Project 3 Report

Analysis

Input:

locations of n nodes with their coordinates

Job:

create a network topology(undirected graph) with following properties:

1. Contains all nodes
2. Degree of each vertex is at least 3
3. Diameter of the graph is at most 4(hop-distance)
4. Total cost of the network topology is as low as possible by the total geometric length of all links

Goal:

Implement two different heuristic algorithm(it does not have to guarantee the exact optimum)

E.g: Branch and Bound, Simulated Annealing, Greedy Local Search, Tabu Search, Genetic Algorithm

Creative ideas will be appreciated.

Two algorithms should be sufficiently different to compete in finding good solution.

Tasks:

- Describe the two algorithms.
- Provide reference to the source
- Provide pseudo code with sufficient comments
- Run the program on randomly generated examples(at least 5 examples), pick n random points in the plane, this can be done by generating random numbers in some range and taking them as coordinates, n >= 15
- show result(nodes` position) graphically.
- Draw some conclusion about how the two algorithms compare

Random Nodes Generating

The project asks to given the location of n nodes in the plane by their coordinates. Assume there will be 15 nodes with random coordinates be created.

Assume coordinates are in the range from 0 to 9, we could generate random nodes like: n1(0,0), n2(9,0)**...**., but there should not have duplicate coordinates thus duplication prevention should be considered.As figure 1:

And show them in the graph:

Hashset is used to prevent coordinate duplication of each node, whenever the generated coordinates are unique, then we create new node with the coordinates, and added to node array.

```
public void setRandomCoordinates(Node[] nodes){
    //x and y
    //from 0 to 9
    System.out.println("Nodes added: ");
    HashSet<Node> temp = new HashSet<Node>();
    for(int i = 0 ; i < nodes.length; i ++){
        int x = (int)(Math.random() * 10 );
        int y = (int)(Math.random() * 10 );
        while(!temp.add(new Node(x,y))){
            x = (int)(Math.random() * 10 );
            y = (int)(Math.random() * 10 );
            System.out.println("Not add");
        }
        nodes[i] = new Node(i,x,y);
        System.out.println("" + x + " " + y);
    }
```

Figure 1

Swing and awt libraries are used to create coordinate system, and draw nodes.

```
Nodes added to ShortestPathGraph: [[(6,9), (3,9), (0,6), (7,8), (1,0), (7,9), (4,4), (8,5), (8,4), (5,0), (4,9), (8,0), (2,7), (5,6), (9,0)]]
```

Figure 1.1

```
import javax.swing.*;
import java.awt.*;
```

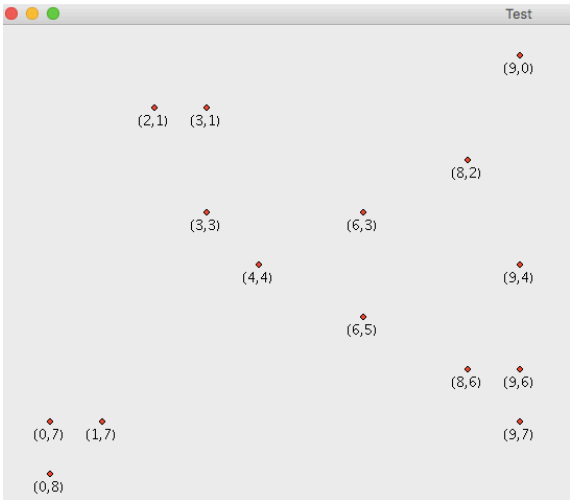


Figure 2

Figure 2 is one random set of nodes created.

Heuristic Algorithms

What we need to do is to create a network topology that:

1. It will contains all the generated nodes(the graph is complete connected)
2. The diameter of the graph is at most 4(refers to the hop-distance).
3. Total cost of the network topology is as low as possible(measured by geometric length).

Branch and Bound Algorithm

B : set of all n-dimensional binary vectors(all combinations), here it represent if one node has link to each other nodes.

f(x): maximum value over B, which represent as the connection of each node leads to the final total geometric length of all links.

$$\max_{x \in B} f(x)$$

$B_k(b)$: subset of B which fixed the first k coordinates, which means the connections of first k nodes has been decided. when k = n, we fixed all the links.

Pseudo Code

Since the project requires every node`s degree no less than 3, if one node has no less than 3 outgoing edges, then we consider this node as fixed. The algorithm here is to pick every node and add three of its least cost outgoing edges into the graph, then mark the node as fixed, then step into next unfixed node to satify its degree requirement. After each node has no less than 3 degree, we then check the graph`s diameter, if it is more than 4, then we add another least cost edge into the graph.

```
for(each node in the graph){
        pick three least cost potential outgoing edges added to graph;
}
for(each node in the graph){
        check the diameter of graph;
        if(diameter > 4){
                pick another least cost edge added into the graph;
                if(diameter still > 4){
                        add another least cost edge;
                }
        }
}
```

Implementation

```
public void pick(){
    List<Node> temp = new ArrayList<Node>(this.ShortestPathGraph.getVertices());
    Collections.shuffle(temp);
    for(Node node1: temp/*this.ShortestPathGraph.getVertices()*/){
        LinkedHashMap<Float,Node> stack = new LinkedHashMap<Float,Node>();
        System.out.println("==========For Node : " + node1 + "============== degree" + node1.getDegree() );
        float threshold = 0;
        if(node1.getDegree() < 3){
            for(Node node2: this.ShortestPathGraph.getVertices()){
                if( node1 != node2){
                    float distance = this.getGeometricDistance(node1, node2);
                    if(node1.getDegree() < 3){
                        stack.put(distance, node2);
                        Edge edge = new Edge(node1,stack.get(distance),distance);
                        this.ShortestPathGraph.addEdge(edge, node1, stack.get(distance));
                        if(!checkDiameter(node1)){          //check diameter , from node1 to all others
                            //this.ShortestPathGraph.removeEdge(edge);
                            System.out.println("Edge adding will lead to diameter > 4, cancel adding action");
                            this.ShortestPathGraph.removeEdge(edge);
                            stack.remove(distance);
                            continue;
                        }
                        this.ShortestPathGraph.removeEdge(edge);
                        if(distance > threshold){
                            threshold = distance;
                        }
                        node1.increaseDegree();
                        node2.increaseDegree();
                    }
                    else if(distance < threshold){
                        //stack.get(threshold).dercreaseDegree();
                        Edge edge = new Edge(node1,node2,distance);
                        this.ShortestPathGraph.addEdge(edge, node1, node2);
                        if(!checkDiameter(node1)){          //check diameter , from node1 to all others
                            //this.ShortestPathGraph.removeEdge(edge);
                            System.out.println("Edge adding will lead to diameter > 4, cancel adding action");
                            this.ShortestPathGraph.removeEdge(edge);
                            continue;
                        }
                        this.ShortestPathGraph.removeEdge(edge);
                        stack.remove(threshold);
                        stack.put(distance, node2);
                        threshold = distance;
                    }
                }
            }
        }
```

Figure 3

First, we assume all nodes are disconnected, each time we pick one unfixed node as starting point, because degree of each vertex in the graph is at least 3, thus each node should has at least 3 outgoing edges.

For each starting point, we pick 3 outgoing links randomly, then compare these 3 links with all other potential links to get 3 links with smallest geometric distance, getGeometricDistance() method will calculate geometric distance between two nodes as Figure 4:

```
private float getGeometricDistance(Node one, Node two){
    return (float) Math.sqrt( Math.pow(one.getX() - two.getX(), 2) + Math.pow(one.getY() - two.getY(), 2) );
}
```

Figure 4

Before we add these 3 links to the final graph, check if graph`s diameter will exceed 4 as Figure 5, if yes, don`t add that link.

```
private boolean checkDiameter(Node one){
    this.DSP = new DijkstraShortestPath<Node,Edge>(this.ShortestPathGraph);
    for(Node node: this.ShortestPathGraph.getVertices()){
        if(node != one){
            List<Edge> tempList = this.DSP.getPath(one, node);
            if(tempList.size() > 4){
                return false;
            }
        }
    }
    return true;
}
```

Figure 5

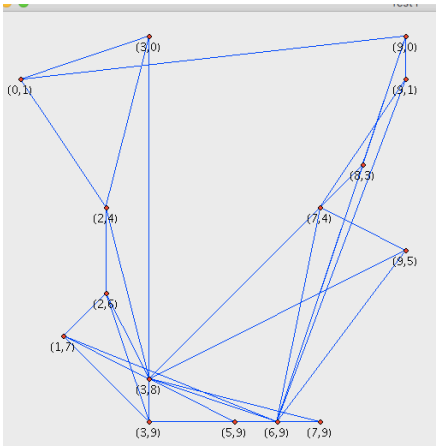
5 Sample Runs

Nodes added to ShortestPathGraph: [[(0,1), (6,9), (3,8), (3,9), (7,9), (7,4), (1,7), (8,3), (5,9), (2,4), (9,5), (2,6), (9,1), (3,0), (9,0)]]

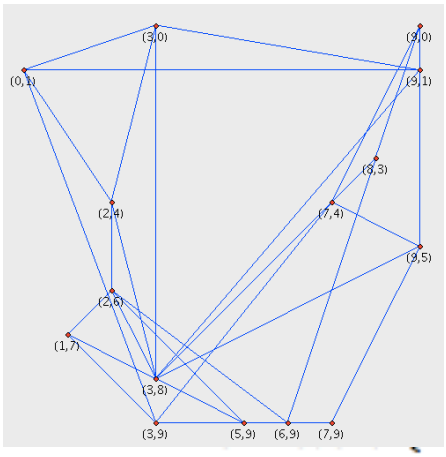
In order to eliminate the influence of the order of nodes, we are going to shuffle their order before running each test as Figure 5.1:

```
List<Node> temp = new ArrayList(this.ShortestPathGraph.getVertices());
Collections.shuffle(temp);
```

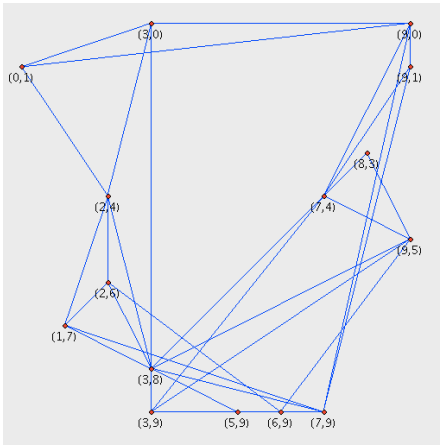
Figure 5.1



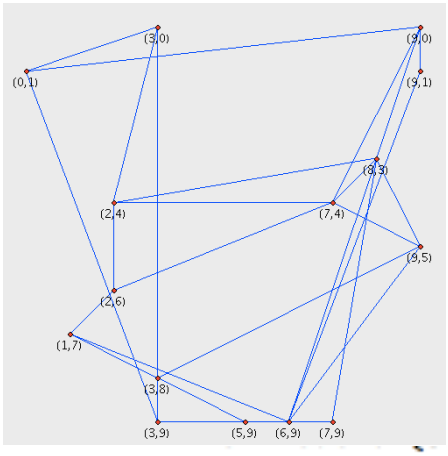
Total Cost: 136.99017



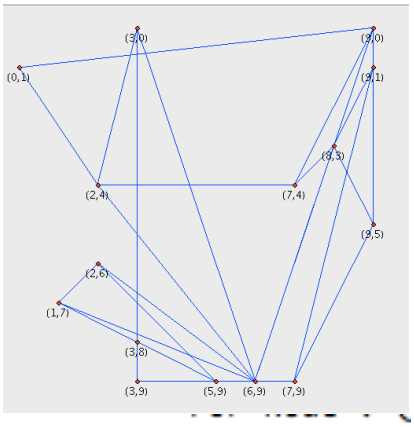
Total Cost: 139.13675



Total Cost: 137.21086



Total Cost: 129.21873



Total Cost: 122.07637

Greedy Local Search

The key to Greedy Local Search algorithm is to find current point`s best neighbor. If no improvement is possible in the neighborhood, then the algorithm stops.

Pseudo Code

For each round, we iterate through all nodes, each node will select its minimum cost edge(if the edge not been selected yet), after 3 round, each node will have a degree of 3, then we check if the graph` diameter is no more than 4, if yes, iterate one more round, each time add one more edge and check the diameter until it meets the requirement.

```
One Round(repeat 3 times):
    for(each node in the graph){
        if(the outgoing edge is not added to the graph && the edge is of minimum
        geometric cost){
            Add the edge to the graph;
            Add its weight to total geometric cost;
        }
```

}

After 3 Round:

for(each node in the graph){

DijkstraShortestPath(the node) finding the diameter;

if(diameter > 4){

Adding one more edge with minimum cost(starting from this node) into the graph;

}

recheck the node;

}

Implementation

Method nextRound() is going to start one round of iteration to add one more edge to each node with minimum cost. Method getGeomerticDistance() will calculate geometric distance between two nodes. Variables index and nextMin will track the minimum cost edge`s destination and its geomerticDistance. After inspect all potential edges, the minimum cost edge will be added into ShortestPathGraph as Figure 6:

```
public void nextRound(){
    for(int i = 0; i < nodes.length; i++){
        float nextMin = Float.MAX_VALUE;
        Node index = null;
        for(int j = i + 1; j < nodes.length; j++){
            float distance = this.getGeometricDistance(nodes[i],nodes[j]);
            if(distance <= nextMin && distance > nodes[i].getDistanceThreshold()){
                index = nodes[j];
                nextMin = distance;
            }
        }
        if(index != null){
            nodes[i].setDistanceThreshold(nextMin);
            Edge edge = new Edge(nodes[i],index,nextMin);
            this.ShortestPathGraph.addEdge(edge, nodes[i], index);
            this.totalGeometricCost += nextMin;
        }
    }
}
```

Figure 6

Method checkDiameter() is going to check the farthest hop from the specified node to all other nodes, the farthest hop will be no larger than the graph`s diameter.

```
private boolean checkDiameter(Node one){
    this.DSP = new DijkstraShortestPath<Node,Edge>(this.ShortestPathGraph);
    for(Node node: this.ShortestPathGraph.getVertices()){
        if(node != one){
            List<Edge> tempList = this.DSP.getPath(one, node);
            if(tempList.size() > 4){
                return false;
            }
        }
    }
    return true;
}
```

Figure 6.1

Method overallDiameter() is going to check the path distance of every node in the graph, to make sure that the diameter will not violate the requirement. That is diameter <= 4.

```
private void overallDiameter(){
    for(Node one : this.nodes){
        float min = Float.MAX_VALUE;
        Node temp = null;
        if(!checkDiameter(one)){
            for(Node two : nodes){
                if(one == two){
                    continue;
                }
                if(this.getGeometricDistance(one, two) < min){
                    Edge edge = new Edge(one,two,this.getGeometricDistance(one, two));
                    if(!this.ShortestPathGraph.containsEdge(edge)){
                        min = this.getGeometricDistance(one, two);
                        temp = two;
                    }
                }
            }
        }
        if(temp != null){
            Edge edge = new Edge(one,temp,this.getGeometricDistance(one, temp));
            this.ShortestPathGraph.addEdge(edge, one, temp);
            this.totalGeometricCost += this.getGeometricDistance(one, temp);
        }
    }
}
```

Figure 6.2

5 Sample Runs

First randomly generated 15 nodes with unique coordinates as follows:

```
[[(0,1), (6,9), (3,8), (3,9), (7,9), (7,4), (1,7), (8,3), (5,9), (2,4), (9,5), (2,6), (9,1), (3,0), (9,0)]]
```

In order to eliminate the influence of the order of nodes, we are going to shuffle their order before running each test as Figure 6.2.1:

```
List<Node> tempNodes = new ArrayList(Arrays.asList(nodes));
Collections.shuffle(tempNodes);
```

Figure 6.2.1

Make five runs of Greedy Local Search, Figure 6.3 is the final graph generated:

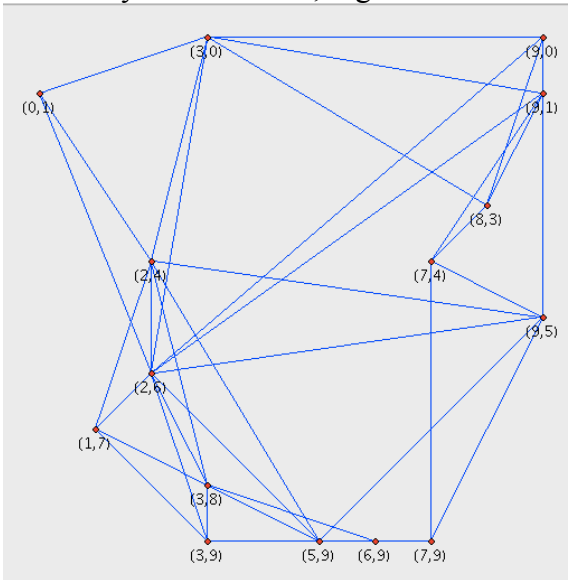


Figure 6.3

Compared to Branch and Bound Algorithm, the final cost and final graph created by Greedy Local Search is always the same as follows:

```
Branch and Bound Total Geometric Cost: 141.78938
Greedy Local Search Total GeometricCost: 155.65414
```

Sample 1

```
Branch and Bound Total Geometric Cost: 136.1207
Greedy Local Search Total GeometricCost: 155.65414
```

Sample 2

```
Branch and Bound Total Geometric Cost: 141.57227
Greedy Local Search Total GeometricCost: 155.65414
```

Sample 3

```
Branch and Bound Total Geometric Cost: 151.33531
Greedy Local Search Total GeometricCost: 155.65414
```

Sample 4

```
Branch and Bound Total Geometric Cost: 151.33531
Greedy Local Search Total GeometricCost: 155.65414
```

Sample 5

ReadMe

Project is written by Java
JUNG is used to draw graphs, download from : http://jung.sourceforge.net/
JFree is used to draw charts, download from: http://www.jfree.org/jfreechart/

Source Code

GreedyLocalSearch.java

```
package HeuristicAlgorithms;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import NetworkElements.Edge;
import NetworkElements.Node;
import NetworkElements.myGraph;
import
edu.uci.ics.jung.algorithms.shortestpath.DijkstraShortestPath;

/**
 *
 * @author Tibbers
 * For each round, we iterate through all nodes, each node select
its minimum cost edge(if not been selected), after 3 round, each
node
 * will have a degree of 3, then we check if the graph`s diameter
is no more than 4, if yes, iterate one more round, each time add
one
 * edge and check the diameter, until it meets the requirement
 */
public class greedyLocalSearch {
     private myGraph<Node,Edge> ShortestPathGraph;
     private DijkstraShortestPath<Node,Edge> DSP;
     private float totalGeometricCost;
     private List<Node> nodes;

     public greedyLocalSearch(Node[] nodes){
          this.ShortestPathGraph = new myGraph<Node,Edge>();
          this.nodes = Arrays.asList(nodes);
          this.nextRound();
          this.nextRound();
          this.nextRound();
          this.overallDiameter();
          System.out.println("Greedy Local Search Total
GeometricCost: " + this.totalGeometricCost);
```

```
	}
	public myGraph<Node,Edge> getShortestPathGraph(){
		return this.ShortestPathGraph;
	}
	/**
	 * run 3 rounds to see if all nodes has degree >= 3, and
the diameter <= 4
	 */
	public void nextRound(){
		List<Node> tempNodes = new
ArrayList(Arrays.asList(nodes));
		Collections.shuffle(tempNodes);
		for(int i = 0; i < nodes.size(); i++){
			float nextMin = Float.MAX_VALUE;
			Node index = null;
			for(int j = i + 1; j < nodes.size(); j++){
				float distance =
this.getGeometricDistance(nodes.get(i),nodes.get(j));
				if(distance <= nextMin && distance >
nodes.get(i).getDistanceThreshold()){
					index = nodes.get(j);
					nextMin = distance;
				}
			}
			if(index != null){
				nodes.get(i).setDistanceThreshold(nextMin);
				Edge edge = new
Edge(nodes.get(i),index,nextMin);
				this.ShortestPathGraph.addEdge(edge,
nodes.get(i), index);
				this.totalGeometricCost += nextMin;
			}
		}
	}

	private void overallDiameter(){
		for(Node one : this.nodes){
			float min = Float.MAX_VALUE;
			Node temp = null;
			if(!checkDiameter(one)){
				for(Node two : nodes){
					if(one == two){
						continue;
					}
```

```
                              if(this.getGeometricDistance(one,
two) < min){
                                   Edge edge = new
Edge(one,two,this.getGeometricDistance(one, two));

      if(!this.ShortestPathGraph.containsEdge(edge)){
                                        min =
this.getGeometricDistance(one, two);
                                        temp = two;
                                   }
                              }
                         }
                    }
                    if(temp != null){
                         Edge edge = new
Edge(one,temp,this.getGeometricDistance(one, temp));
                         this.ShortestPathGraph.addEdge(edge, one,
temp);
                         this.totalGeometricCost +=
this.getGeometricDistance(one, temp);
                    }
               }
      }

      /**
       * Check diameter start from node one, if diameter <= 4,
return true
       * @param one
       * @return
       */
      private boolean checkDiameter(Node one){
           this.DSP = new
DijkstraShortestPath<Node,Edge>(this.ShortestPathGraph);
           for(Node node: this.ShortestPathGraph.getVertices()){
                if(node != one){
                     List<Edge> tempList = this.DSP.getPath(one,
node);
                     if(tempList.size() > 4){
                          return false;
                     }
                }
           }
           return true;
      }
```

```
	private float getGeometricDistance(Node one, Node two){
		return (float) Math.sqrt( Math.pow(one.getX() -
two.getX(), 2) + Math.pow(one.getY() - two.getY(), 2) );
	}
}
```

branchAndBound.java

```
package HeuristicAlgorithms;

import NetworkElements.*;
import edu.uci.ics.jung.algorithms.shortestpath.DijkstraShortestPath;


import java.util.*;

public class branchAndBound {
	public myGraph<Node,Edge> ShortestPathGraph;
	DijkstraShortestPath<Node,Edge> DSP;
	float totalGeometricCost;

	/**
	 * for each node, pick its 3 least cost outgoing edges, added to graph, then set this
node as fixed, going to next node
	 */
	public void pick(){
		List<Node> temp = new
ArrayList<Node>(this.ShortestPathGraph.getVertices());
		Collections.shuffle(temp);
		for(Node node1: temp/*this.ShortestPathGraph.getVertices()*/){
			LinkedHashMap<Float,Node> stack = new
LinkedHashMap<Float,Node>();
			System.out.println("==========For Node : " + node1 +
"============== degree" + node1.getDegree() );
			float threshold = 0;
			if(node1.getDegree() < 3){
				for(Node node2: this.ShortestPathGraph.getVertices()){
					if( node1 != node2){
						float distance =
this.getGeometricDistance(node1, node2);
						if(node1.getDegree() < 3){
							stack.put(distance, node2);
							Edge edge = new
Edge(node1,stack.get(distance),distance);
```

```
                this.ShortestPathGraph.addEdge(edge, node1, stack.get(distance));
                                                        if(!checkDiameter(node1)){
                //check diameter , from node1 to all others

                //this.ShortestPathGraph.removeEdge(edge);
                                                                System.out.println("Edge
adding will lead to diameter > 4, cancel adding action");

                this.ShortestPathGraph.removeEdge(edge);
                                                                stack.remove(distance);
                                                                continue;
                                                        }

                this.ShortestPathGraph.removeEdge(edge);
                                                        if(distance > threshold){
                                                                threshold = distance;
                                                        }
                                                        node1.increaseDegree();
                                                        node2.increaseDegree();
                                                }
                                                else if(distance < threshold){

                //stack.get(threshold).dercreaseDegree();
                                                        Edge edge = new
Edge(node1,node2,distance);

                this.ShortestPathGraph.addEdge(edge, node1, node2);
                                                        if(!checkDiameter(node1)){
                //check diameter , from node1 to all others

                //this.ShortestPathGraph.removeEdge(edge);
                                                                System.out.println("Edge
adding will lead to diameter > 4, cancel adding action");

                this.ShortestPathGraph.removeEdge(edge);
                                                                continue;
                                                        }

                this.ShortestPathGraph.removeEdge(edge);
                                                        stack.remove(threshold);
                                                        stack.put(distance, node2);
                                                        threshold = distance;
                                                }
                                        }
                                }
```

```
                        }
                        //Create 3 minimun cost edges insert to graph
                        for(float key : stack.keySet()){
                                this.totalGeometricCost += key;
        //add to total cost
                                Edge edge = new Edge(node1,stack.get(key),key);
                                this.ShortestPathGraph.addEdge(edge, node1,
stack.get(key));
                        }

                }
                System.out.println("Branch and Bound Total Geometric Cost: " +
this.totalGeometricCost);
                //System.out.println("Edges : " +
Arrays.asList(this.ShortestPathGraph.getEdges()));
        }


        private void resetEdges(){
                for(Edge edge : this.ShortestPathGraph.getEdges()){
                        this.ShortestPathGraph.removeEdge(edge);
                }
        }
        /**
         * Check diameter start from node one, if diameter <= 4, return true
         * @param one
         * @return
         */
        private boolean checkDiameter(Node one){
                this.DSP = new
DijkstraShortestPath<Node,Edge>(this.ShortestPathGraph);
                for(Node node: this.ShortestPathGraph.getVertices()){
                        if(node != one){
                                List<Edge> tempList = this.DSP.getPath(one, node);
                                if(tempList.size() > 4){
                                        return false;
                                }
                        }
                }
                return true;
        }

        private float getGeometricDistance(Node one, Node two){
                return (float) Math.sqrt( Math.pow(one.getX() - two.getX(), 2) +
Math.pow(one.getY() - two.getY(), 2) );
        }
```

```
}
```

Test.java

```
package test;
import java.util.*;
import NetworkElements.*;
import Visual.drawGraph;
public class Test {
	drawGraph frame;

	public Test(){
	}
	/**
	 * Assign random coordinates for all nodes and avoid
duplicate
	 * @param nodes
	 * arrays for all nodes
	 */
	public void setRandomCoordinates(Node[] nodes){
		//x and y
		//from 0 to 9
		System.out.println("Nodes Created: ");
		HashSet<Node> temp = new HashSet<Node>();
		for(int i = 0 ; i < nodes.length; i ++){
			int x = (int)(Math.random() * 10 );
			int y = (int)(Math.random() * 10 );
			while(!temp.add(new Node(x,y))){
				x = (int)(Math.random() * 10 );
				y = (int)(Math.random() * 10 );
				System.out.println("Not add");
			}
			nodes[i] = new Node(i,x,y);
			System.out.println("" + x + " " + y);
		}
	}
	public Node[] setRandomCoordinates(){
		//x and y
		//from 0 to 9
		System.out.println("Nodes Created: ");
		Node[] nodes = new Node[15];
		HashSet<Node> temp = new HashSet<Node>();
		for(int i = 0 ; i < 15; i ++){
```

```
                int x = (int)(Math.random() * 10 );
                int y = (int)(Math.random() * 10 );
                while(!temp.add(new Node(x,y))){
                        x = (int)(Math.random() * 10 );
                        y = (int)(Math.random() * 10 );
                        System.out.println("Not add");
                }
                nodes[i] = new Node(i,x,y);
                System.out.println("" + x + " " + y);
            }
            return nodes;
        }
}
```

runTest.java

```
package test;

import java.util.*;

import HeuristicAlgorithms.*;
import NetworkElements.*;
import Visual.drawGraph;
import edu.uci.ics.jung.algorithms.*;
public class runTest {
        private Node[] nodes;
        private ArrayList<Edge> edges;
        //private myGraph<Node,Edge> ShortestPathGraph;
        branchAndBound BB;
        public runTest(){
                nodes = new Node[15];
                edges = new ArrayList();
                this.BB = new branchAndBound();
        }

        public static void main(String args[]){

                Test testBed = new Test();
                Node[] nodes = new Node[15];//testBed.setRandomCoordinates();
                //(0,1), (6,9), (3,8), (3,9), (7,9), (7,4), (1,7), (8,3), (5,9), (2,4), (9,5), (2,6),
(9,1), (9,0), (3,0)
                nodes[0] = new Node(0,1);
                nodes[1] = new Node(6,9);
                nodes[2] = new Node(3,8);
                nodes[3] = new Node(3,9);
```

```
                nodes[4] = new Node(7,9);
                nodes[5] = new Node(7,4);
                nodes[6] = new Node(1,7);
                nodes[7] = new Node(8,3);
                nodes[8] = new Node(5,9);
                nodes[9] = new Node(2,4);
                nodes[10] = new Node(9,5);
                nodes[11] = new Node(2,6);
                nodes[12] = new Node(9,1);
                nodes[13] = new Node(9,0);
                nodes[14] = new Node(3,0);

                runTest test1 = new runTest();
                runTest test2 = new runTest();
                drawGraph graph1 = new drawGraph("Test1");
                drawGraph graph2 = new drawGraph("Test2");

                graph1.setSize(1000,1000);
                graph1.setVisible(true);
                graph1.addNodes(nodes);
                test1.BB.ShortestPathGraph = new myGraph();
                test1.BB.ShortestPathGraph.addVertices(nodes);
                System.out.println("Nodes added to ShortestPathGraph: " +
Arrays.asList(test1.BB.ShortestPathGraph.getVertices()));
                test1.BB.pick();
                //graph1.addEdges(test1.BB.ShortestPathGraph.getEdges());

                greedyLocalSearch GLS = new greedyLocalSearch(nodes);
                graph1.addEdges(GLS.getShortestPathGraph().getEdges());
/*
                graph2.setSize(1000,1000);
                graph2.setVisible(true);
                graph2.addNodes(nodes);
                test2.BB.ShortestPathGraph = new myGraph();
                test2.BB.ShortestPathGraph.addVertices(nodes);
                System.out.println("Nodes added to ShortestPathGraph: " +
Arrays.asList(test2.BB.ShortestPathGraph.getVertices()));
                test2.BB.pick();
                graph2.addEdges(test2.BB.ShortestPathGraph.getEdges());
*/

        }
}
```

drawGraph.java

```
package Visual;
```

```java
import javax.swing.*;
import java.awt.*;

import java.util.*;
import NetworkElements.*;
public class drawGraph extends JFrame{
	int width;
	int height;

	ArrayList<Node> nodes;
	ArrayList<Edge> edges;
	/**
	 * Constructor
	 */
	public drawGraph(){
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		nodes = new ArrayList<Node>();
		edges = new ArrayList<Edge>();
		width = 1000;
		height = 1000;
		this.setSize(1000,1000);
		this.setVisible(true);
	}
	/**
	 * Constructor 2
	 * @param name
	 */
	public drawGraph(String name){
		this.setTitle(name);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		nodes = new ArrayList<Node>();
		edges = new ArrayList<Edge>();
		width = 1000;
		height = 1000;
		this.setSize(1000,1000);

	}
	/*
	 * Add all nodes to graph
	 */
	public void addNodes(Node[] nodes){
		for(Node node : nodes){
			if(this.nodes.add(node)){
```

```
                    System.out.println("Node added : " + node);
                }
                this.repaint();
            }

        }

        public void addEdges(ArrayList<Edge> edges){
            for(Edge edge: edges){
                this.edges.add(edge);
                this.repaint();
            }
        }

        public void addEdges(Collection<Edge> edges){
            for(Edge edge: edges){
                this.edges.add(edge);
                this.repaint();
            }
        }

        public void showGraph(){
            this.setVisible(true);
        }

        public void paint(Graphics g){
            FontMetrics f = g.getFontMetrics();
            int nodeHeight = 5;//Math.max(this.height,
f.getHeight());
            g.setColor(Color.blue);
            for(Edge e: edges){
                g.drawLine(50 + e.getNodeOne().getX() * 50, 50 +
e.getNodeOne().getY() * 50, 50 + e.getNodeTwo().getX() * 50, 50 +
e.getNodeTwo().getY() * 50);
            }
            for(Node n : nodes){
                int nodeWidth = 5;//Math.max(this.width,
f.stringWidth(n.toString()) + width/2);
                g.setColor(Color.red);
                g.fillOval(50 + n.getX() * 50 - nodeWidth/2, 50 +
n.getY() * 50 - nodeHeight/2, nodeWidth, nodeHeight);
                g.setColor(Color.black);
                g.drawOval(50 + n.getX() * 50 - nodeWidth/2, 50 +
n.getY() * 50 - nodeHeight/2, nodeWidth, nodeHeight);
```

```
                g.drawString("(" + n.getX() + "," + n.getY() +
")" , 50 + n.getX() * 50 - f.stringWidth(n.toString())/2, 60 +
n.getY() * 50 + f.getHeight()/2);
            }
      }
/*
      public static void main(String[] args){
            drawGraph frame = new drawGraph("Test Window");

            frame.setSize(1000,1000);
            frame.setVisible(true);
            //add nodes add edges
      }
*/
}
```