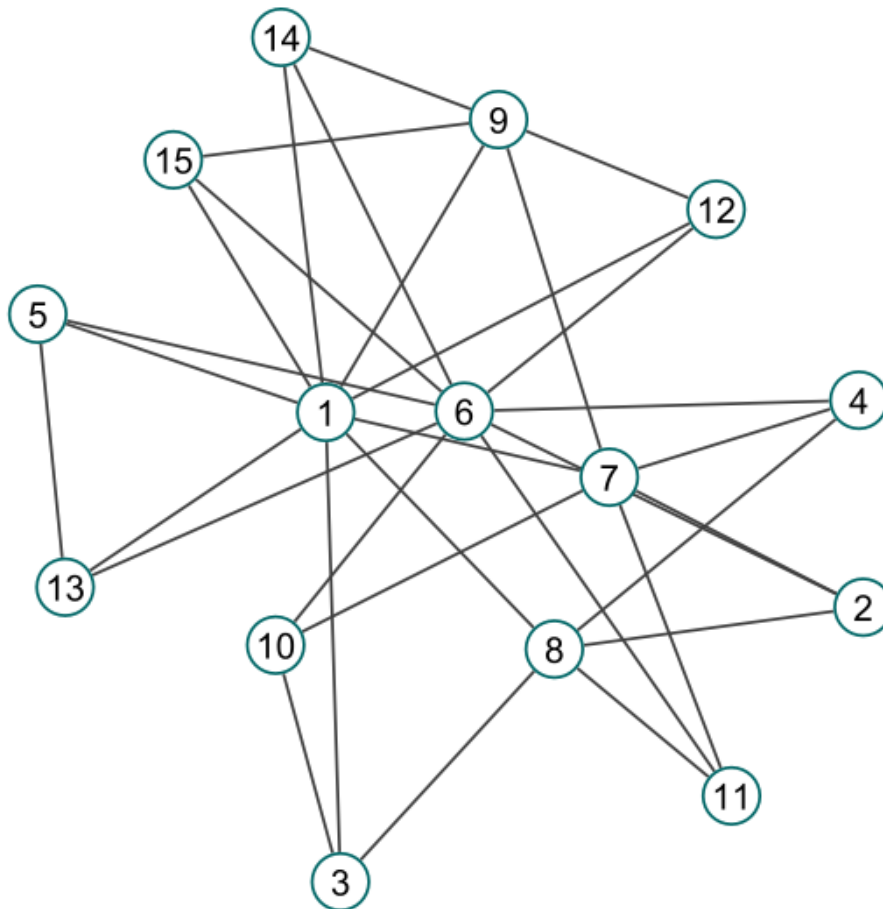
### Algorithm comparison:

- From the above figure, we can see that both the algorithms have increasing trend in the cost as the number of nodes in the network grows.
- Cost of the network in the case of deletion heuristic is significantly better compared to construction heuristic.
- In the construction algorithm, at each step edge with lowest distance is added to the network. If the subset of vertices in the graph are closer to each other, the algorithm will add the unnecessary low-cost edges to the network.
- Deletion heuristic algorithm process all edges before stopping whereas the construction algorithm stops when the given degree and diameter constraint is satisfied. Hence the run time for construction algorithm is better than deletion algorithm.
- Both the algorithms do not guarantee optimal solution

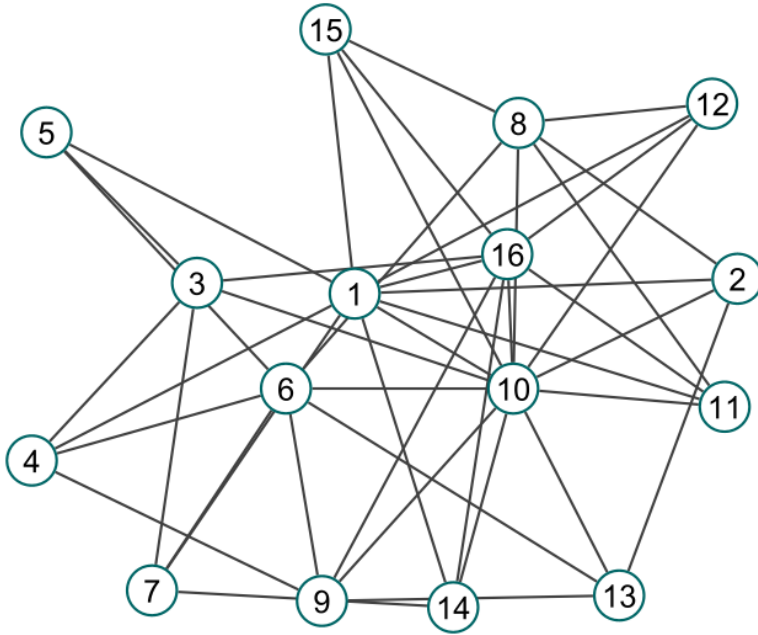
### Experimental run examples:

#### Deletion heuristic algorithm:

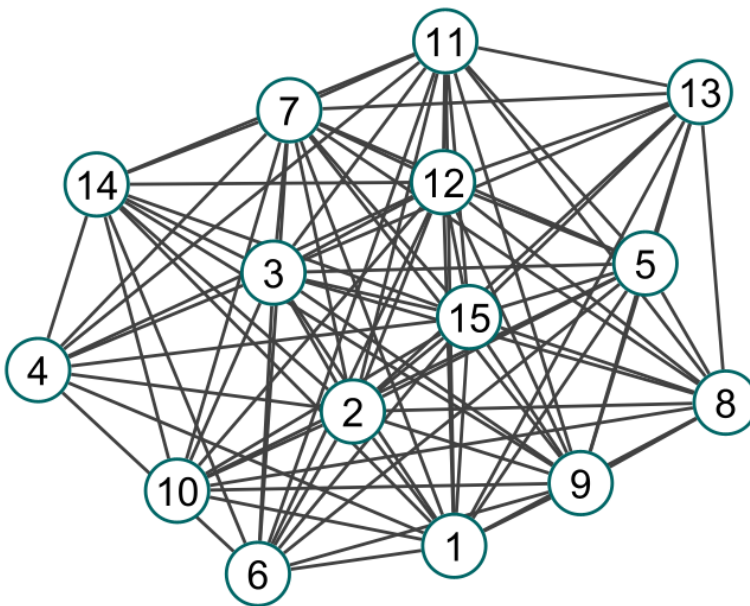
Number of nodes=15



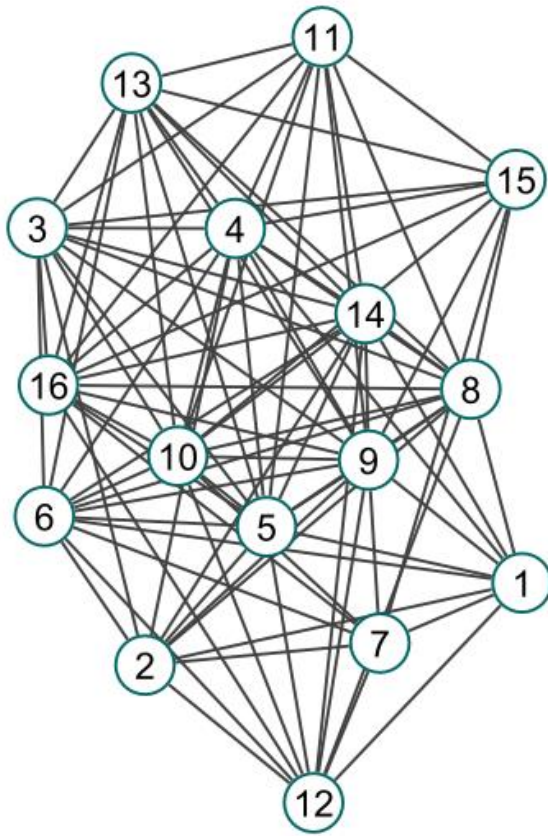
Number of nodes = 16



Construction heuristic algorithm:  
Number of nodes=15



Number of nodes=16



ReadMe:

Technologies/Tools used:

Programming language: Java

IDE: IntelliJ

Visualization tool: MS Excel, Cytoscape

How to run the program:

- Create an empty java project in IntelliJ
- Create a package called project3
- Create the java files Graph.java, Edge.java and Vertex.java
- Copy the code from the appendix section to respective java files
- Run the Graph.java
- Output is generated at console csv files are generated
- Open the output csv files and visualize using the tool cytoscape

## Appendix:

### Source Code:

Graph.java

```
package project3;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

public class Graph {
    HashMap<Integer,Vertex> vertexMap;
    List<Edge> edgeList;
    int n;

    Graph(int n){
        this.n = n;
        Random random = new Random();
        vertexMap = new HashMap<>();
        edgeList = new ArrayList<>();
        for(int i=1;i<=n;i++){
            int x = random.nextInt(100);
            int y = random.nextInt(100);
            vertexMap.put(i,new Vertex(i,x,y));
        }
    }

    public void createCompleteGraph(){
        int edgeCount=1;
        for(int i=1;i<=n;i++){
            for(int j=i+1;j<=n;j++){
                Edge e1 = new Edge(vertexMap.get(i),vertexMap.get(j),edgeCount);
                vertexMap.get(i).getEdgeMap().put(edgeCount,e1);
                // edgeCount++;
                Edge e2 = new Edge(vertexMap.get(j),vertexMap.get(i),edgeCount);
                vertexMap.get(j).getEdgeMap().put(edgeCount,e2);
                edgeCount++;
                edgeList.add(e1);
            }
        }
        Collections.sort(edgeList, new Comparator<Edge>() {
            @Override
            public int compare(Edge o1, Edge o2) {
                return ((Double)o1.getDistance()).compareTo( o2.getDistance());
            }
        });
    }
}
```



```

public void computeAllEdges(){
    int edgeCount=1;
    for(int i=1;i<=n;i++){
        for(int j=i+1;j<=n;j++){
            Edge e1 = new Edge(vertexMap.get(i),vertexMap.get(j),edgeCount);
            edgeCount++;
            edgeList.add(e1);
        }
    }
    Collections.sort(edgeList, new Comparator<Edge>() {
        @Override
        public int compare(Edge o1, Edge o2) {
            return ((Double)o1.getDistance()).compareTo( o2.getDistance());
        }
    });
}

public double deletionHeuristics(){
    createCompleteGraph();
    int index = edgeList.size()-1;
    double cost=0;
    while(index>=0){
        Edge edge = edgeList.get(index);
        int fromNode = edge.from.vertexNum;
        int toNode = edge.to.vertexNum;
        Edge edge1 = vertexMap.get(fromNode).getEdgeMap().remove(edge.edgeNum);
        Edge edge2 = vertexMap.get(toNode).getEdgeMap().remove(edge.edgeNum);
        if(diameterAndDegreeCheck()){
            //System.out.println("index="+index);
            edgeList.set(index,null);
        } else{
            vertexMap.get(fromNode).getEdgeMap().put(edge1.edgeNum,edge1);
            vertexMap.get(toNode).getEdgeMap().put(edge2.edgeNum,edge2);
        }
        index--;
    }
    for(Edge edge: edgeList){
        if(edge!=null){
            // System.out.println("edge:
            "+edge.from.vertexNum+", "+edge.to.vertexNum+ " : "+edge.distance);
            cost+=edge.distance;
        }
    }
    return cost;
}

public boolean diameterAndDegreeCheck(){
    Iterator<Integer> itr = vertexMap.keySet().iterator();
    while (itr.hasNext()){
        int node = itr.next();
        if(vertexMap.get(node).getEdgeMap().size()<3)
            return false;
    }

    itr = vertexMap.keySet().iterator();
    while (itr.hasNext()){

```

```

        int node = itr.next();
        if(!bfs(node)){
            return false;
        }
    }
    return true;
}

public boolean bfs(int s){
    Set<Integer> visited = new HashSet<>();
    LinkedList<Integer> queue = new LinkedList<Integer>();
    visited.add(s);
    queue.add(s);
    int hopCount=0;
    while (queue.size() != 0 && hopCount<4 )
    {
        s = queue.poll();
        for (Map.Entry<Integer, Edge> entry :
vertexMap.get(s).getEdgeMap().entrySet()){
            int toNode = entry.getValue().to.vertexNum;
            if (!visited.contains(toNode))
            {
                visited.add(toNode);
                queue.add(toNode);
            }
        }
        hopCount++;
    }

    if(visited.size()==n)
        return true;
    return false;
}

public double constructionHeuristics(){
    computeAllEdges();
    int index = 0;
    double cost=0;
    List<Edge> result = new ArrayList<>();
    while(index < edgeList.size()){
        Edge edge = edgeList.get(index);
        result.add(edge);
        int fromNode = edge.from.vertexNum;
        int toNode = edge.to.vertexNum;
        Edge edge1 = new Edge(edge.from, edge.to, edge.edgeNum);
        Edge edge2 = new Edge(edge.to, edge.from, edge.edgeNum);
        vertexMap.get(fromNode).getEdgeMap().put(edge1.edgeNum, edge1);
        vertexMap.get(toNode).getEdgeMap().put(edge2.edgeNum, edge2);
        if(diameterAndDegreeCheck()){
            break;
        }
        index++;
    }
}

```

```

        for(Edge edge: result){
            if(edge!=null){
                // System.out.println("edge:
"+edge.from.vertexNum+", "+edge.to.vertexNum+ " : "+edge.distance);
                cost+=edge.distance;
            }
        }
        edgeList = result;
        return cost;
    }

    public static void main(String[] arg) throws IOException {
        for(int n=15;n<20;n++){
            Graph g = new Graph(n);
            System.out.println("Deletion algorithm n= "+n+" cost= "+
g.deletionHeuristics());

            if(n==15 || n==16){
                BufferedWriter fw = new BufferedWriter(new
FileWriter("deletionAlgo_n_"+n+"_graph.csv"));
                for(Edge edge: g.edgeList){
                    if(edge!=null){
                        // System.out.println("edge:
"+edge.from.vertexNum+", "+edge.to.vertexNum+ " : "+edge.distance);
                        fw.write(edge.from.vertexNum+", "+edge.to.vertexNum+ ", "+
edge.distance);
                        fw.newLine();
                    }
                }
                fw.close();
            }
        }

        for(int n=15;n<20;n++){
            Graph g = new Graph(n);
            System.out.println("Construction Algorithm n= "+n+" cost= "+
g.constructionHeuristics());
            if(n==15 || n==16){
                BufferedWriter fw = new BufferedWriter(new
FileWriter("constructionAlgo_n_"+n+"_graph.csv"));
                for(Edge edge: g.edgeList){
                    if(edge!=null){
                        //System.out.println("edge:
"+edge.from.vertexNum+", "+edge.to.vertexNum+ " : "+edge.distance);
                        fw.write(edge.from.vertexNum+", "+edge.to.vertexNum+ ", "+
edge.distance);
                        fw.newLine();
                    }
                }
                fw.close();
            }
        }
    }
}

```

```
    }  
}
```

Vertex.java

```
package project3;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class Vertex {  
    Map<Integer,Edge> edgeMap;  
    int vertexNum;  
    int x;  
    int y;  
  
    Vertex(int vertexNum, int x, int y){  
        this.vertexNum = vertexNum;  
        this.x = x;  
        this.y = y;  
        edgeMap = new HashMap<>();  
    }  
    public Map<Integer, Edge> getEdgeMap() {  
        return edgeMap;  
    }  
  
    public void setEdgeMapt(Map<Integer, Edge> edgeMap) {  
        this.edgeMap = edgeMap;  
    }  
  
    public int getVertexNum() {  
        return vertexNum;  
    }  
  
    public void setVertexNum(int vertexNum) {  
        this.vertexNum = vertexNum;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
  
    public int getDegree(){
```

```

        return edgeMap.size();
    }
    public int hashCode() {
        return vertexNum;
    }
}

```

Edge.java

```

package project3;

public class Edge {
    Vertex from;
    Vertex to;
    double distance;
    int edgeNum;

    Edge(Vertex from, Vertex to, int edgeNum){
        this.from = from;
        this.to = to;
        this.edgeNum = edgeNum;
        distance = Math.sqrt(Math.pow((from.x - to.x),2) + Math.pow((from.y -
to.y),2));
    }

    public Vertex getFrom() {
        return from;
    }

    public void setFrom(Vertex from) {
        this.from = from;
    }

    public Vertex getTo() {
        return to;
    }

    public void setTo(Vertex to) {
        this.to = to;
    }

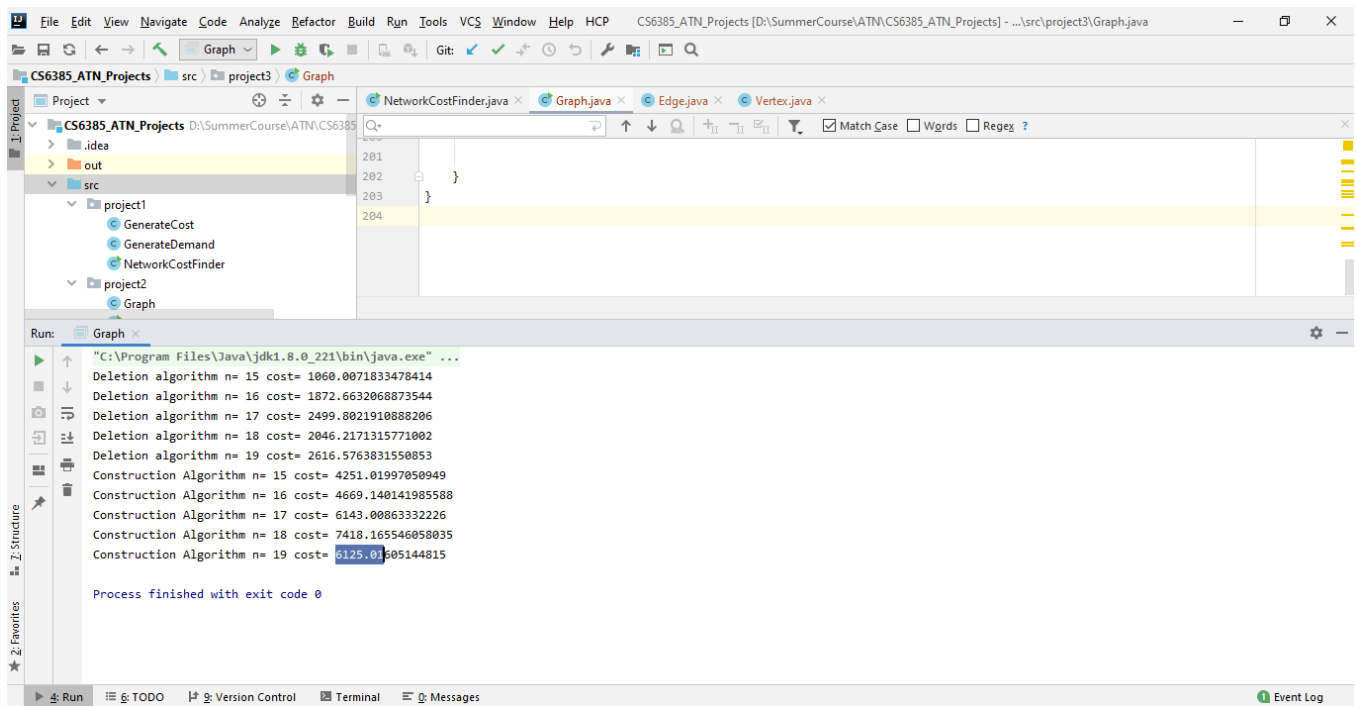
    public double getDistance() {
        return distance;
    }

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

```

## Program Output:

### Console output:



```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Deletion algorithm n= 15 cost= 1060.0071833478414
Deletion algorithm n= 16 cost= 1872.6632068873544
Deletion algorithm n= 17 cost= 2499.8021910888206
Deletion algorithm n= 18 cost= 2046.2171315771002
Deletion algorithm n= 19 cost= 2616.5763831550853
Construction Algorithm n= 15 cost= 4251.01997050949
Construction Algorithm n= 16 cost= 4669.140141985588
Construction Algorithm n= 17 cost= 6143.00863332226
Construction Algorithm n= 18 cost= 7418.165546058035
Construction Algorithm n= 19 cost= 9125.01305144815

Process finished with exit code 0
```

## Reference

1. <http://www.terpconnect.umd.edu/~raghavan/preprints/heurtut.pdf>
2. <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
3. <https://cytoscape.org/>