CS 6385
Algorithmic Aspects of Telecommunication Networks
SUMMER 2017

Project 3
Network Topology Using Heuristic Method

Submitted

By

Aravind R
(**Net ID: axr156530**)

Instructor:
Andras Farago

Objective

The goal is to create and implement two different heuristic algorithms for this network topology design problem, and experiment with them. The Experimental setting consists of a complete undirected graph on n >=15 nodes. The graph generated must adhere to two rules: 1) every node must be reachable to every other node within 4 hops. 2) Every node must be connected to at least 3 other nodes. The heuristic algorithms ensure that the condition is met but compromises on optimum cost. The algorithm does not ensure optimum cost for the generated topology but ensure that an acceptable solution is found. The terminating condition is when all the nodes are connected in the network and all nodes have degree >=3.The project further compares the performance of two heuristic algorithm with Local Greedy Search and Branch and Bound algorithms.

Program Methodology

The project requires us to implement two heuristic algorithms to generate the network topology of n nodes where n>=15.The generated network must be fully connected and each of the nodes should be reachable within 4 hops from any other node. The principle goal is to minimize the overall cost of the network. Heuristic algorithms ensure that an acceptable solution is found based on the given conditions. In the submission I have implemented Bound and Branch algorithm and local greedy search algorithm to generate the topologies. A quantitative comparison of both algorithms is the done based on the results obtained for different values of n. A brief description of both algorithm is given below.

1) Local Greedy Search Algorithm

Local search is a method for finding a solution to a problem. It is based on iteratively improving an assignment of the variables until all constraints are satisfied. In particular, local search algorithms typically modify the value of a variable in an assignment at each step. The new assignment is close to the previous one in the space of assignment, hence the name local search. Greedy version of the local search algorithm proceeds by changing the current assignment by always trying to decrease (or at least, non-increase) its cost. The main problem of these algorithms is the possible presence of plateaus, which are regions of the space of assignments where no local move decreases cost. For the current scenario, the algorithm can be modified to always choose the least cost link among the set of current selected node set. The implementation avoids local optimum plateaus by considering the least cost links among all the selected nodes instead of just the current one. To further optimize cost, A link is selected as long as it is not largest incoming link to the destination node. If the selected link is the local maxima the algorithm can get stuck, but such a scenario was not encountered during the trial runs of the algorithm for values of n>=15. The terminating condition for the algorithm is set as when a connected topology is obtained and each of the nodes has degree greater than 3.

2) Branch and Bound

Branch and bound is an algorithm design paradigm for discrete and combinatorial problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. In the current implementation, an arbitrary node (in our case node 0) is selected as the starting node. The tree is created by adding the smallest link among the current node to the tree. During each addition, hop count from the root node is calculated. A node is added to the tree if and only if the hop count from the root node is less than 4. Thus the upper bound for the algorithm is set as hop count<=4.For the lower bound we select the degree count as 3. All nodes should have at least 3 degrees. Thus for each non leaf node we add at least 3 nodes if the cost permits. The terminating condition is the same as that of the previous algorithm.

Implementation

Representation

1) Network topology representation

Each node of the network is represented by the class of Node. The node object consists of x ,y coordinates and the degree of the node. Each of the nodes are generated randomly by using the Random class function of java. Once all nodes of the network are generated, a cost matrix is calculated to find the cost of moving from one node to another. This also functions as the adjacency matrix for the project. To store the selected links we keep a topology matrix. All selected links in the matrix is represented as 1. All ignored links are represented as 0. To ensure that the obtained topology is connected, we keep an adjacency list. Each node of the graph will have an entry and a list of neighbors that it is connected to in the list. The adjacency list ensures that no node is omitted from the overall topology even if it is not connected.

2)Finding the connectedness of the graph

To find the reliability of a network, we have to simulate the various combinations of the network. Each link of the network can be in two states, either up(1) or down(0). The whole network is up if the network is connected. I.e. each and every node in the network is reachable. In order to find whether the given topology is connected or not, we use the Breadth First Search algorithm. The BFS algorithm, we take an arbitrary node and visit all its immediate neighbors first. We store the neighbors in a queue as they are visited. After visiting all the neighbors of the current node, we then pop the next node from the queue and visit all the neighbors of that node. The termination condition is when there are no mode nodes in the queue. For a connected graph, each and every node is reachable from every other node in the

network. This means that at the end of BFS, every node in the network must have been visited.

Technologies used

Programming Language	Java
Operating System	Windows 10
IDE	NetBeans
Visualization Tools	MS Excel,Gephi

Program Details

Main Program	Local_Search.java

- The Local Search Algorithm implemented in Local_Search.java
- The nodes are represented as object of the Node class which stores details such as (x, y) coordinates and the degree of the node.
- The nodes are generated randomly and a cost matrix for the same is generated as well.
- There is a queue array list that maintains the list of visited nodes so far.
- To check the connectivity, the program maintains a separate adjacency list of all the nodes.
- The algorithm proceeds by selecting the link with the smallest cost from among the neighborhood of visited nodes so far.
- For each of the iteration, hop count is checked so that it does not exceed 4.
- The selected link is represented as 1 in link matrix.
- The system then uses Breadth First Search for checking if the given topologies are connected or not.
- To avoid local optimum, a link to a neighbor of the visited node is selected only if that link is not the maximum link to that node.
- Termination condition is checked for each of the iterations.
- At the end of the breadth first search algorithm, we check if all the nodes are visited. If all the nodes are visited then it is a connected network.
- The termination condition also ensures that each of the nodes have degree >=3
-

Main Program	Branch_Bound.java

- The Branch_Bound.java program implements the branch and bound algorithm.
- The topology matrix, cost matrix is same as that described in the previous algorithm.
- The program maintains a topology tree using a stack data structure.
- Initially the node 0 is select as the root node. The algorithm then select the lowest cost link from among links.
- On selection of the child node the, parent node is pushed to the stack.
- On each of the iterations, of the algorithm, we check the hop count of the current node with that of the root node.
- The current lowest link is selected only if hop count from root is less than 5 .
- If no optimum is found, the current node is set to the node from the top of the stack. I.e. the

program backtracks.

- We calculate the cost of each addition of the link.
- The algorithm takes the node degree as the lower bound and the hop count as the upper bound.
- During each iteration, graph connectedness and degree of each node is checked for termination.
- The program always tries to select the lowest cost links there by trying to minimize the cost.

Algorithm

The simulation can be done as follows:
Local Search greedy method

1. Read the number of nodes
2. Generate randomly n nodes using random points.
3. Calculate the cost matrix based on the generated points.
4. Set the current node as 0.
5. Identify the smallest link in the neighborhood of the current visited nodes.
6. If the link not visited calculate the hop count
7. If hop count is less than four, select the current node.
8. Add the node to the Queue of visited nodes.
9. Set the link status as selected.
10. Calculate the cost.
11. Increment the degree count of source node and destination nodes,
12. Increment the hop count of all visited node to the selected node as hops to previous node +1.
13. If graph is connected and degree of all nodes>=3 the step 14 else step 5
14. Print the total topology and cost of the topology
15. Stop

Branch and Bound

1. Read the number of nodes
2. Generate randomly n nodes using random points.
3. Calculate the cost matrix based on the generated points.
4. Initialize the stack
5. Select node 0 as the current node.
6. Set the node 0 as root node of the tree.
7. Find the smallest link among the current node links.
8. Calculate the hop count of current node from root node.
9. If current hop count <=4(upper bound) select node and step 10.
10. Push the current node to stack
11. Set link as selected.
12. Set the new node as selected goto step15 .
13. Else Pop the last element from stack (Backtracking) and goto step 7
14. Check if graph connected and each node degree 3(lower bound) Step 15 else step 7.
15. Calculate the current cost .
16. Print the Network Topology
17. End

Running the Program

1. Copy the Local_Search.java, Branch_Bound.java program to the location to run.
2. To Compile Javac Local_Search.java Branch_Bound.java
3. To run the program Java Local_Search and java Branch_Bound

Program

Local_Search.java

```
package local_search;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;
class Nodes
{
    int x;
    int y;
    int degree;
    Nodes(int x,int y,int n)
    {
        this.x=x;
        this.y=y;
        degree=0;
    }
    public void print_nodes()
    {
        System.out.println("x = "+x+" y = "+y+" degree = "+degree);
    }
   public int get_cost(Nodes n1,Nodes n2)
   {
       int length=(int)Math.sqrt( Math.abs(Math.pow((n1.x-n2.x),2.0)-Math.pow((n1.y-n2.y),2.0)));
       return length;
   }
}

public class Local_Search
{
     void print_nw(int topo[][],int m,int n)
    {
        int i,j;
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                System.out.print("\t"+topo[i][j]);
            }
            System.out.print("\n");
        }
    }
    public static void main(String[] args)
    {
        ArrayList<Nodes> q = new ArrayList();
       Random r= new Random(100);
        System.out.println("Local Search!! Enter the number of nodes:");
        Scanner in = new Scanner(System.in);
        int nm = in.nextInt();
        ArrayList<Integer> []nod = new ArrayList[nm];   //adj list
        int sum=0;
         int mat[][]=new int [nm][];
         int topo[][]=new int[nm][nm];
         int i,j;
          Local_Search b=new Local_Search();
         for(i=0;i<nm;i++)
         {
             nod[i] = new ArrayList();
```

```
    for(j=0;j<nm;j++)
    {
        if(i==j)
            topo[i][j]=-1;
        else
            topo[i][j]=0;
    }
}
 b.print_nw(topo, nm,nm);
ArrayList<Integer> visited= new ArrayList();   //visited nodes
for(i=0;i<nm;i++)
{
    int x=r.nextInt(100);
    int y=r.nextInt(100);
    System.out.println("x="+x+"y="+y);
    Nodes n=new Nodes(x,y,nm);
    q.add(n);
}
System.out.println("Network Topology:");
for( i=0;i<nm;i++)   //presetting the NW topology
{
    Nodes n1=q.get(i);
    mat[i]=new int[nm];
    for(j=0;j<nm;j++)
    {
        Nodes n2=q.get(j);
        mat[i][j]=n1.get_cost(n1, n2);
        if(i==j)
            mat[i][j]=Integer.MAX_VALUE;
        System.out.print("\t"+mat[i][j]);
    }
    System.out.print("\n");
}
 visited.add(0);
 int l=0;
 while(l<100)
 {
       int smallest=Integer.MAX_VALUE,src=0,dest=0;
       for(i=0;i<visited.size();i++)//finding the smallest among the visited links
       {
           int k=visited.get(i);
           for(int c=0;c<nm;c++)
           {
               if(mat[k][c]<smallest)
               {
                   if(c==src && dest==k)
                   {
                       System.out.println("c="+c+"k="+k);
                       continue;           //not to fall in local optimum
                   }
                   smallest=mat[k][c];
                   dest=c;
                   src=k;
               }
           }
       }
       if(topo[src][dest]==0)
       {
           System.out.println(src+","+dest);
           sum+=mat[src][dest];
           topo[src][dest]+=1;
           nod[src].add(dest);
           q.get(src).degree+=1;
           q.get(dest).degree+=1;
           if(topo[dest][src]==0)                  //Check if alternate path exist
           {
               topo[dest][src]+=1;
               nod[dest].add(src);
           }
           mat[src][dest]=Integer.MAX_VALUE;       //not revisit the link again
           mat[dest][src]=Integer.MAX_VALUE;
```

```
                    visited.add(dest);                    //adding to visited list
                    for(i=0;i<nm;i++)                    //ve hop count<4 if the path is accepted
                    {
                        if((topo[i][src]+1)<=4 && topo[i][dest]==0 && topo[i][src]>0)        //if
hop count through current selection is less than n and no other path selected
                        {
                               topo[i][dest]=topo[i][src]+1;
                               mat[i][dest]=Integer.MAX_VALUE;
                        }
                    }
                }
                else
                {

                }
                l++;
               if(b.isconnected( nod,nm,q))
                   break;
          }
          System.out.println("Total cost:"+sum);
          for(i=0;i<q.size();i++)
          {
              System.out.println("Node "+i+" : "+q.get(i).degree);
          }
    }
     boolean isconnected( ArrayList<Integer> []nod,int n,ArrayList<Nodes> no)
    {
        boolean flag=true;
      try{
      ArrayList<Integer> q = new ArrayList();
      q.add(0);
      int visited[]=new int[n], sum=0;
      while(!q.isEmpty())
      {
          int curr=q.remove(0);
          for(int i=0;i<nod[curr].size();i++)
          {
              int k=nod[curr].get(i);
              if(visited[k]!=1)
              {
                    visited[k]=1;
                    sum+=1;
                    q.add(nod[curr].get(i));
              }
          }
      }
      if (sum!=n)
          flag=false;
      for(int k=0;k<n;k++)
          if(no.get(k).degree<3)
          {
            flag=false;
            break;
          }

      }
      catch(Exception e){/*System.out.println("Here :"+e);*/}
      return flag;
  }
    int hop_count(int src,int dest,int topo[][],int nm)
    {
        int flag=1,i,j;
        for(i=0;i<nm;i++)
        {
                if((topo[i][src]+1)>2 && topo[i][dest]==0)
                {
                    flag=0;
                }
        }
        return flag;
    }}
```

Branch_Bound.java

```
package branch_bound;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;
class Node
{
    int x;
    int y;
    int degree;

    Node(int x,int y,int n)
    {
        this.x=x;
        this.y=y;
        degree=0;

    }
   public void print_nodes()
   {
       System.out.println("x = "+x+" y = "+y+" degree = "+degree);
   }
   public int get_cost(Node n1,Node n2)
   {
       int length=(int)Math.sqrt( Math.abs(Math.pow((n1.x-n2.x),2.0)-Math.pow((n1.y-n2.y),2.0)));
       return length;
   }
}
class Links
{
    Node n1;
    Node n2;
    int cost;
    public Links(Node n1,Node n2,int cost)
    {
        this.n1=n1;
        this.n2=n2;
        this.cost=cost;
    }
}
public class Branch_Bound
{
    void print_nw(int topo[][],int m,int n)
    {
        int i,j;
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                System.out.print("\t"+topo[i][j]);
            }
            System.out.print("\n");
        }
    }

    public static void main(String[] args)
    {
        ArrayList<Node> q = new ArrayList();

        Random r= new Random(100);
        System.out.println("Enter the number of nodes:");
        Scanner in = new Scanner(System.in);
        int nm = in.nextInt();
        int cost=0;
        int sum=0;
        int mat[][]=new int [nm][];             //cost matrix
        int topo[][]=new int[nm][nm];           //topology matrix
        int hops[][]=new int[nm][nm];           //hops matrix
        ArrayList<Integer> []nod = new ArrayList[nm];
        int i,j;
```

```
Branch_Bound b=new Branch_Bound();
for(i=0;i<nm;i++)  //generation the nodes
{
    int x=r.nextInt(100);
    int y=r.nextInt(100);
    System.out.println("x="+x+"y="+y);
    Node n=new Node(x,y,nm);
    q.add(n);
}
ArrayList<Integer> Stack= new ArrayList();   //visited nodes
for( i=0;i<nm;i++)   //presetting with cost value the NW topology
{
    Node n1=q.get(i);
    mat[i]=new int[nm];
    nod[i]=new ArrayList();
    for(j=0;j<nm;j++)
    {
        Node n2=q.get(j);
        int c;
        if(i==j)
        {
            c=Integer.MAX_VALUE;
            topo[i][j]=-1;
            mat[i][j]=c;
        }
        else
        {
           c= n1.get_cost(n1, n2);
            mat[i][j]=c;
            topo[i][j]=0;
        }
        System.out.print("\t"+mat[i][j]);
    }
    System.out.print("\n");
}
b.print_nw(topo, nm,nm);
Stack.add(0);   //Adding zero node for the tree
int l=0,hop=0;
l=Stack.remove(0);    //retreive the first element of the stack
int counter=0;

while(!b.isconnected(nod,nm,q))//!b.isconnected(hops,nm))
{
    int k=b.smallest(l,mat,topo,nm);    //Getting the smallest from the current node
    if((hop+1)<=4 && k>-1 )
    {
         topo[l][k]=1;                   //select the link in the topology
        topo[k][l]=1;
        cost+=mat[l][k];
        System.out.println(""+l+","+k);
        hops[k][l]=hop+1;
        hops[l][k]=hop+1;
        nod[l].add(k);
        nod[k].add(l);
        q.get(l).degree+=1;
        q.get(k).degree+=1;
        Stack.add(0,l);                //add the node to top of the stack
        l=k;
        hop++;

    }
    else
    {
        l=Stack.remove(0);
        hop--;
    }
    counter++;
}
System.out.println("Topology");
b.print_nw(topo,nm,nm);
System.out.println("Hops");
```

```
        b.print_nw(hops, nm,nm);
        for(i=0;i<nm;i++)
        {
            System.out.println("Degree of node :"+i+" :"+q.get(i).degree);
        }
        System.out.println("Cost : "+cost);
    }
     boolean isconnected( ArrayList<Integer> []nod,int n,ArrayList<Node> no)
    {
        boolean flag=true;
      try{
      ArrayList<Integer> q = new ArrayList();
      q.add(0);
      int visited[]=new int[n], sum=0;

      while(!q.isEmpty())
      {
          int curr=q.remove(0);
          for(int i=0;i<nod[curr].size();i++)
          {
              int k=nod[curr].get(i);
              if(visited[k]!=1)
              {
                    visited[k]=1;
                    sum+=1;
                    q.add(nod[curr].get(i));
              }
          }
      }
      if (sum!=n)
          flag=false;
      for(int k=0;k<n;k++)
          if(no.get(k).degree<3)
          {
            flag=false;
            break;
          }

      }
      catch(Exception e){/*System.out.println("Here :"+e);*/}
      return flag;
  }
    int smallest(int i,int mat[][],int topo[][],int n)
    {
        int j,smallest=Integer.MAX_VALUE,idx=-1;
        int flag=0;
        try
        {
        for(j=0;j<n;j++)
        {
            if(smallest>mat[i][j] && topo[i][j]==0)
            {
                smallest=mat[i][j];
                idx=j;
            }
        }
         for(int k=0;k<n;k++)
        {
            if(mat[idx][k]>smallest && idx!=k)
            {
                flag=1;
            }
        }
      }catch(Exception e){/*System.out.println("Here!@"+e);*/}
        if(flag==0)
            idx=-1;
        return idx;
    }
}
```

Analysis

Generated Topologies

Branch and Bound

1) N = 15

Cost = 1226

Node coordinates

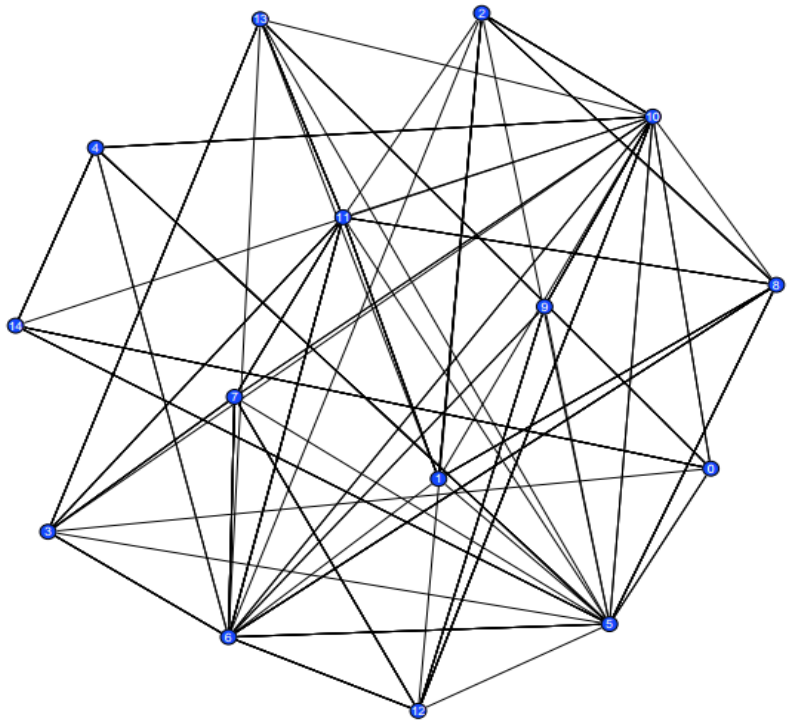
```
Enter the number of nodes:
15
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
```

Links

(0,14) (14,4) (4,5) (5,8) (5,14) (5,6) (5,10) (5,7) (5,0) (5,11) (5,12) (5,1) (5,2) (5,9) (5,3) (5,13) (4,10) (10,2) (10,12) (10,9) (10,6)

(10,6) (10,0) (10,7) (10,8) (10,11) (10,14) (10,1) (10,3) (10,13) (4,6) (6,7) (6,8) (6,11) (6,3) (6,12) (6,1) (6,2) (6,9) (6,13)

Topology Generated



2) N = 16

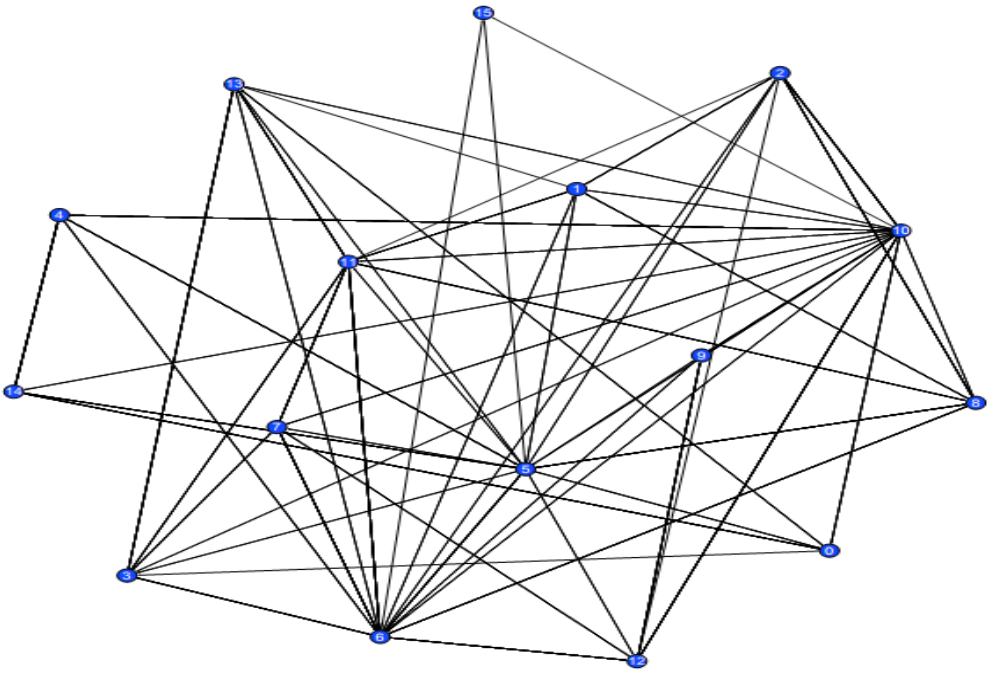
Cost = 1380

Links

(0,14) (14,4) (4,5) (5,8) (5,14) (5,6) (5,10) (5,7) (5,0) (5,11) (5,12) (5,1) (5,2) (5,9) (5,15) (5,3) (5,13) (4,10) (10,2) (10,12) (10,9)

(10,6) (10,0) (10,7) (10,8) (10,11) (10,14) (10,1) (10,3) (10,13) (10,15) (4,6) (6,7) (6,8) (6,11) (6,3) (6,12) (6,1) (6,2) (6,9) (6,15) (6,13)

Topology Generated



---

3) N = 17

Cost = 1496

Node coordinates

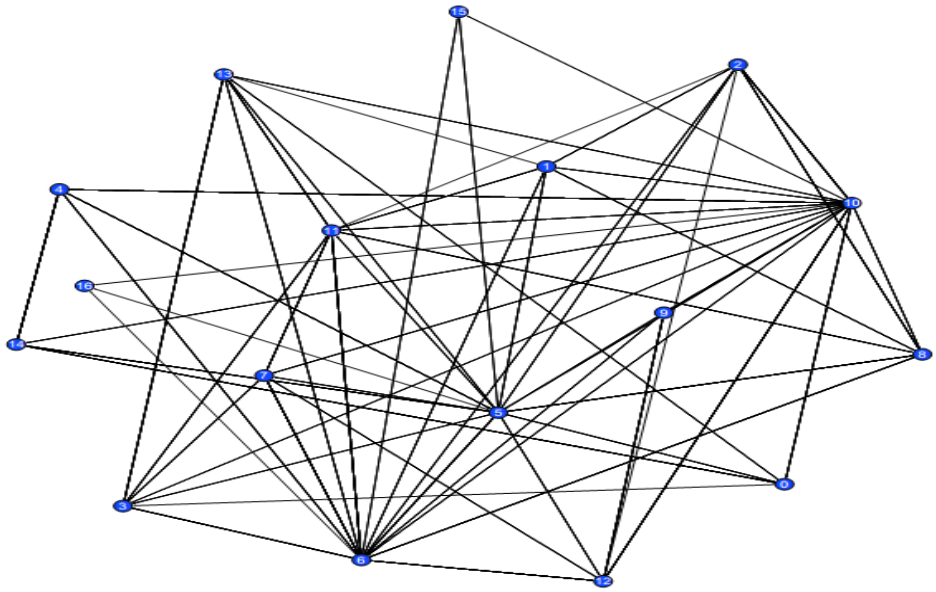
```
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73
```

Links

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

(10,6) (10,0) (10,7) (10,8) (10,11) (10,14) (10,1) (10,3) (10,13) (10,15) (4,6) (6,7) (6,8) (6,11) (6,3) (6,12) (6,1) (6,2) (6,9) (6,15) (6,13) (6,16)

Topology Generated



---

4) N = 18

Cost = 1669

Node coordinates

x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73
x=44y=94

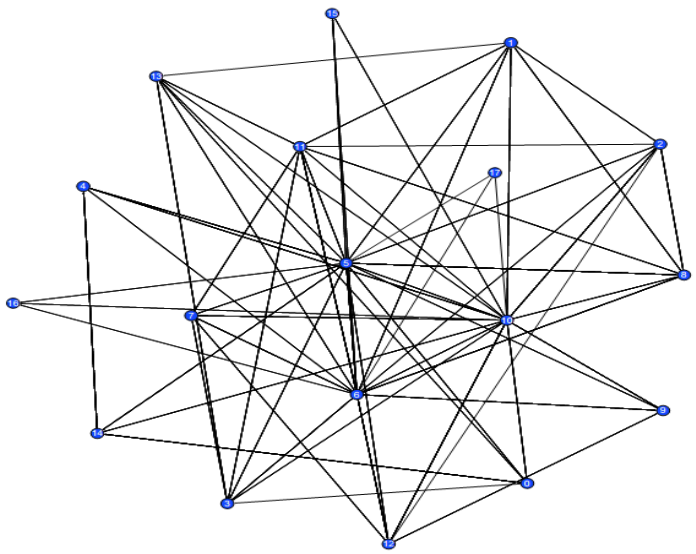
Links

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

(10,12) (10,9) (10,16) (10,6) (10,0) (10,7) (10,8) (10,11) (10,14) (10,1) (10,3) (10,13) (10,15) (10,17) (4,6) (6,7) (6,8) (6,11) (6,3)

(6,12) (6,1) (6,2) (6,9) (6,15) (6,13) (6,17) (6,16)

Topology Generated



5) N = 19

Cost = 1758

Node coordinates

x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73
x=44y=94

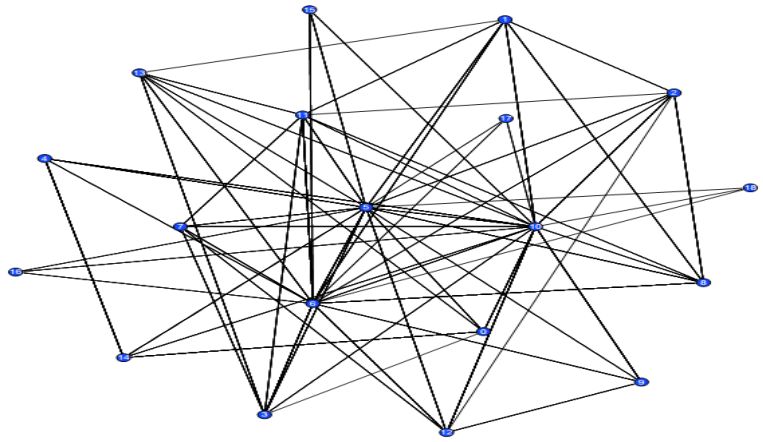
Links

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

(10,12) (10,9) (10,16) (10,6) (10,0) (10,7) (10,8) (10,11) (10,14) (10,1) (10,3) (10,13) (10,15) (10,17) (4,6) (6,7) (6,8) (6,11) (6,3)

(6,12) (6,1) (6,2) (6,9) (6,15) (6,13) (6,17) (6,16)

Topology Generated



Local Greedy Search

1) N = 15

Cost = 715

Node coordinates

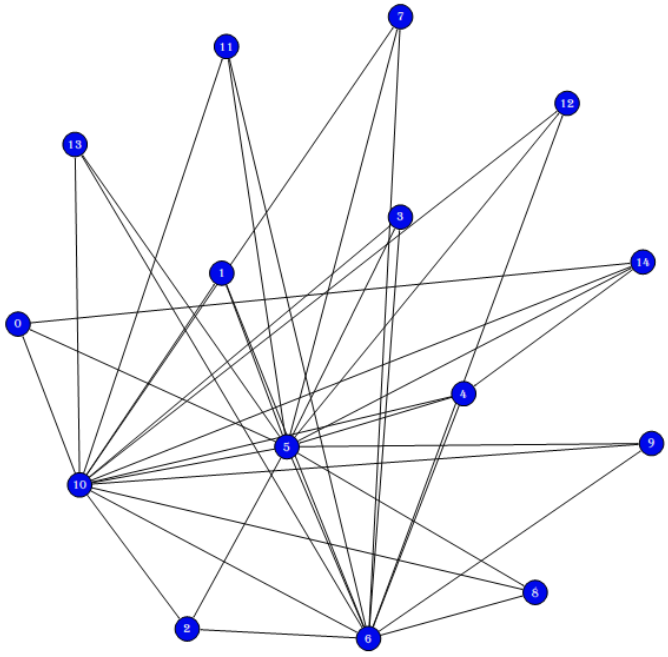
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28

Links

(0,14) (14,4) (4,5) (5,8) (8,2) (2,10) (8,1) (8,6) (6,7) (7,11) (11,13) (13,3) (11,6) (10,12) (12,9) (1,2) (11,1) (11,8) (9,10) (5,14) (6,5) (6,12) (3,6) (12,7)

(3,7) (3,11) (0,13) (10,4) (1,13) (11,2) (12,2) (10,5) (10,6) (6,4) (0,10) (7,10) (0,3) (5,0) (6,9)

Topology Generated



---

2) N = 16

Cost = 875

Node coordinates

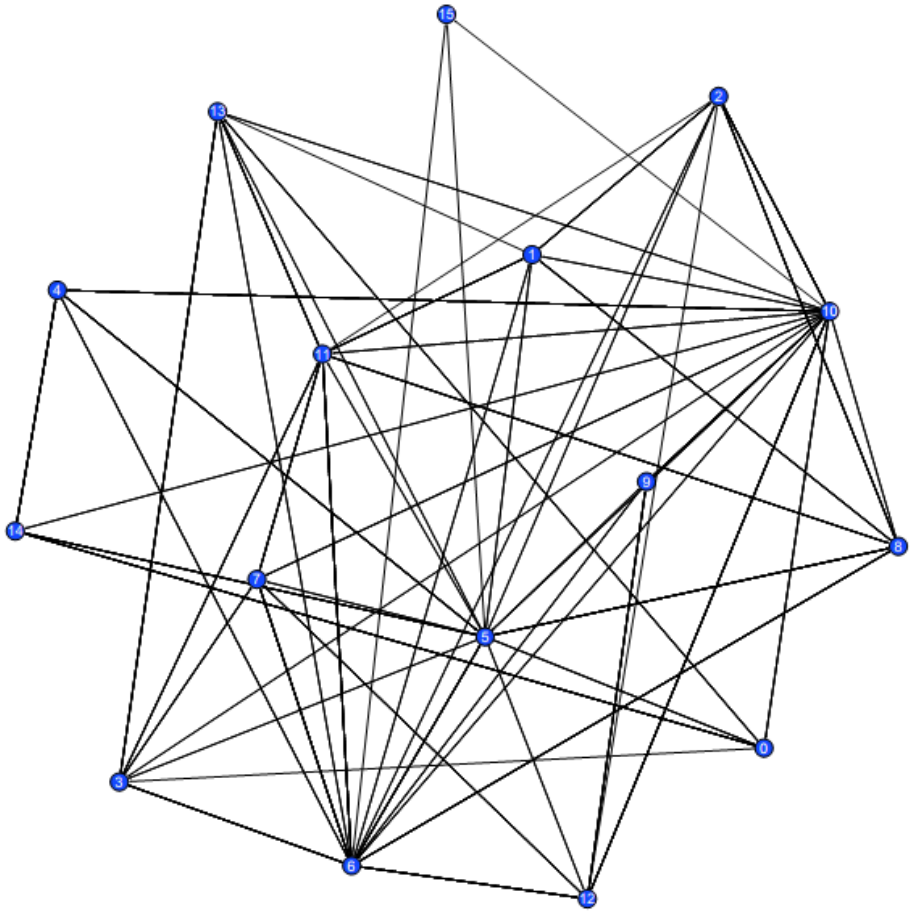
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95

Links

(0,14) (14,4) (4,5) (5,8) (8,2) (2,10) (8,1) (1,15) (8,6) (6,7) (7,11) (11,13) (13,3) (11,6) (10,12) (12,9) (1,2) (11,1) (11,8) (15,8) (9,10) (5,14) (15,11) (6,5)

(6,12) (3,6) (12,7) (3,7) (3,11) (0,13) (10,4) (1,13) (11,2) (13,15) (12,2) (10,5) (10,6) (15,7) (6,4) (0,10) (7,10) (0,3) (15,14) (5,0) (15,6) (6,9)

Topology Generated



3) N = 17

Cost = 935

Node coordinates

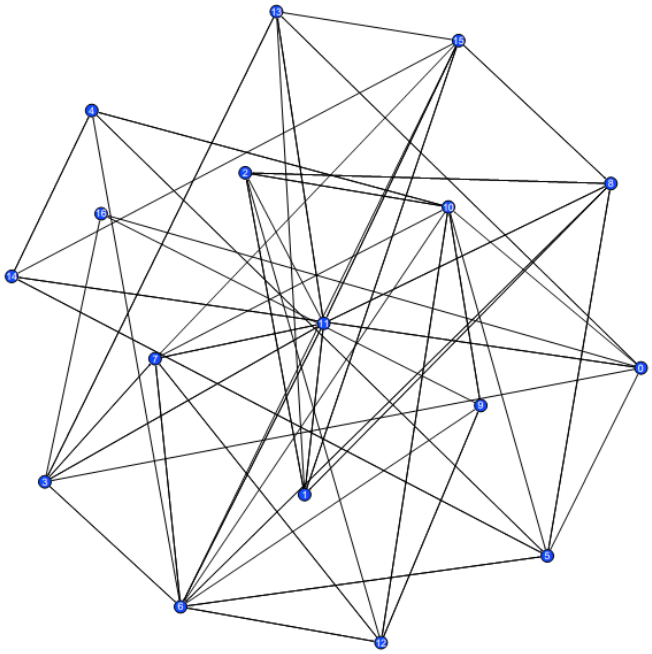
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73

Links

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

(15,11) (6,5) (0,16) (6,12) (3,6) (12,7) (3,7) (3,11) (0,13) (10,4)

Topology Generated



---

4) N = 18

Cost = 527

Coordinates

```
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73
x=44y=94
x=87y=68
```

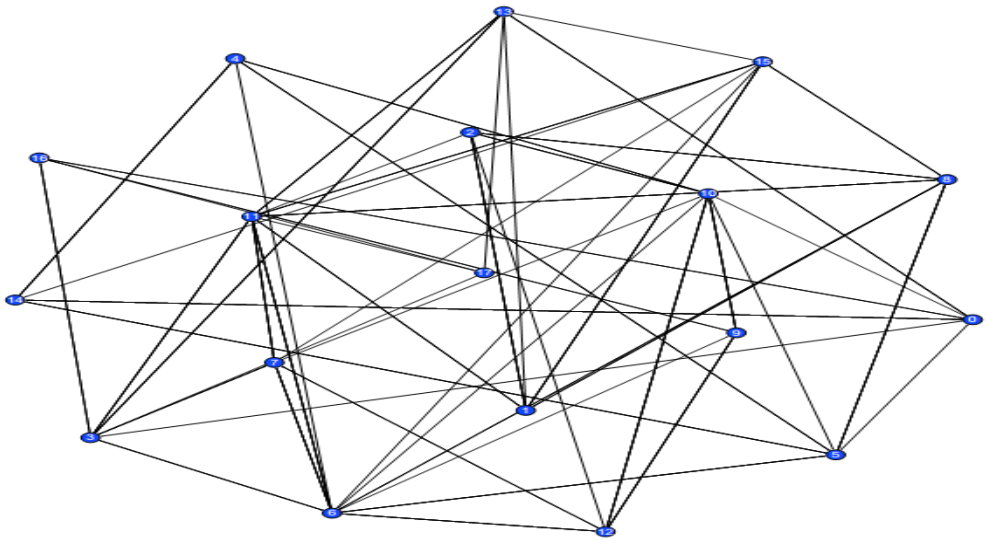
Links

(0,14) (14,4) (4,5) (5,8) (8,2) (2,10) (8,1) (1,15) (8,6) (6,7) (7,11) (11,17) (17,13) (17,3) (13,11) (3,13) (11,6) (10,12)

(12,9) (1,2) (11,1) (11,8) (3,16) (15,8) (9,10) (9,16) (16,17) (5,14) (15,11) (6,5) (0,16) (6,12) (3,6) (12,7) (3,7) (3,11) (0,1

(0,13) (10,4)

Topology Generated



5) N = 19

Cost = 527

Links

(0,14) (14,4) (4,5) (5,8) (8,2) (2,10) (2,18) (18,8) (8,1) (1,15) (8,6) (6,7) (7,11) (11,17) (17,13) (17,3) (13,11) (3,13)

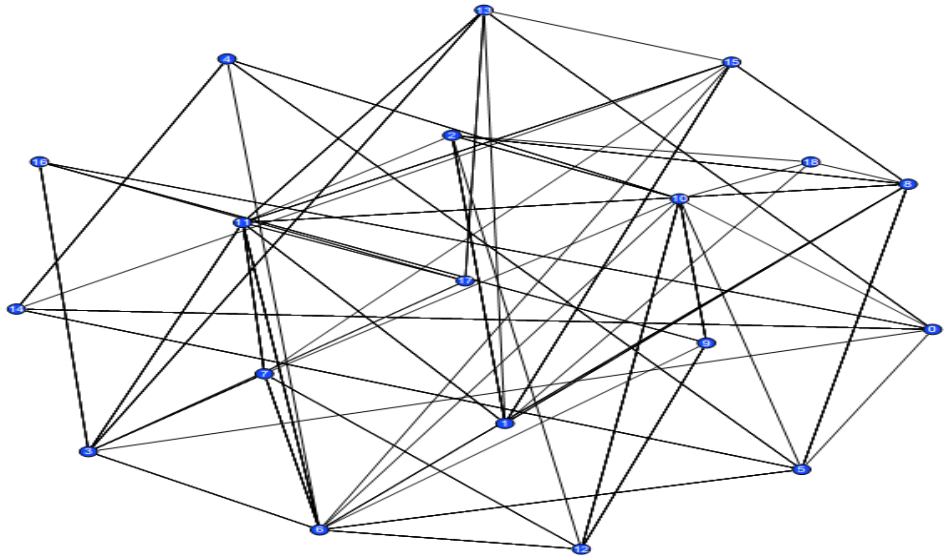
(11,6) (10,12) (12,9) (1,2) (11,1) (11,8) (3,16) (1,18) (15,8) (9,10) (9,16) (16,17) (5,14) (18,10) (15,11) (6,5) (0,16) (6,12) (3,6) (12,7)

(3,7) (3,11) (0,13) (10,4)

Coordinates

```
x=15y=50
x=74y=88
x=91y=66
x=36y=88
x=23y=13
x=22y=17
x=56y=57
x=52y=59
x=80y=78
x=73y=19
x=53y=28
x=65y=72
x=67y=31
x=48y=92
x=0y=28
x=74y=95
x=16y=73
x=44y=94
x=87y=68
```

Topology Generated



Branch and Bound

n	Cost
15	1226
16	1380
17	1496

18	1669
19	1758
20	2564
30	3296
50	4114
70	6675
100	9883

- In the observations, we can see that the cost of the network increases as the number of the nodes increases.
- The cost found by the iteration is not ideal and there are a lot of links that are included as a result of the condition : minimum 3 degree for each of the node.
- The algorithm tries to avoid the high cost links.

Local Greedy Search

n	Cost
15	715
16	875
17	495
18	527
19	573
20	516
30	1133
50	1454
70	1639
100	2262

- The values found by Local Greedy search is much more efficient .
- The cost also varies according to the placement of the nodes. This is evident in the dip in total costs between values of n..
- The increase in cost with number of node is not a constant one.

Variation in cost in each step

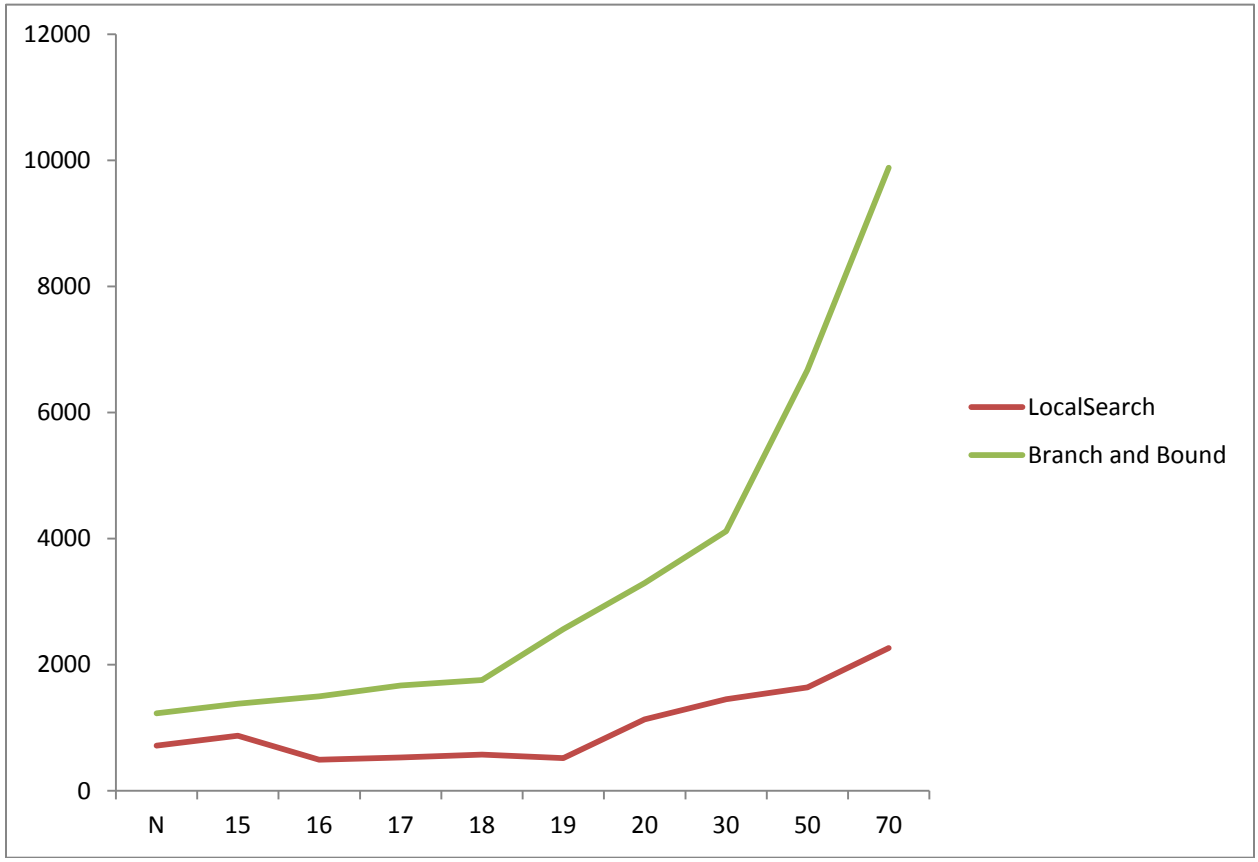
Branch and bound and Local Search : n=15

Step	Cost (B&B)	Cost(Local Greedy)
1	16	16
2	33	33
3	36	36
4	54	54
5	73	58
6	94	58
7	122	66
8	151	77
9	183	80
10	217	80
11	259	90
12	307	101
13	355	113
14	405	126
15	474	136
16	544	149
17	569	162
18	569	175
19	582	192
20	599	211
21	627	232
22	657	255
23	687	278
24	729	301
25	771	325
26	824	349
27	880	374
28	937	399
29	1000	424
30	1029	449
31	1032	474
32	1043	502
33	1055	530
34	1078	559
35	1101	589
36	1126	619
37	1159	650

38	1192	682
39	1226	715

Conclusions

1) Comparison



The graph shows that the Branch and Bound Algorithm costs more than the Local greedy Search Algorithm. This is because the local greedy search algorithm implementation always search for the minimum link from the set of nodes that are in the neighborhood. This allows the algorithm to always select the lowest link from the set of available nodes. In contrasts the Branch and Bound algorithm always searches for the minimum links from the set of current node under consideration. The result is limited number of nodes to move based on the current node. On the step by step iterations we see that the Local search tends to find local minimum of all nodes that are currently selected. Thus stepwise increase in total cost is better in local search as compared to that of branch and bound algorithm. The current implementation of the both algorithm shows that the Local search algorithm also finds better solutions as the number of nodes in the network increases when compared to the Branch and bound algorithm. The results are based on the current implementation of both algorithms. If the branch and bound algorithm are implemented based on cost for lower and upper bound , instead of hop count and Node degrees, a better performance could have been achieved.

Reference

1. Class notes
2. Branch and bound : https://en.wikipedia.org/wiki/Branch_and_bound
3. Greedy Local Search: https://www.researchgate.net/post/What_is_the_difference_between_local_search_paradigm_and_greedy_paradigm
4. Hueristic comparison : https://pdfs.semanticscholar.org/b0cb/70cc6b3670909bed45a16fa99657082e122c.pdf
5. Gephi : https://gephi.org/