CS 6385.0W1 – Summer 2017

Algorithmic Aspects of Telecommunication Network

Project:3
Network Topology Design of Nodes

by

Bhakti Khatri [Net ID: brk160030]

➢ Project Description:

The theme of this project is to create a Network Topology having given the location of n nodes in a plane by their coordinates. In order to implement this network topology design problem, the aim is to create and implement two heuristic algorithms. Here, heuristic means that it just have to be a reasonable solution and not an optimum one.

➢ Network Description:

➢ Network topology:

Create a network topology such that it has the following properties:

1. It contains all the given nodes.
2. The degree of each vertex in the graph is at least 3, that is, each node is connected to at least 3 other nodes.
3. 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. (This can be checked by any shortest path algorithm.) 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.
4. 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.

➢ Inputs and Outputs:

Input is location of n nodes in a plane is given by their coordinates.

Output is a network topology (represented by an undirected graph) such that it has the above mentioned properties.

➢ Scenarios:

Run the program on randomly generated examples (at least 5 examples), with both algorithms. The instances are created as follows. Pick n random points in the plane. This can be done by generating random numbers in some range and taking them as coordinates of the points. The value of n should be chosen such that the problem is not trivially small (say, n >= 15), but the running time is still reasonable (it does not run for days). Show how the random input is generated and present the actual data.

➢ Assumptions:

Network topology is a undirected graph. We are assuming that the nodes are always connected.

➢ Goal:

Given the coordinate location of n nodes in a plane, create a network topology design such that it satisfies the given properties. Thus, the goal is to create and implement two different heuristic algorithms for this network topology design problem, and experiment with them.

➢ Heuristic Algorithms:

1. **Constructive Heuristic Algorithm**
2. **Local Search Heuristic Algorithm**

1. **Constructive Heuristic Algorithm:** A constructive heuristic algorithm is a type of heuristic method which starts with an empty solution and repeatedly extends the current solution until a complete solution is obtained. Examples of some constructive heuristics developed for famous problems are: flow shop scheduling, vehicle routing problem and open shop problem.

○ Constructive Algorithm:

In combinatorial optimization problems, every solution x is a subset of the ground set E.
A constructive heuristic iteratively updates a subset x (t) as follows:

1. it starts from an empty subset: x (0) = ∅ (it is a subset of the optimal solution)

2. at each iteration t, if selects an element i (t) ∈ E as "the best one" among the "admissible" elements (one tries to make x(t) to be a subset of a feasible and optimal solution)
3. inserts i (t) in the current subset x (t) : x (t+1) := x (t) ∪ i (t) (no backtrack is allowed)
4. it goes back to step 2 until the solution is complete (if further enlarged, the solution would not remain feasible)

○ Constructive Psuedo Code:

A constructive algorithm (for minimization) can be described as follows:

```
Algorithm Greedy(I)
x := ∅;
x ∗ := ∅; f ∗ := +∞; { Best incumbent solution }
While Ext(x) 6= ∅ do
i := arg min i∈Ext(x) ϕ (i, x);
x := x ∪ {i};
If x ∈ X and f (x) < f ∗ then x∗ := x; f ∗ := f (x);
 Return (x ∗ , f ∗ );
```

The sequence of subsets visited by the algorithm depends on:

• the search space F, i.e. the set Ext(x);
• the function ϕ : E × F → R we use to select how to extend the current subset x (t) → x (t+1)

○ How it works:

A constructive heuristic finds the optimum when the current subset x (t) at every iteration t is contained in an optimal solution.
This property is valid for x (0) = ∅, but is usually lost in some later iteration t.
A constructive heuristic executes at most n = |E| iterations. The complexity of each step is affected by:
1. the computation of Ext(x);
2. the evaluation of ϕ (i, x) for each i ∈ Ext(x);
3. the selection of the minimum value and the corresponding element;
4. the update of x (and possibly other data-structures). In general, the resulting complexity turns out to be a low order polynomial. T (n) = n (γExt (n) + γϕ (n)).

2. **Local Search Heuristic Algorithm:** In computer science, local search is a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

   - ### Local Search Pseudo Code:

     Local search is an iterative algorithm that moves from one solution S to another S' according to some neighbourhood structure.

     Local search procedure usually consists of the following steps:

   1. Initialisation. Choose an initial schedule S to be the current solution and compute the value of the objective function F(S).
   2. Neighbour Generation. Select a neighbour S' of the current solution S and compute F(S').
   3. Acceptance Test. Test whether to accept the move from S to S'. If the move is accepted, then S' replaces S as the current solution; otherwise S is retained as the current solution.
   4. Termination Test. Test whether the algorithm should terminate. If it terminates, output the best solution generated; otherwise, return to the neighbour generation step.

   We consider four local search algorithms: Iterative Improvement, Threshold Accepting, Simulated Annealing, and Tabu Search. For all of them, steps 1, 2, and 4 are the same while step 3 is different.

   We assume that a schedule is represented as a permutation of job numbers (J1, J2, … , Jn). This can always be done for a single machine processing system or for permutation flow shop; for other models more complicate structures are used.

   In Step 1, a starting solution can be obtained by one of the constructive heuristics described in the previous lectures or it can be specified by a random job permutation. If local search procedure is applied several times, then it is reasonable to use random initial schedules.

To generate a neighbour S’ in Step 2, a neighbourhood structure should be specified beforehand. Often the following types of neighbourhoods are considered:
- transpose neighbourhood in which two jobs occupying adjacent positions in the sequence are interchanged: (1, 2, 3, 4, 5, 6, 7) → (1, 3, 2, 4, 5, 6, 7);
- swap neighbourhood in which two arbitrary jobs are interchanged: (1, 2, 3, 4, 5, 6, 7) → (1, 6, 3, 4, 5, 2, 7);
- insert neighbourhood in which one job is removed from its current position and inserted elsewhere: (1, 2, 3, 4, 5, 6, 7) → (1, 3, 4, 5, 6, 2, 7).

Neighbours can be generated randomly, systematically, or by some combination of the two approaches.

In Step 3, the acceptance rule is usually based on values F(S) and F(S’) of the objective function for schedules S and S’. In some algorithms only moves to ‘better’ schedules are accepted (schedule S’ is better than S if F(S’) < F(S); in others it may be allowed to move to ‘worse’ schedules. Sometimes ‘wait and see’ approach is adopted.

The algorithm terminates in Step 4, if the computation time exceeds the prespecified limit or after completing the prespecified number of iterations.

In what follows we specify Step 3 “Acceptance Test” for each type of the local search algorithm.

➢ Program Details:

- It consists of implementation of two heuristic algorithms: Local Search & Constructive Heuristic Algorithms.
- Run the file ExperimentAlgorithms.java. It shows analysis of both the algorithms.
- A total of 6 experiments are run which can be configured using numberOfExamples in ExperimentAlgorithms.java
- Each experiment simulates a graph of 18 nodes in a 2d plane with x-y cords lying between (0,0) and (100,100)
- Both the algorithms are run to find optimal cost of the total graph subject to the following conditions:

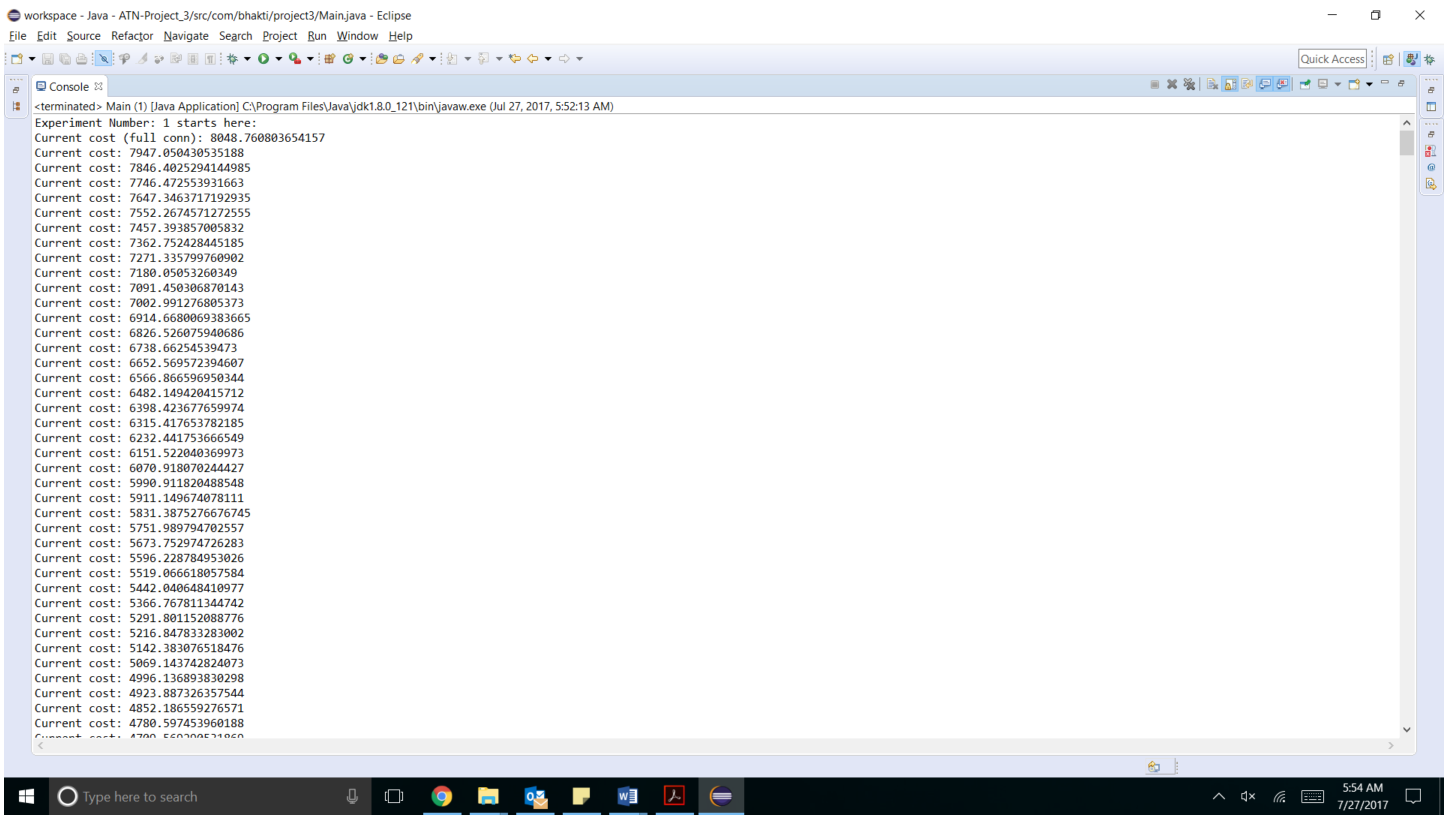
  a) Graph should be connected

b) Graph should have degree of at least 3 for each node
c) Diameter of graph should be at most 4

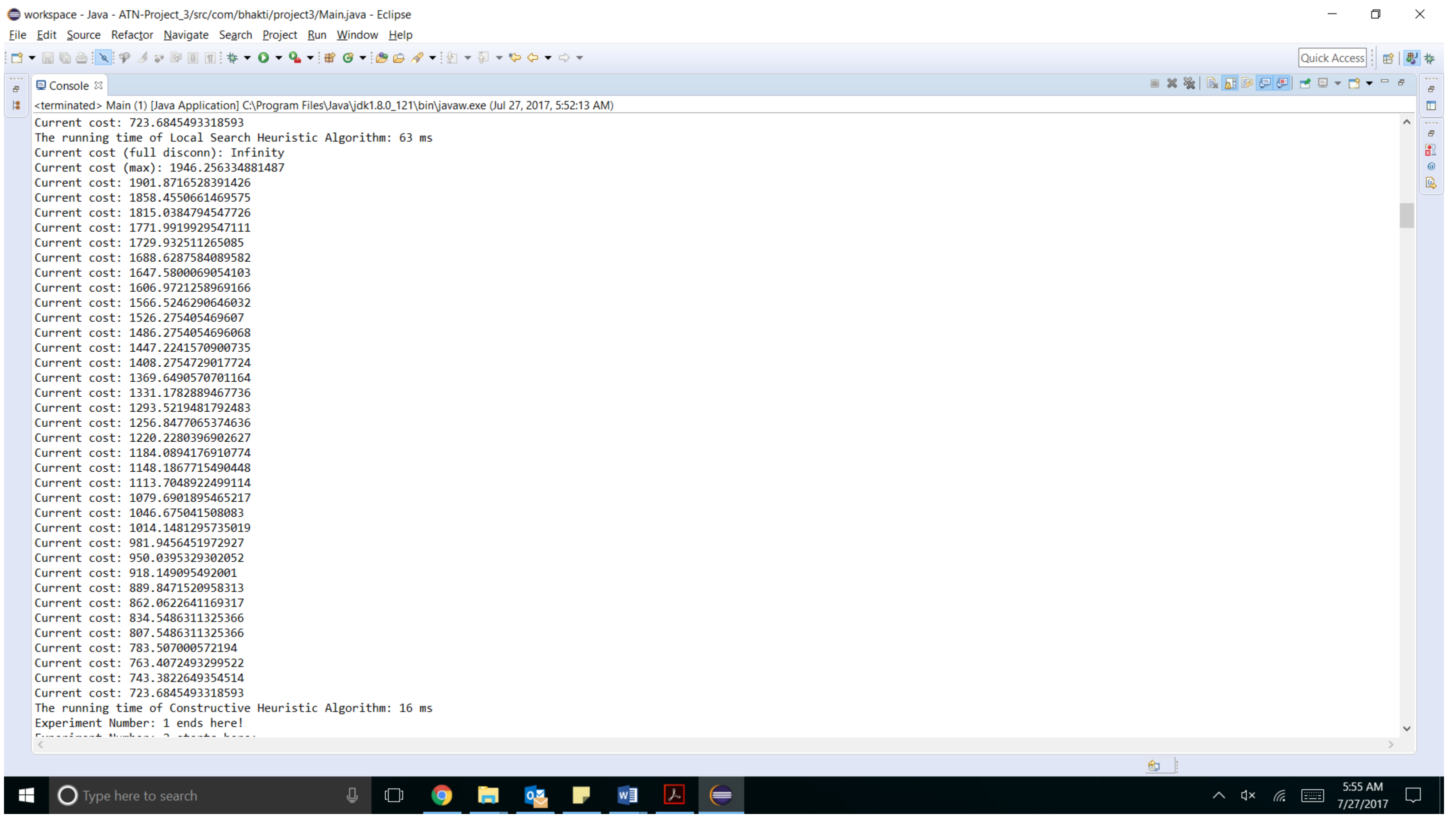
a) is ensured by CheckConnectedNodes.java logic
b) is ensured by implementing logic in GraphProperties.java
c) is ensured by Dijkstra's shortest path applied on the graph with weights = 1

➢ ReadMe:

- ❖ Run ExperimentAlgorithms.java which gives the following output in console window:


workspace - Java - ATN-Project_3/src/com/bhakti/project3/Main.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Console
Type here to search
5:54 AM
7/27/2017

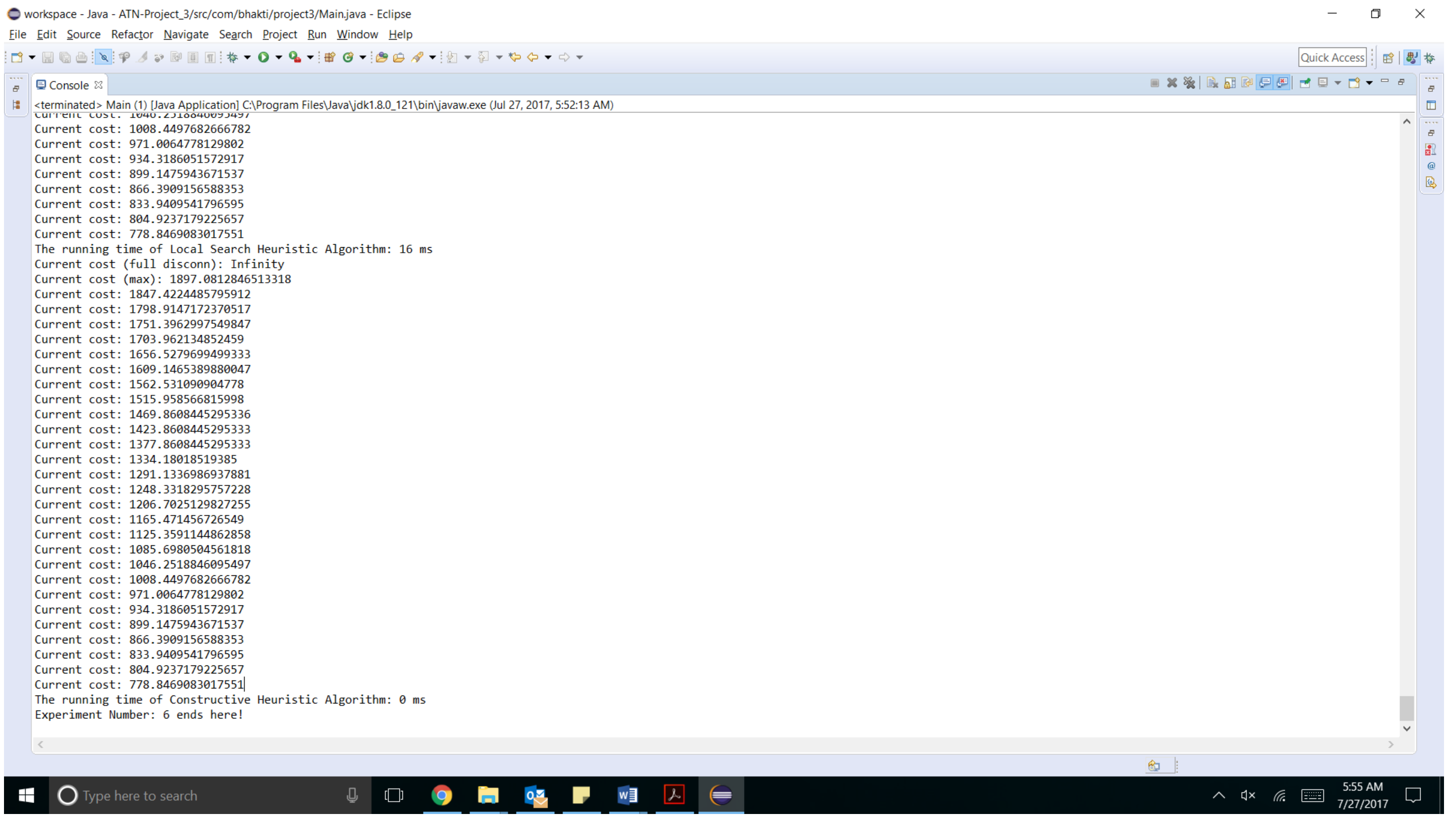
```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Jul 27, 2017, 5:52:13 AM)
Experiment Number: 1 starts here:
Current cost (full conn): 8048.760803654157
Current cost: 7947.050430535188
Current cost: 7846.4025294144985
Current cost: 7746.472553931663
Current cost: 7647.3463717192935
Current cost: 7552.2674571272555
Current cost: 7457.393857005832
Current cost: 7362.752428445185
Current cost: 7271.335799760902
Current cost: 7180.05053260349
Current cost: 7091.450306870143
Current cost: 7002.991276805373
Current cost: 6914.6680069383665
Current cost: 6826.526075940686
Current cost: 6738.66254539473
Current cost: 6652.569572394607
Current cost: 6566.866596950344
Current cost: 6482.149420415712
Current cost: 6398.423677659974
Current cost: 6315.417653782185
Current cost: 6232.441753666549
Current cost: 6151.522040369973
Current cost: 6070.918070244427
Current cost: 5990.911820488548
Current cost: 5911.149674078111
Current cost: 5831.3875276676745
Current cost: 5751.989794702557
Current cost: 5673.752974726283
Current cost: 5596.228784953026
Current cost: 5519.066618057584
Current cost: 5442.040648410977
Current cost: 5366.767811344742
Current cost: 5291.801152088776
Current cost: 5216.847833283002
Current cost: 5142.383076518476
Current cost: 5069.143742824073
Current cost: 4996.136893830298
Current cost: 4923.887326357544
Current cost: 4852.186559276571
Current cost: 4780.597453960188
```



workspace - Java - ATN-Project_3/src/com/bhakti/project3/Main.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Console
Type here to search
5:55 AM
7/27/2017

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Jul 27, 2017, 5:52:13 AM)
Current cost: 723.6845493318593
The running time of Local Search Heuristic Algorithm: 63 ms
Current cost (full disconn): Infinity
Current cost (max): 1946.256334881487
Current cost: 1901.8716528391426
Current cost: 1858.4550661469575
Current cost: 1815.0384794547726
Current cost: 1771.9919929547111
Current cost: 1729.932511265085
Current cost: 1688.6287584089582
Current cost: 1647.5800069054103
Current cost: 1606.9721258969166
Current cost: 1566.5246290646032
Current cost: 1526.275405469607
Current cost: 1486.2754054696068
Current cost: 1447.2241570900735
Current cost: 1408.2754729017724
Current cost: 1369.6490570701164
Current cost: 1331.1782889467736
Current cost: 1293.5219481792483
Current cost: 1256.8477065374636
Current cost: 1220.2280396902627
Current cost: 1184.0894176910774
Current cost: 1148.1867715490448
Current cost: 1113.7048922499114
Current cost: 1079.6901895465217
Current cost: 1046.675041508083
Current cost: 1014.1481295735019
Current cost: 981.9456451972927
Current cost: 950.0395329302052
Current cost: 918.149095492001
Current cost: 889.8471520958313
Current cost: 862.0622641169317
Current cost: 834.5486311325366
Current cost: 807.5486311325366
Current cost: 783.507000572194
Current cost: 763.4072493299522
Current cost: 743.3822649354514
Current cost: 723.6845493318593
The running time of Constructive Heuristic Algorithm: 16 ms
Experiment Number: 1 ends here!
```



workspace - Java - ATN-Project_3/src/com/bhakti/project3/Main.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Console
Type here to search
5:55 AM
7/27/2017

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Jul 27, 2017, 5:52:13 AM)
Current cost: 1008.4497682666782
Current cost: 971.0064778129802
Current cost: 934.3186051572917
Current cost: 899.1475943671537
Current cost: 866.3909156588353
Current cost: 833.9409541796595
Current cost: 804.9237179225657
Current cost: 778.8469083017551
The running time of Local Search Heuristic Algorithm: 16 ms
Current cost (full disconn): Infinity
Current cost (max): 1897.0812846513318
Current cost: 1847.4224485795912
Current cost: 1798.9147172370517
Current cost: 1751.3962997549847
Current cost: 1703.962134852459
Current cost: 1656.5279699499333
Current cost: 1609.1465389880047
Current cost: 1562.531090904778
Current cost: 1515.958566815998
Current cost: 1469.8608445295336
Current cost: 1423.8608445295333
Current cost: 1377.8608445295333
Current cost: 1334.18018519385
Current cost: 1291.1336986937881
Current cost: 1248.3318295757228
Current cost: 1206.7025129827255
Current cost: 1165.471456726549
Current cost: 1125.3591144862858
Current cost: 1085.6980504561818
Current cost: 1046.2518846095497
Current cost: 1008.4497682666782
Current cost: 971.0064778129802
Current cost: 934.3186051572917
Current cost: 899.1475943671537
Current cost: 866.3909156588353
Current cost: 833.9409541796595
Current cost: 804.9237179225657
Current cost: 778.8469083017551
The running time of Constructive Heuristic Algorithm: 0 ms
Experiment Number: 6 ends here!
```

➢ Output experimental values:

❖ **Experiment Run-1:**

Points:

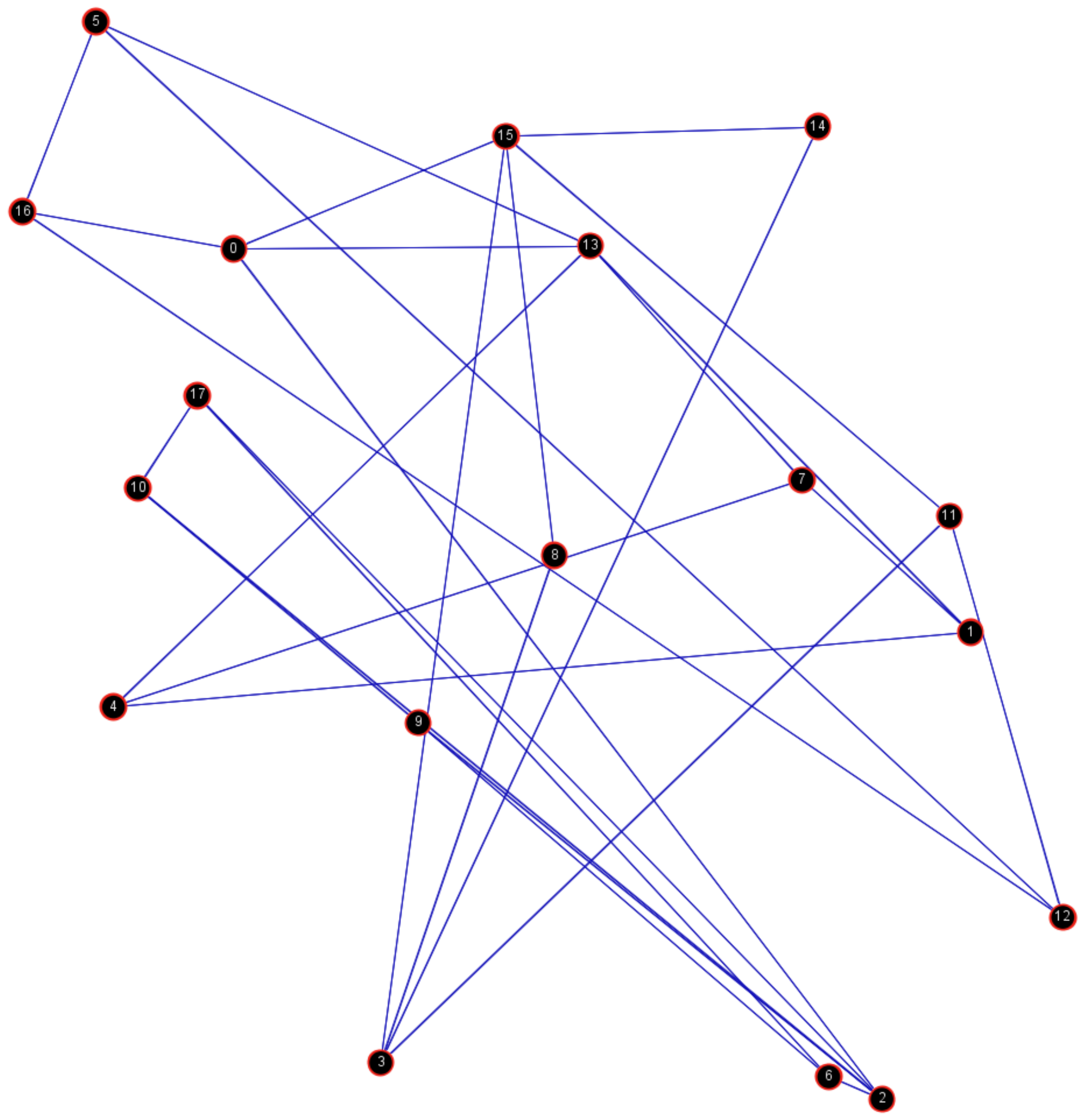
N	Point (x-coordinates)	Point (y-coordinates)
1	82	48
2	57	59
3	76	34
4	24	93
5	76	79
6	70	94
7	18	55
8	40	56
9	15	99
10	23	86
11	56	11
12	81	32
13	88	69
14	92	22
15	40	35
16	35	84
17	75	41
18	72	4

Node-Edges Connection:

Source	Target
0	2
0	13
0	15
0	16
1	4
1	7
1	13
2	0
2	6
2	9
2	10
2	17
3	8
3	11
3	14
3	15
4	1
4	7
4	13
5	12
5	13
5	16
6	2
6	9
6	17

7	1
7	4
7	13
8	3
8	15
9	2
9	6
9	10
10	2
10	9
10	17
11	3
11	12
11	15
12	5
12	11
12	16
13	0
13	1
13	4
13	5
13	7
14	3
14	15
15	0
15	3
15	8
15	11
15	14
16	0
16	5
16	12
17	2
17	6
17	10

Network Topology (Graph):



- **Experiment Run-2:**
  Points:

n	Point (x-coordinates)	Point (y-coordinates)
1	72	2
2	66	71
3	53	99
4	76	62
5	41	39
6	62	1
7	70	41

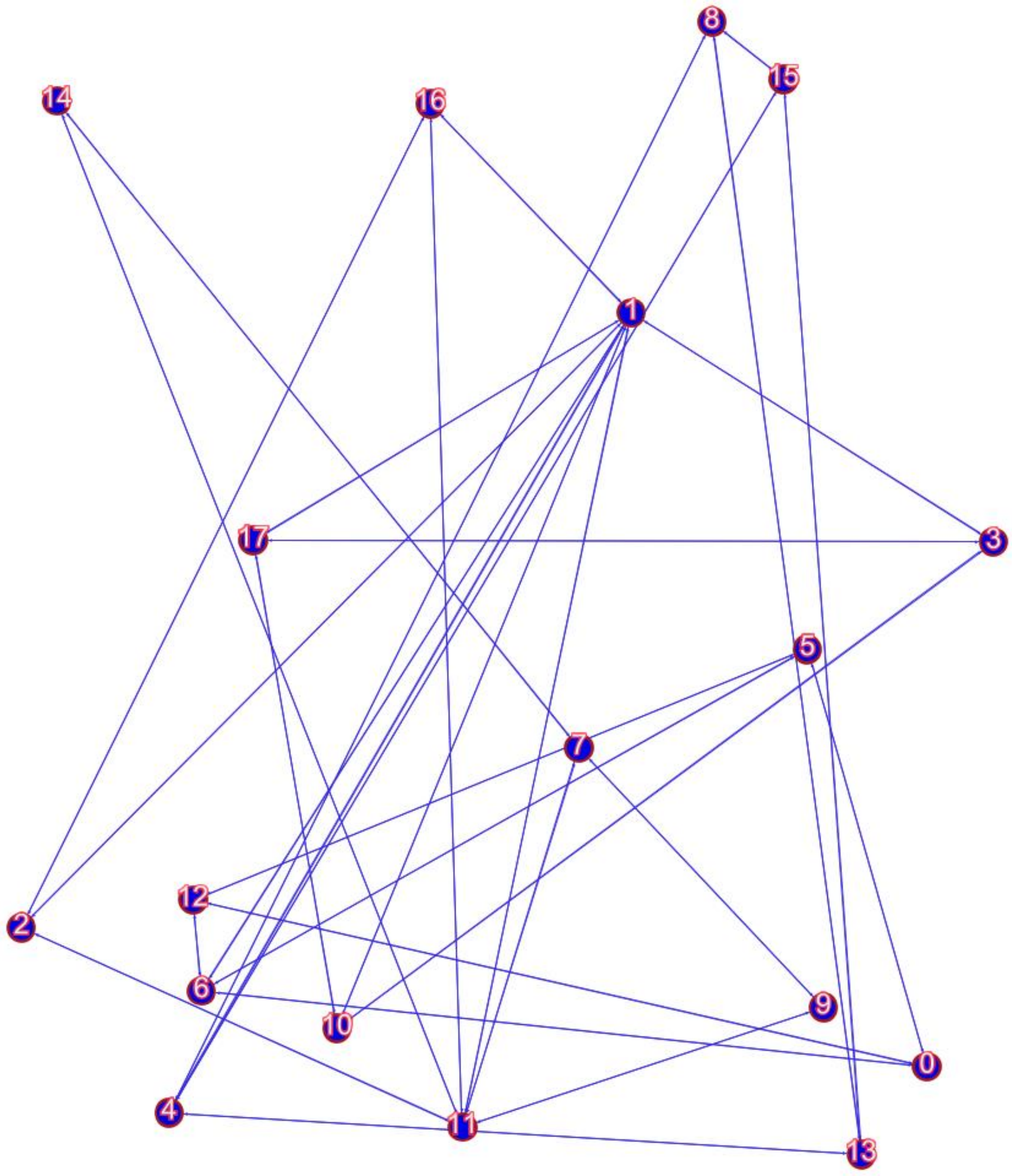
8	0	59
9	34	14
10	5	85
11	93	63
12	28	92
13	57	31
14	27	5
15	6	86
16	8	11
17	48	98
18	91	49

Node-Edges Connection:

Source	Target
0	5
0	6
0	12
1	2
1	3
1	4
1	6
1	10
1	11
1	16
1	17
2	1
2	11
2	16
3	1
3	10
3	17
4	1
4	8
4	13
4	15
5	0
5	6
5	12
6	0
6	1
6	5
6	12
7	9
7	11
7	14
8	4
8	13
8	15
9	7
9	11
10	1

10	3
10	17
11	1
11	2
11	7
11	9
11	14
11	16
12	0
12	5
12	6
13	4
13	8
13	15
14	7
14	11
15	4
15	8
15	13
16	1
16	2
16	11
17	1
17	3
17	10

Network Topology (Graph):



❖ **Experiment Run-3:**

Points:

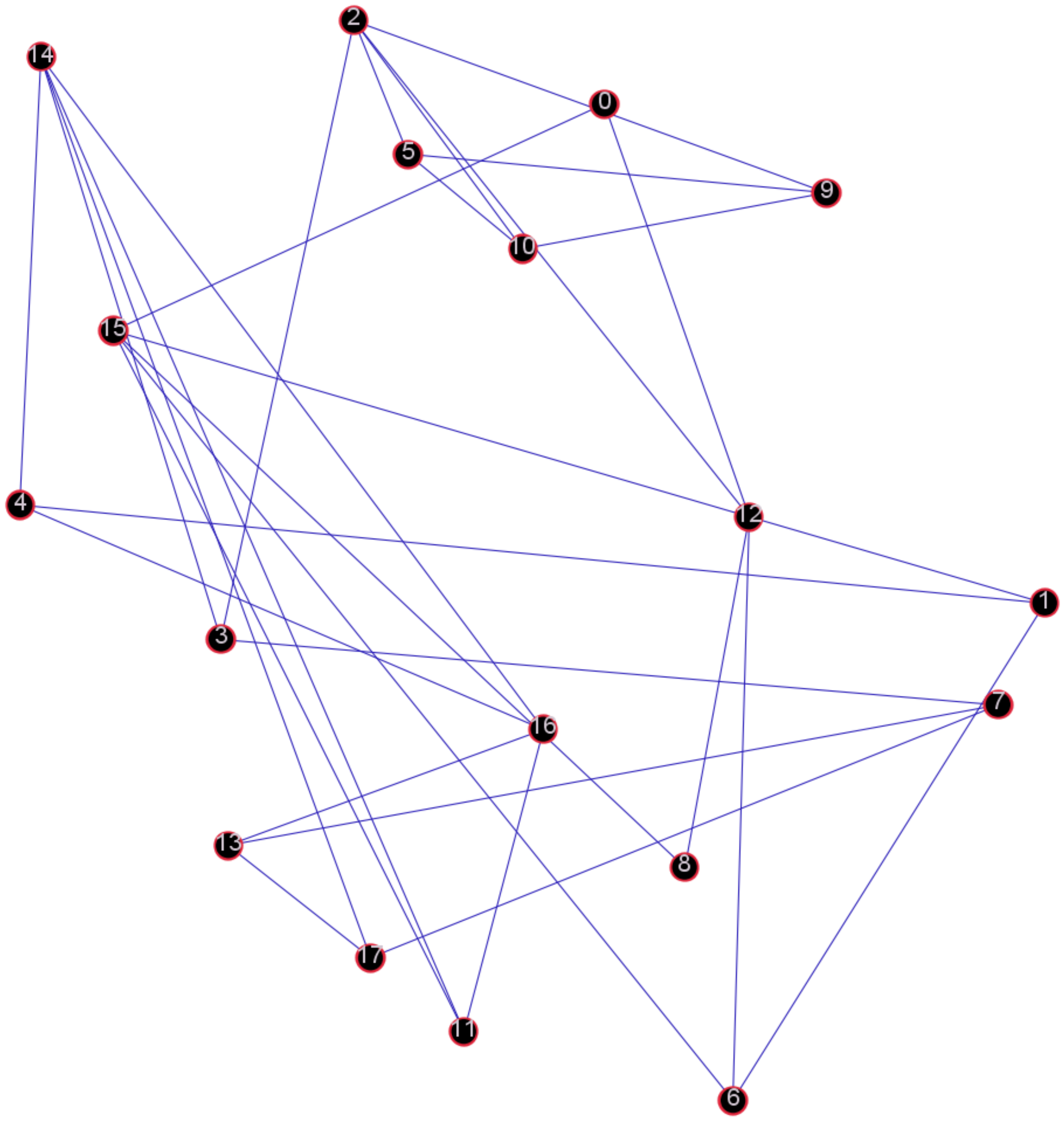
n	Point (x-coordinates)	Point (y-coordinates)
1	88	56
2	53	93
3	60	31
4	50	26
5	18	84
6	80	16
7	71	99
8	7	4
9	93	56
10	94	16
11	86	10
12	27	62
13	76	62
14	4	43
15	26	50
16	58	74
17	4	66
18	15	36

Node-Edges Connection:

Source	Target
0	12
0	15
1	4
1	6
1	15
2	3
2	5
2	9
2	10
2	12
3	2
3	7
3	14
4	1
4	14
4	16
5	2
5	9
5	10
6	1
6	12
6	15
7	3
7	13
7	17
8	12

8	15
9	2
9	5
9	10
10	2
10	5
10	9
11	14
11	15
11	16
12	0
12	2
12	6
12	8
13	7
13	16
13	17
14	3
14	4
14	11
14	16
14	17
15	0
15	1
15	6
15	8
15	11
16	4
16	11
16	13
16	14
17	7
17	13
17	14

Network Topology (Graph):



❖ **Experiment Run-4:**

Points:

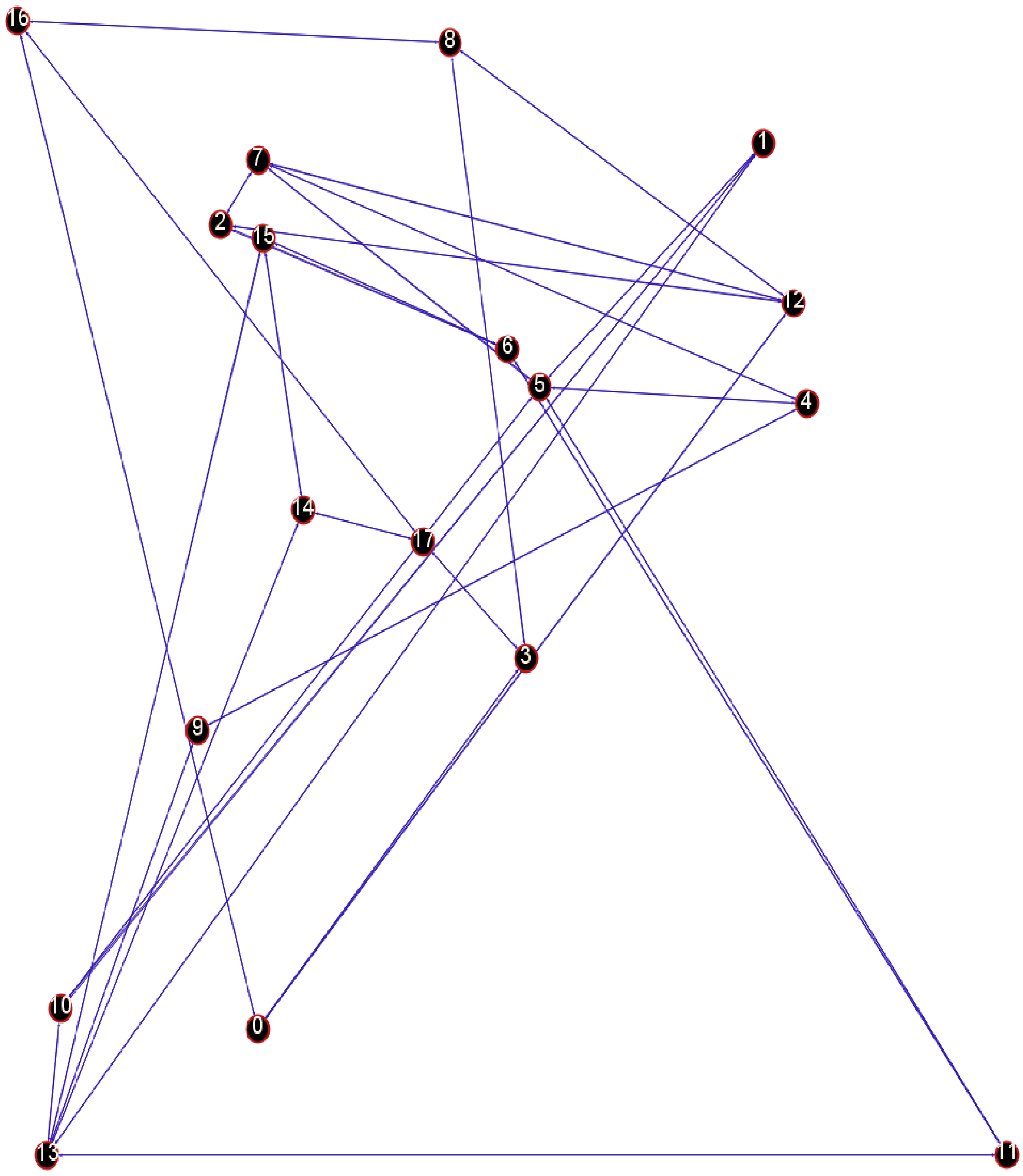
n	Point (x-coordinates)	Point (y-coordinates)
1	33	97
2	94	31
3	76	90
4	39	78
5	94	63
6	89	50
7	73	59
8	85	81
9	39	87
10	90	52
11	89	11
12	77	57
13	63	89
14	75	42
15	24	19
16	45	25
17	2	93
18	22	65

Node-Edges Connection:

Source	Target
0	3
0	12
0	16
1	5
1	10
1	13
2	6
2	7
2	12
3	0
3	8
3	17
4	5
4	7
4	9
5	1
5	4
5	7
5	10
5	11
6	2
6	11
6	15
7	2
7	4
7	5

7	12
8	3
8	12
8	16
9	4
9	13
10	1
10	5
10	13
11	5
11	6
11	13
12	0
12	2
12	7
12	8
13	1
13	9
13	10
13	11
13	14
13	15
14	13
14	15
14	17
15	6
15	13
15	14
16	0
16	8
16	17
17	3
17	14
17	16

Network Topology (Graph):



- **Experiment Run-5:**

Points:

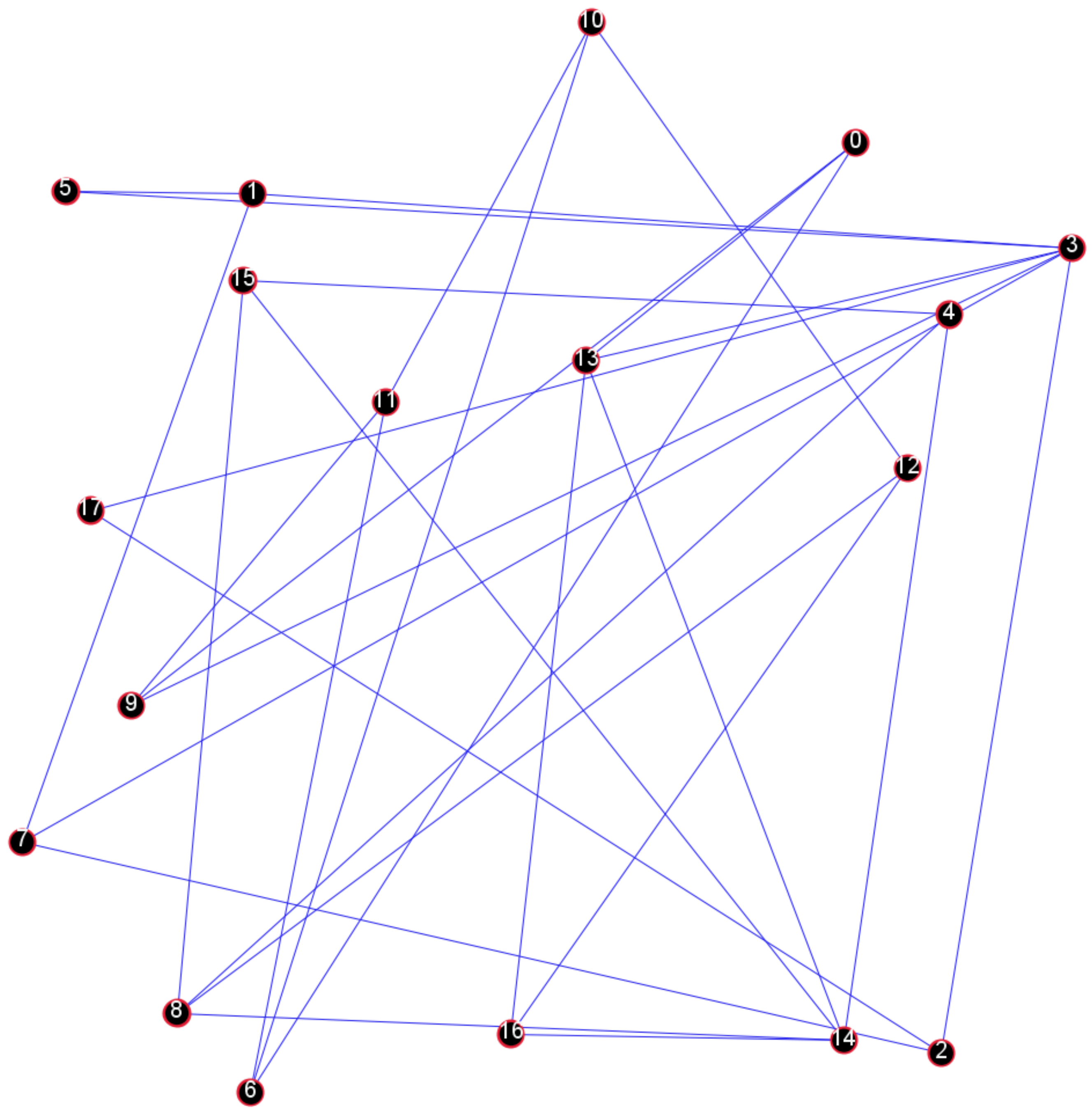
n	Point (x-coordinates)	Point (y-coordinates)
1	46	28
2	78	27
3	76	1
4	71	18
5	44	93
6	99	23
7	30	28
8	70	20
9	27	69
10	49	14
11	8	22
12	34	0
13	14	57
14	55	37
15	47	56
16	34	85
17	39	43
18	92	18

Node-Edges Connection:

Source	Target
0	6
0	9
0	13
1	3
1	5
1	7
2	3
2	7
2	17
3	1
3	2
3	5
3	7
3	9
3	13
3	17
4	8
4	14
4	15
5	1
5	3
6	0
6	10
6	11
7	1
7	2

7	3
8	4
8	12
8	14
8	15
9	0
9	3
9	11
10	6
10	11
10	12
11	6
11	9
11	10
12	8
12	10
12	16
13	0
13	3
13	14
13	16
14	4
14	8
14	13
14	15
14	16
15	4
15	8
15	14
16	12
16	13
16	14
17	2
17	3

Network Topology (Graph):



- **Experiment Run-6:**

Points:

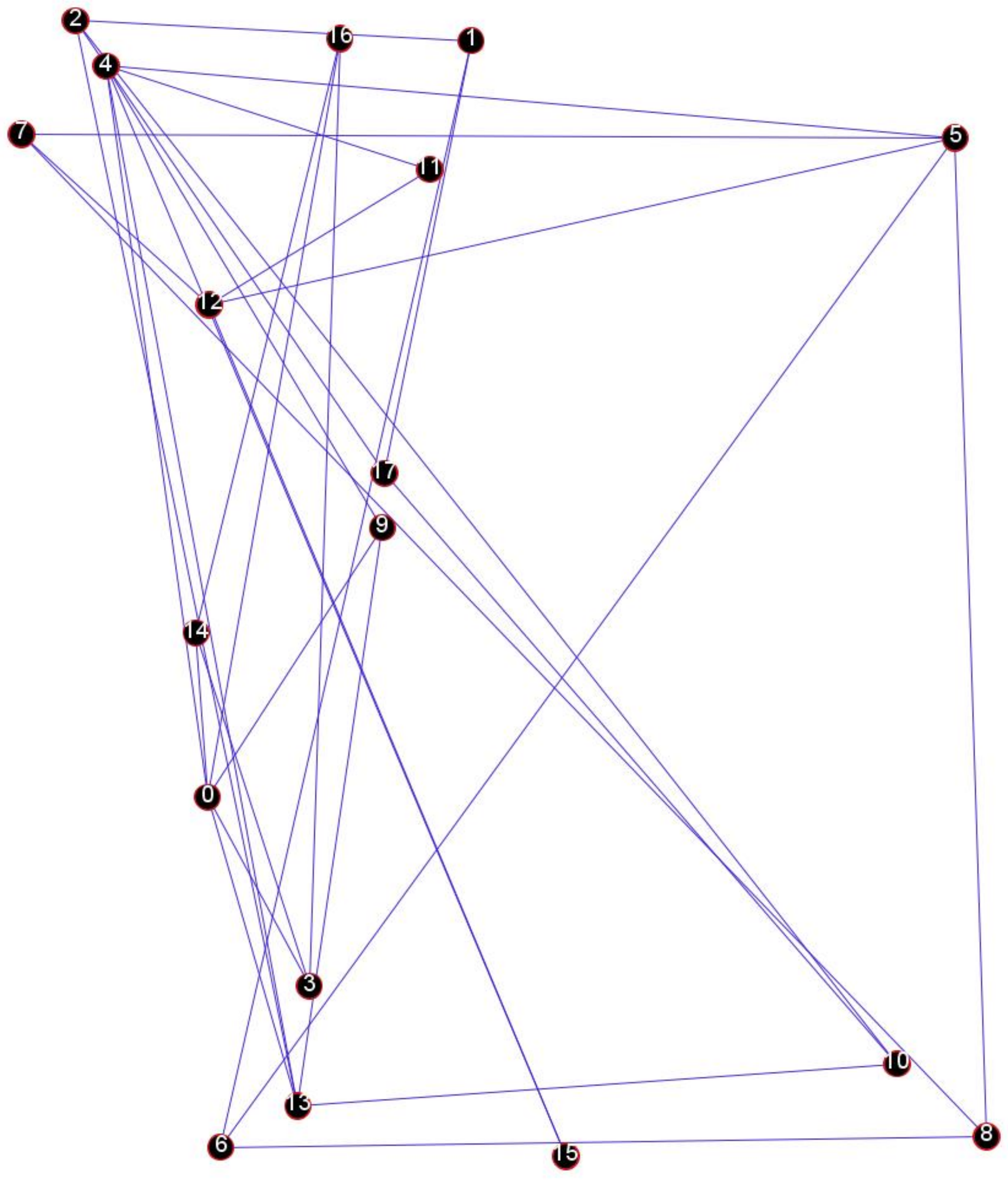
n	Point (x-coordinates)	Point (y-coordinates)
1	23	20
2	26	43
3	18	96
4	89	35
5	48	78
6	88	19
7	65	37
8	91	82
9	36	58
10	18	28
11	98	76
12	14	49
13	32	46
14	99	15
15	76	15
16	47	41
17	58	38
18	5	11

Node-Edges Connection:

Source	Target
0	3
0	4
0	9
0	13
0	14
0	16
1	2
1	6
1	17
2	1
2	10
2	13
2	17
3	0
3	14
3	16
4	0
4	5
4	9
4	11
4	13
4	15
5	4
5	6
5	7
5	8

5	12
6	1
6	5
6	8
7	5
7	8
7	12
8	5
8	6
8	7
9	0
9	4
9	13
10	2
10	13
10	17
11	4
11	12
12	5
12	7
12	11
12	15
13	0
13	2
13	4
13	9
13	10
14	0
14	3
14	16
15	4
15	12
16	0
16	3
16	14
17	1
17	2
17	10

Network Topology (Graph):



➢ Observation:

Thus, observing the costs from the results, we can infer that the Constructive Heuristic Algorithm takes less amount of time and cost to build the complete solution. Constructive algorithm extends the empty solution until we get the complete solution whereas Local Search algorithm takes the complete solution and tries to improve it via local moves. Thus, it is better among the two algorithms.

➢ Conclusion:

The cost of these algorithms at first decreases and then increases. Thus, it is not steady but if more than 25 iterations are performed then it may lead to a decrease in the cost. 20 iterations are enough to reach a solution that does not improve significantly in further iterations. The experimental running time of the algorithms keeps on decreasing as we increase the iterations.

➢ References:

(1) Professor Andras Farago's Lecture Notes
(2) www.wikipedia.com

➢ Appendix:

[Source Code:]

ExperimentAlgorithms.java

```
package com.bhakti.project3;

import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

public class ExperimentAlgorithms
{
	/*Given at least 5 examples; so lets take 6 examples here*/
	private static int numberOfExamples = 6;

	/*Take n random points in the plane; n >= 15; lets take n = 18 here*/
	private static int n = 18;
	public static int x,y;
	public static long Algo1Time1, Algo1Time2, Algo2Time1, Algo2Time2;
	public static IndexMinPQ<Double> pqMin;
	public static void main(String[] args)
	{
		for(int ex=0;ex<numberOfExamples;ex++)
```

```
            {
                  System.out.println("Experiment Number: "+(ex+1)+" starts here:");
                  HashMap<Integer, CoordinatePoints> points = new HashMap<Integer,
CoordinatePoints>();
                  NumberFormat nf = NumberFormat.getInstance();

                  /*Creates n number of points on a plane by generating x and y coordinates from
0-100*/
                  for(int i=0;i<=n;i++)
                  {
                        x = (int)(Math.random()*(100));
                        y = (int)(Math.random()*(100));
                        points.put(i, new CoordinatePoints(x, y));
                        //System.out.println("points: "+points.get(i));
                  }

                  for(Map.Entry<Integer, CoordinatePoints> entry: points.entrySet())
                  {
                        int key = entry.getKey();
                        CoordinatePoints value = entry.getValue();
                        //System.out.println("Point number "+(key)+" is:
"+"("+(value.x)+","+(value.y)+")");
                  }

                  /*Running the Local Search Heuristic Algorithm*/
                  Algo1Time1 = System.currentTimeMillis();
                  Algorithm2_LocalSearch.init(n, points);
                  Algo1Time2 = System.currentTimeMillis();
                  System.out.println("The running time of Local Search Heuristic Algorithm:
"+(Algo1Time2-Algo1Time1)+" ms");

                  //run Constructive Heuristic Algorithm
                  Algo2Time1 = System.currentTimeMillis();
                  Algorithm1_Constructive.init(n, points);
                  Algo2Time2 = System.currentTimeMillis();
                  System.out.println("The running time of Constructive Heuristic Algorithm:
"+(Algo2Time2-Algo2Time1)+" ms");

                  System.out.println("Experiment Number: "+(ex+1)+" ends here!");
            }
      }

}
```

Algorithm1_Constructive.java

```
package com.bhakti.project3;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import com.bhakti.project3.*;

public class Algorithm1_Constructive
{
      public static void init(int n, HashMap<Integer, CoordinatePoints> points)
      {
            int[][] adjmatrix = new int[n][n];
```

```
		List<NodeEdge> edge = new ArrayList<NodeEdge>();

		/*Generates an empty graph for the components*/
		GenerateGraph graph = GraphProperties.generateEmptyGraph(n ,points);
		graph.setAdjMatrix(adjmatrix);

		/*Generates edges for points i & j*/
		for(int i=0;i<n;i++)
		{
			for(int j=0;j<n;j++)
			{
				if(i<j)
				{
					edge.add(new NodeEdge(i, j, GraphProperties.getEdgeWeight(i, j, 
points)));
				}
			}
		}

		/*Sort the edge cost in ascending order*/
		Collections.sort(edge);

		/*Check the conditions (1),(2) and (3) of pdf i.e all the nodes are connected, degree 
is at least 3 and diameter is at most 4*/
		boolean isNodeConnected = new CheckConnectedNodes(graph).isNodeConnected();
		boolean isAtLeastDegreeThree = GraphProperties.isDegreeThree(graph, 3);
		boolean isAtMostDiameterFour = GraphProperties.isDiameterFour(graph, 4);
		boolean allThreeConditionsSatisfied = isNodeConnected && isAtLeastDegreeThree && 
isAtMostDiameterFour;
		double currCost = Double.POSITIVE_INFINITY;
		System.out.println("Constructive Current Cost [fully disconnected]: " + currCost);

		for (NodeEdge edg : edge)
		{
			int i = edg.getNodei();
			int j = edg.getNodej();
			graph.getAdjMatrix()[i][j] = 1;
			graph.getAdjMatrix()[j][i] = 1;

			isNodeConnected = new CheckConnectedNodes(graph).isNodeConnected();
			isAtLeastDegreeThree = GraphProperties.isDegreeThree(graph, 3);
			isAtMostDiameterFour = GraphProperties.isDiameterFour(graph, 4);
			allThreeConditionsSatisfied = isNodeConnected && isAtLeastDegreeThree && 
isAtMostDiameterFour;

			//System.out.println("isNodeConnected: "+isNodeConnected);
			//System.out.println("isAtLeastDegreeThree: "+isAtLeastDegreeThree);
			//System.out.println("isAtMostDiameterFour: "+isAtMostDiameterFour);
			//System.out.println("allThreeConditionsSatisfied: 
"+allThreeConditionsSatisfied);

		if(allThreeConditionsSatisfied)
		{
			currCost = GraphProperties.getCost(graph);
			System.out.println("Constructive Current Cost [max] -> " + currCost);
			break;
		}
		}

		/*Track and save the edges that cannot be removed*/
		HashSet<String> unremovableEdges = new HashSet<String>();
		while(true)
		{
			/*Find the edge that has the maximum weight else break*/
			List<NodeEdge> edg = graph.edges();
			double wt = 0.0;
			NodeEdge heaviestEdge = null;
			for (NodeEdge ed : edg)
			{
				if(wt < ed.getWeight() && !unremovableEdges.contains(ed.getNodei() + ""
+ ed.getNodej()) && !unremovableEdges.contains(ed.getNodej() + "" + ed.getNodei()))
				{
					wt = ed.getWeight();
					heaviestEdge = ed;
				}
			}

			if(heaviestEdge==null)
			{
				break;
			}
			graph.getAdjMatrix()[heaviestEdge.getNodei()][heaviestEdge.getNodej()] = 0;
			graph.getAdjMatrix()[heaviestEdge.getNodej()][heaviestEdge.getNodei()] = 0;

			/*Check if all the 3 conditions satisfy*/
			isNodeConnected = new CheckConnectedNodes(graph).isNodeConnected();
		isAtLeastDegreeThree = GraphProperties.isDegreeThree(graph, 3);
		isAtMostDiameterFour = GraphProperties.isDiameterFour(graph, 4);
		allThreeConditionsSatisfied = isNodeConnected && isAtLeastDegreeThree &&
isAtMostDiameterFour;

		if(allThreeConditionsSatisfied)
		{
			//System.out.println("in here");
			currCost = GraphProperties.getCost(graph);
			System.out.println("Constructive Current Cost -> " + currCost);
		}
		else
		{
			graph.getAdjMatrix()[heaviestEdge.getNodei()][heaviestEdge.getNodej()] = 1;
			graph.getAdjMatrix()[heaviestEdge.getNodej()][heaviestEdge.getNodei()] = 1;
			unremovableEdges.add(heaviestEdge.getNodei() + "" + heaviestEdge.getNodej());
			unremovableEdges.add(heaviestEdge.getNodej() + "" + heaviestEdge.getNodei());
		}
		}

	}
}
```

Algorithm2_LocalSearch.java:

```
package com.bhakti.project3;

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import com.bhakti.project3.*;

public class Algorithm2_LocalSearch
{
	public static void init(int n, HashMap<Integer, CoordinatePoints> points)
	{
		/*Generate a fully connected graph and check if it satisfies all the 3 conditions*/
		GenerateGraph graph = GraphProperties.generateFullyConnectedGraph(n, points);

		boolean isNodeConnected = new CheckConnectedNodes(graph).isNodeConnected();
		boolean isAtLeastDegreeThree = GraphProperties.isDegreeThree(graph, 3);
		boolean isAtMostDiameterFour = GraphProperties.isDiameterFour(graph, 4);
		boolean allThreeConditionsSatisfied = isNodeConnected && isAtLeastDegreeThree &&
isAtMostDiameterFour;

		HashSet<String> unremovableEdges = new HashSet<String>();
		double currentCost = GraphProperties.getCost(graph);
		System.out.println("Local Search Current Cost [fully connected] -> " + currentCost);

		while(true)
		{
			/*Finds an edge with the largest weight and remove it else break*/
			List<NodeEdge> edges = graph.edges();
			double weight = 0.0;
			NodeEdge heaviestEdge = null;
			for (NodeEdge edge : edges)
			{
				if(weight < edge.getWeight() &&
!unremovableEdges.contains(edge.getNodei() + "" + edge.getNodej()) &&
!unremovableEdges.contains(edge.getNodej() + "" + edge.getNodei()))
				{
					weight = edge.getWeight();
					heaviestEdge = edge;
				}
			}

			if(heaviestEdge==null)
			{
				break;
			}

			graph.getAdjMatrix()[heaviestEdge.getNodei()][heaviestEdge.getNodej()] = 0;
			graph.getAdjMatrix()[heaviestEdge.getNodej()][heaviestEdge.getNodei()] = 0;

			isNodeConnected = new CheckConnectedNodes(graph).isNodeConnected();
			isAtLeastDegreeThree = GraphProperties.isDegreeThree(graph, 3);
			isAtMostDiameterFour = GraphProperties.isDiameterFour(graph, 4);
			allThreeConditionsSatisfied = isNodeConnected && isAtLeastDegreeThree &&
isAtMostDiameterFour;

			if(allThreeConditionsSatisfied)
			{
			currentCost = GraphProperties.getCost(graph);
			System.out.println("Local Search Current Cost -> " + currentCost);
		}
		else
		{
			graph.getAdjMatrix()[heaviestEdge.getNodei()][heaviestEdge.getNodej()] = 1;
			graph.getAdjMatrix()[heaviestEdge.getNodej()][heaviestEdge.getNodei()] = 1;
			unremovableEdges.add(heaviestEdge.getNodei() + "" + heaviestEdge.getNodej());
			unremovableEdges.add(heaviestEdge.getNodej() + "" + heaviestEdge.getNodei());
		}
		}
	}
```

```
}
```

CheckConnectedNodes.java:

```
package com.bhakti.project3;

public class CheckConnectedNodes
{
	private boolean[] markVertex;
	private int[] id;
	private int[] size;
	private int countConnections;

	/*Checks a vertex is connected to how many other vertices*/
	public CheckConnectedNodes(GenerateGraph g)
	{
		markVertex = new boolean[g.nVertices()];
		id = new int[g.nVertices()];
		size = new int[g.nVertices()];
		for (int i = 0; i < g.nVertices(); i++)
		{
			if (!markVertex[i])
			{
				DFS(g, i);
				countConnections++;
			}
		}
	}

	/*Uses DFS to find the connections*/
	public void DFS(GenerateGraph g, int i)
	{
		markVertex[i] = true;
		id[i] = countConnections;
		size[countConnections]++;
		for (int j : g.adjacent(i))
		{
			if (!markVertex[j])
			{
				DFS(g, j);
			}
		}
	}

	/*Checks if all the nodes are connected in the graph*/
	public boolean isNodeConnected()
	{
		if (countConnections == 1)
		{
			return true;
		}
		else
		{
			return false;
		}
	}

	public int id(int i)
	{
		return id[i];
	}

	public int size(int i)
	{
		return size[id[i]];
	}

	public int countConnections()
	{
		return countConnections;
	}

}
```

CalculateDijkstraShortestPath.java:

```
package com.bhakti.project3;

import java.util.Vector;
import java.util.stream.Collectors;

public class CalculateDijkstraShortestPath
{
	private NodeEdge[] edgeBtwn;
	private double[] distBtwn;
	private IndexMinPQ<Double> pqMin;
	public int count=0;

	public double distBtwn(int i)
	{
		return distBtwn[i];
	}

	/*Checks the path between edges*/
	public boolean hasPathTo(int i)
	{
		return distBtwn[i] < Double.POSITIVE_INFINITY;
	}

	/*Finds the Dijkstra's shortest path*/
	public CalculateDijkstraShortestPath(GenerateGraph gg, int dt)
	{
		count++;
		distBtwn = new double[gg.nVertices()];
		edgeBtwn = new NodeEdge[gg.nVertices()];

		for (int i=0;i<distBtwn.length;i++)
		{
			distBtwn[i]=Double.POSITIVE_INFINITY;
		}

		distBtwn[dt] = 0.0;
		pqMin = new IndexMinPQ<Double>(gg.nVertices());
		pqMin.insert(dt, distBtwn[dt]);

		while (!pqMin.isEmpty())
		{
			int i = pqMin.delMin();
```

```
                    //System.out.println("Adjacent edges of" + i + " are: " +
gg.adjacentEdges(i).stream().map(NodeEdge::getNodej).collect(Collectors.toList()));

                    for (NodeEdge e : gg.adjacentEdges(i))
                    {
                            relaxAndUpdate(e, i);
                    }
            }
    }

    /*Relax an edge and updates its corresponding vertex*/
    private void relaxAndUpdate(NodeEdge e, int i)
    {
            int j = e.other(i);
            if (distBtwn[j] > distBtwn[i] + e.getWeight())
            {
                    distBtwn[j] = distBtwn[i] + e.getWeight();
                    edgeBtwn[j] = e;
                    if (pqMin.contains(j))
                    {
                            pqMin.decreaseKey(j, distBtwn[j]);
                    }
                    else
                    {
                            pqMin.insert(j, distBtwn[j]);
                    }
            }
    }

}
```

CoordinatePoints.java:

```
package com.bhakti.project3;

//Creates a point with (x,y) coordinates.
public class CoordinatePoints
{
        public int x;
        public int y;

        public CoordinatePoints(int x, int y)
        {
                this.x = x;
                this.y = y;
        }

        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;
	}

}
```

GenerateGraph.java:

```
package com.bhakti.project3;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.stream.Collectors;

import com.bhakti.project3.*;

public class GenerateGraph
{
	private int nVertices;
	private int[][] adjMatrix;
	private HashMap<Integer, CoordinatePoints> points;

	/*Generates a graph of n vertices*/
	public GenerateGraph(int nVertices, HashMap<Integer, CoordinatePoints> points)
	{
		this.nVertices = nVertices;
		this.adjMatrix = new int[nVertices][nVertices];
		this.points = points;
	}

	public int nVertices()
	{
		return nVertices;
	}

	public void setAdjMatrix(int[][] adj)
	{
		this.adjMatrix = adj;
	}

	public int[][] getAdjMatrix()
	{
		return adjMatrix;
	}

	public HashMap<Integer, CoordinatePoints> getPoints()
	{
		return this.points;
	}

	/*Get all the edges of the graph*/
	public List<NodeEdge> edges()
	{
		List<NodeEdge> edges = new ArrayList<NodeEdge>();
		for(int i=0;i<nVertices;i++)
		{
			for(int j=0;j<nVertices;j++)
			{
				if(i<j && adjMatrix[i][j]==1)
				{
					edges.add(new NodeEdge(i, j, getWt(i, j)));
				}
			}
		}
		return edges;
	}

	/*Gets all the adjacent edges*/
	public List<NodeEdge> adjacentEdges(int i)
	{
		List<NodeEdge> edges = new ArrayList<NodeEdge>();
		for(int j=0;j<nVertices;j++)
		{
			if(adjMatrix[i][j]==1)
			{
				edges.add(new NodeEdge(i, j, getWt(i, j)));
			}
		}
		//System.out.println("edges:
"+edges.stream().map(NodeEdge::getNodei).collect(Collectors.toList()));
		return edges;
	}

	/*Returns edge weight*/
	private Double getWt(int i, int j)
	{
		if(this.points==null)
			return 1.0;
		CoordinatePoints p1 = points.get(i);
		CoordinatePoints p2 = points.get(j);
		double x1 = p1.getX();
		double x2 = p2.getX();
		double y1 = p1.getY();
		double y2 = p2.getY();
		return Math.sqrt(Math.pow(x1-x2, 2) + Math.pow(y1-y2, 2));
	}

	/*Returns vertex degree*/
	public int degree(int i)
	{
		int sum=0;
		for(int j=0;j<nVertices;j++)
		{
			sum=sum+adjMatrix[i][j];
		}
		return sum;
	}

	/*Returns the adjacent vertex*/
	public List<Integer> adjacent(int i)
	{
		List<Integer> list = new ArrayList<Integer>();
		for(int j=0;j<this.nVertices;j++)
		{
			if(adjMatrix[i][j]==1)
			{
					list.add(j);
				}
			}
			return list;
		}
}
```

GraphProperties.java:

```
package com.bhakti.project3;
import java.util.HashMap;
import com.bhakti.project3.*;

public class GraphProperties
{
	/*Checks if the diameter of the graph is at most 4*/

	public static boolean isDiameterFour(GenerateGraph graph, int hops)
	{
		GenerateGraph graphWithEqualDist = new GenerateGraph(graph.nVertices(), null);
		int[][] adj = new int[graph.nVertices()][graph.nVertices()];
		for(int i=0;i<graph.nVertices();i++)
		{
			for(int j=0;j<graph.nVertices();j++)
			{
				if(graph.getAdjMatrix()[i][j]==1)
					{
						adj[i][j] = 1;
					}
			}
		}
		graphWithEqualDist.setAdjMatrix(adj);

		for (int i = 0; i < graphWithEqualDist.nVertices(); i++)
		{
			CalculateDijkstraShortestPath sp = new
CalculateDijkstraShortestPath(graphWithEqualDist, i);
			for (int j = 0; j < graphWithEqualDist.nVertices(); j++)
			{
				double d = sp.distBtwn(j);
				//System.out.println("Distance between " + i + " & " + j + ": " + d);
				if (d > 4.00)
				{
					return false;
				}
			}
		}
		return true;
	}


	/*Checks if each node is connected to at least 3 other nodes*/
	public static boolean isDegreeThree(GenerateGraph g, int n)
	{
		for (int i=0;i<g.nVertices();i++)
		{
			if (g.degree(i)<3)
			{
				return false;
			}
		}
		return true;
	}

	/*Generates a connected graph for the given nodes and points*/
	public static GenerateGraph generateFullyConnectedGraph(int n, HashMap<Integer, 
CoordinatePoints> points)
	{
		GenerateGraph graph = new GenerateGraph(n, points);
		int[][] adj = new int[n][n];
		for(int i=0;i<adj.length;i++)
		{
			for(int j=0;j<adj.length;j++)
			{
				if(i!=j)
				{
					adj[i][j] = 1;
				}
			}
		}
		graph.setAdjMatrix(adj);
		return graph;
	}

	/*Gives the cost of the graph*/
	public static double getCost(GenerateGraph graph)
	{
		double wt = 0.0;
		for(NodeEdge e: graph.edges())
		{
			wt = wt + e.getWeight();
		}
		return wt;
	}

	/*Generates an empty graph for the given nodes and points*/
	public static GenerateGraph generateEmptyGraph(int n, HashMap<Integer, CoordinatePoints> 
points)
	{
		GenerateGraph graph = new GenerateGraph(n, points);
		int[][] adj = new int[n][n];
		graph.setAdjMatrix(adj);
		return graph;
	}

	/*Get the edge weight between points*/
	public static Double getEdgeWeight(int i, int j, HashMap<Integer, CoordinatePoints> points)
	{
		CoordinatePoints p1 = points.get(i);
		CoordinatePoints p2 = points.get(j);
		double x1 = p1.getX();
		double x2 = p2.getX();
		double y1 = p1.getY();
		double y2 = p2.getY();
		return Math.sqrt(Math.pow(x1-x2, 2) + Math.pow(y1-y2, 2));
	}
}
```

NodeEdge.java:

```
package com.bhakti.project3;

//Creates an edge between two nodes
public class NodeEdge implements Comparable<NodeEdge>
{
	private int nodei;
	private int nodej;
	private double weight;

	public NodeEdge(int i, int j, double wt)
	{
		this.nodei = i;
		this.nodej = j;
		this.weight = wt;
	}

	public void setNodei(int i)
	{
		this.nodei = i;
	}

	public void setNodej(int j)
	{
		this.nodej = j;
	}

	public void setWeight(double wt)
	{
		this.weight = wt;
	}

	public int getNodei()
	{
		return nodei;
	}

	public int getNodej()
	{
		return nodej;
	}

	public double getWeight()
	{
		return weight;
	}

	public int other(int i)
	{
		if(i==this.nodei)
		{
			return this.nodej;
		}
		else if(i==this.nodej)
		{
			return this.nodei;
		}
		else
		{
```

```
                return -1;
            }
        }

        //Compares the edge weight of two nodes
        @Override
        public int compareTo(NodeEdge e)
        {
            if(this.getWeight() > e.getWeight())
            {
                return +1;
            }
            else if(this.getWeight() == e.getWeight())
            {
                return 0;
            }
            else
            {
                return -1;
            }
        }

}
```

IndexMinPQ.java:

```
package com.bhakti.project3;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IndexMinPQ<Key extends Comparable<Key>> implements Iterable<Integer>
{
        private int maxN;
        private int n;
        private int[] pq;
        private int[] qp;
        private Key[] keys;

        /*Initializes an empty indexed priority queue*/
        public IndexMinPQ(int maxN)
        {
            if (maxN < 0)
                throw new IllegalArgumentException();
            this.maxN = maxN;
            n = 0;
            keys = (Key[]) new Comparable[maxN + 1]; // make this of length maxN??
            pq = new int[maxN + 1];
            qp = new int[maxN + 1]; // make this of length maxN??
            for (int i = 0; i <= maxN; i++)
                qp[i] = -1;
        }

        /*Returns true if this priority queue is empty*/
        public boolean isEmpty()
        {
            return n == 0;
        }

        /*Compares index of the priority queue*/
```

```java
public boolean contains(int i)
{
	if (i < 0 || i >= maxN)
		throw new IndexOutOfBoundsException();
	return qp[i] != -1;
}

/*Gives the number of keys of this priority queue */
public int size()
{
	return n;
}

/*Checks the item linked with the index*/
public void insert(int i, Key key)
{
	if (i < 0 || i >= maxN)
		throw new IndexOutOfBoundsException();
	if (contains(i))
		throw new IllegalArgumentException("index is already in the priority queue");
	n++;
	qp[i] = n;
	pq[n] = i;
	keys[i] = key;
	swim(n);
}

/*Gives the item with minimum element*/
public int minIndex()
{
	if (n == 0)
		throw new NoSuchElementException("Priority queue underflow");
	return pq[1];
}

/*Gives the minimum key*/
public Key minKey()
{
	if (n == 0)
		throw new NoSuchElementException("Priority queue underflow");
	return keys[pq[1]];
}

/*Removes a minimum key and returns its associated index*/
public int delMin()
{
	if (n == 0)
		throw new NoSuchElementException("Priority queue underflow");
	int min = pq[1];
	exch(1, n--);
	sink(1);
	assert min == pq[n + 1];
	qp[min] = -1; // delete
	keys[min] = null; // to help with garbage collection
	pq[n + 1] = -1; // not needed
	return min;
}

/*Returns the key associated with index*/
public Key keyOf(int i)
{
		if (i < 0 || i >= maxN)
			throw new IndexOutOfBoundsException();
		if (!contains(i))
			throw new NoSuchElementException("index is not in the priority queue");
		else
			return keys[i];
	}

	/*Change the key associated with index i to the specified value*/
	public void changeKey(int i, Key key)
	{
		if (i < 0 || i >= maxN)
			throw new IndexOutOfBoundsException();
		if (!contains(i))
			throw new NoSuchElementException("index is not in the priority queue");
		keys[i] = key;
		swim(qp[i]);
		sink(qp[i]);
	}

	/*Change the key associated with index i to the specified value*/
	public void change(int i, Key key)
	{
		changeKey(i, key);
	}

	/*Decrease the key associated with index i to the specified value*/
	public void decreaseKey(int i, Key key)
	{
		if (i < 0 || i >= maxN)
			throw new IndexOutOfBoundsException();
		if (!contains(i))
			throw new NoSuchElementException("index is not in the priority queue");
		if (keys[i].compareTo(key) <= 0)
			throw new IllegalArgumentException("Calling decreaseKey() with given argument 
would not strictly decrease the key");
		keys[i] = key;
		swim(qp[i]);
	}

	/*Increase the key associated with index i to the specified value*/
	public void increaseKey(int i, Key key)
	{
		if (i < 0 || i >= maxN)
			throw new IndexOutOfBoundsException();
		if (!contains(i))
			throw new NoSuchElementException("index is not in the priority queue");
		if (keys[i].compareTo(key) >= 0)
			throw new IllegalArgumentException(
					"Calling increaseKey() with given argument would not strictly 
increase the key");
		keys[i] = key;
		sink(qp[i]);
	}

	/*Remove the key associated with index*/
	public void delete(int i)
	{
		if (i < 0 || i >= maxN)
			throw new IndexOutOfBoundsException();
		if (!contains(i))
```

```
                throw new NoSuchElementException("index is not in the priority queue");
            int index = qp[i];
            exch(index, n--);
            swim(index);
            sink(index);
            keys[i] = null;
            qp[i] = -1;
    }

    private boolean greater(int i, int j)
    {
            return keys[pq[i]].compareTo(keys[pq[j]]) > 0;
    }

    private void exch(int i, int j)
    {
            int swap = pq[i];
            pq[i] = pq[j];
            pq[j] = swap;
            qp[pq[i]] = i;
            qp[pq[j]] = j;
    }

    private void swim(int k)
    {
            while (k > 1 && greater(k / 2, k))
            {
                    exch(k, k / 2);
                    k = k / 2;
            }
    }

    private void sink(int k)
    {
            while (2 * k <= n)
            {
                    int j = 2 * k;
                    if (j < n && greater(j, j + 1))
                            j++;
                    if (!greater(k, j))
                            break;
                    exch(k, j);
                    k = j;
            }
    }

    public Iterator<Integer> iterator()
    {
            return new HeapIterator();
    }

    private class HeapIterator implements Iterator<Integer>
    {
            private IndexMinPQ<Key> copy;
            public HeapIterator()
            {
                    copy = new IndexMinPQ<Key>(pq.length - 1);
                    for (int i = 1; i <= n; i++)
                            copy.insert(pq[i], keys[pq[i]]);
            }
```

```
		public boolean hasNext()
		{
			return !copy.isEmpty();
		}

		public void remove()
		{
			throw new UnsupportedOperationException();
		}

		public Integer next()
		{
			if (!hasNext())
				throw new NoSuchElementException();
			return copy.delMin();
		}
	}
}
```