**Algorithmic Aspects of Telecommunication Networks**

**Project 2 – Aparna Pavithran (axp161730)**

**Implementation of Nagamochi Ibaraki Algorithm to find minimum cut in an undirected graph.**

**Index**

|  |  |  |
| --- | --- | --- |
| 1 | Objective | 3 |
| 2 | Nagamochi Ibaraki Algorithm | 3 |
| 3 | Pseudo code | 4 |
| 4 | Code Implementation | 4 |
| 5 | Experimental Results | 6 |
| 6 | Graphs | 19 |
| 7 | Observation and Conclusion | 23 |
| 8 | Appendix : Source code | 24 |
| 9 | Readme | 32 |
| 10 | References | 32 |

1. **Objective**

**Input**: Number of nodes (n) in the Graph, number of edges (m) in the Graph.

* To implement Nagamochi Ibaraki Algorithm.
* To find minimum cut in an undirected graph.
* To find the edge connectivity of the graph for various values of m.
* To find the smallest edge connectivity, largest edge connectivity, and difference between small and large connectivity for a given value of m.
* To find average edge connectivity for a given value of m by creating 4 different graphs by taking random edges.

1. **Nagamochi Ibaraki Algorithm**

Nagamochi and Ibaraki developed a simple deterministic algorithm for minimum cuts. This is by using edge connectivity of the graph and edge connectivity of two nodes.

The edge connectivity of the graph is denoted by and edge connectivity between two nodes x and y are denoted as .

**Description**:

* Done in a recursive way on the graph G to find connectivity of graph G as .
* Do Maximum Adjacency (MA) ordering and pick two nodes as the last two nodes in the MA ordering.
* Let the picked nodes be x and y. Merge these nodes and make a new Graph called .
* Over all connectivity of the graph G is .
* Now do the same steps for the new graph Gxy.
* Finally will get the minimum connectivity for the entire graph by recursion.

**MA Ordering:**

Let the graph G has nodes say n in numbers.

* Take any random node 1.
* Add 1 to the already chosen list.
* Take any other node i+1 that has maximum number of edges connected with the nodes in already chosen list {1,….i}.
* where denotes the degree of the node Vn.

Repeating this steps for the merged graph Gxy will make sure to get the minimum edge connectivity for the entire Graph.

Number of nodes is always 21.

For any edge given between 20 and 200, create 5 graphs and find out the edge connectivity for all the graphs. Edges are taken randomly so that edge connectivity may differ for the same number of edges.

Take the average of edge connectivity for each value of m.

For each value of m take the difference between smallest and largest value of edge connectivity.

1. **Pseudo code**

Step 1: Read values of number of nodes (n) and number of edges (m).

Step 2: Create an adjacency list for the Graph G by giving random edges.

Step 3: Find the minimum cut of the graph. If the graph is disconnected, Minimum cut is 0, in adjacencyListCreation.

Step 4: While number of nodes > 2

Step 5: If G is connected then, MA ordering of nodes of graph G, are created. MA ordering is V1, V2, … Vn.

Step 6: Edge connectivity is calculated, and it is degree of Vn.

Step 7: Nodes Vn-1 and Vn are merged to get a new graph. Smallest number of the node is given as the name of merged node.

Step 8: Find the new adjacency list for the new merged graph.

Step 9: Find the minimum edge connectivity as .

Step 10: Repeat the steps 4 to 8 and calculate the minimum edge connectivity.

Step 11: Take 5 graphs for each value of m and find out edge connectivity for each graphs.

Step 12: Take the difference between large and small edge connectivity for each value of m.

1. **Code Implementation**

The program is implemented in Java in MAC OS and by using Eclipse IDE.

The implementation takes any value as number of nodes, but here it is given as n=21 as it is fixed. If n is given as a negative number then by default n is considered as 21. The number of edges for input 21 must be between 20 and 200. For other inputs it must be between n-1 and (n\*(n-1)-n)/2, as this is an undirected graph. If any input is given other than this, random value is taken between 20 and 200 and continue to create a graph.

Multiple classes and methods in the classes are explained in detail here.

**Driver.Java**

**Main method**

This is the driver class of the program. This takes input values, collect output values, find out the average of edge connectivity (lamda values), find out the smallest and largest of edge connectivity and their differences.

This calls a method adjacencyListCreation inside Graph.java to create adjacency lists. This is called in a loop as 5 times to create 5 graphs and calculate 5-edge connectivity for 5 different graphs for the same m and n values.

**Graph.java**

**adjacencyListCreation method**

This method creates an adjacency list with m edges. As this is an undirected graph each edge is considered as 2 values in adjacency list.

If the graph is disconnected then the minimum cut is calculated as 0 and return the result. Else NIAlg method inside NagamochiIbarakiAlg.java is called. NIAlg method takes input as adjacency list and n value.

**NagamochiIbarakiAlg.java**

**NIAlg method**

This method takes adjacency list and n values as input. If n is equal to 2 then degree of the node is returned as the edge connectivity. As this graph is with out self-loop, just the number of edges connected will give the result (parallel edges are counted).

If the graph is disconnected then the minimum cut is calculated as 0 and return the result. Else find the MA ordering of the given graph by calling MAOrdering method.

After MAOrdering, MAOrdering will return the last two nodes in the order as well as the edge connectivity of those two picked nodes. Call Merge method with the two nodes as the input. Merge method will return new adjacency list for the merged graph. Call NIAlg method again with the new adjacency list and n-1.

**MAOrdering method**

Input of the method is adjacency list of the graph and number of nodes in the graph.

This method picks a node say as node 0 and start ordering the other nodes based on the number of times each connected to the picked nodes list.

Output of the method is an array with the last two nodes and the degree of the last node.

Merge method

This method takes input as adjacency list of the graph, two nodes which need to be merged, the number of the nodes in the graph.

The method simply merges the two nodes and names the new node as the smallest of the node number. Create a new adjacency list if needed with parallel edges.

Output of the method is the new adjacency lsit.

**Note**:- Entire source code is given in the Appendix.

1. **Experimental Results**

n – number of nodes = 21

**Sample values**

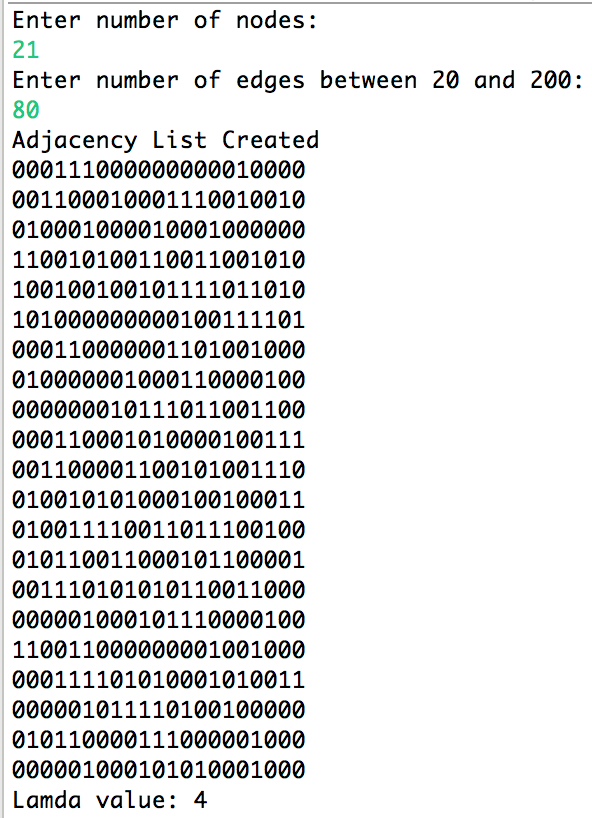
**Let m=80, 84, 88 and 92.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SNo | Number of edges | Edge Connectivity Or Lamda value | | | | | | | |
| Graph 1 | Graph 2 | Graph 3 | Graph 4 | Graph 5 | Small Value | Large Value | Average Value |
| 1 | 80 | 4 | 3 | 3 | 3 | 3 | 3 | 4 | 3.2 |
| 2 | 84 | 4 | 4 | 4 | 2 | 4 | 2 | 4 | 3.6 |
| 3 | 88 | 1 | 5 | 4 | 4 | 5 | 1 | 5 | 3.8 |
| 4 | 92 | 6 | 3 | 4 | 5 | 5 | 3 | 6 | 4.6 |

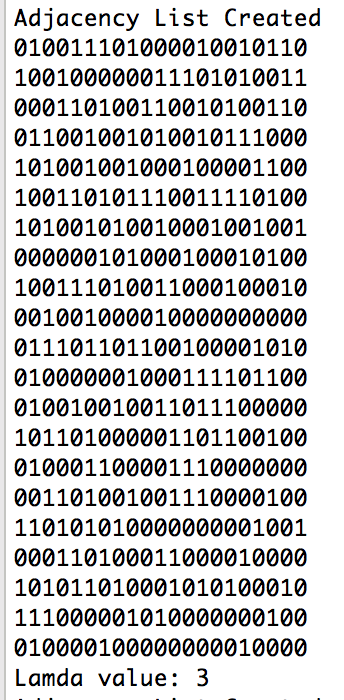
**Screen shots**

**m=80**

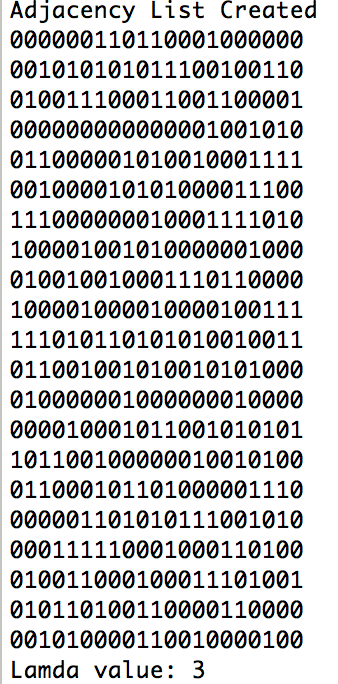
**Graph 1 Adjacency List:**

****

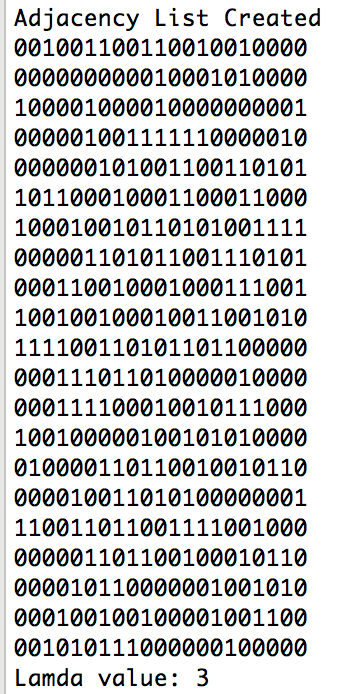
**Graph 2 Adjacency List:**

****

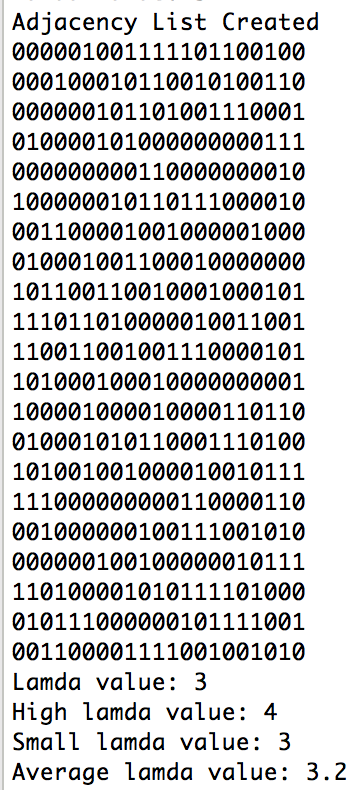
**Graph 3 Adjacency List:**

****

**Graph 4 Adjacency List:**

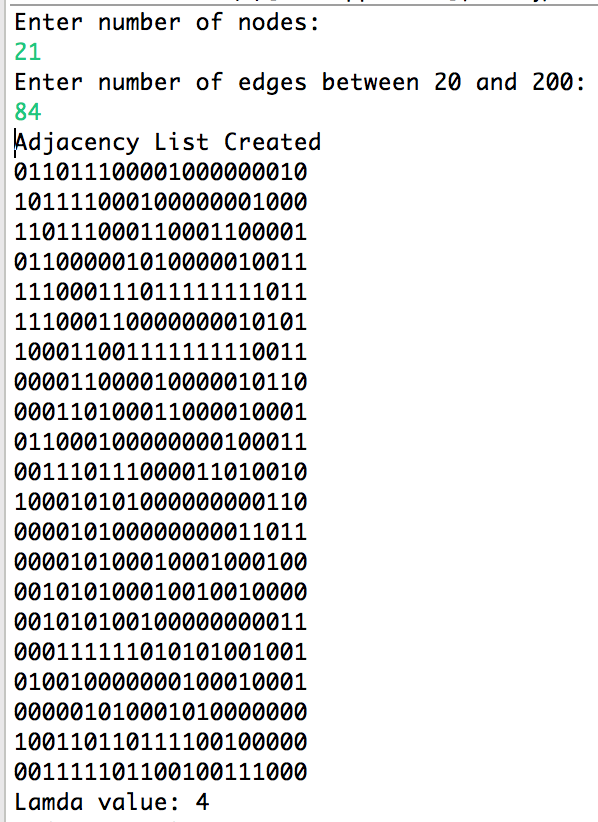
****

**Graph 5 Adjacency List:**

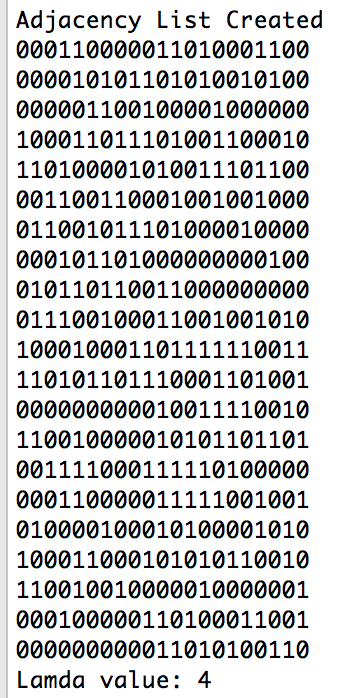
****

**m=84**

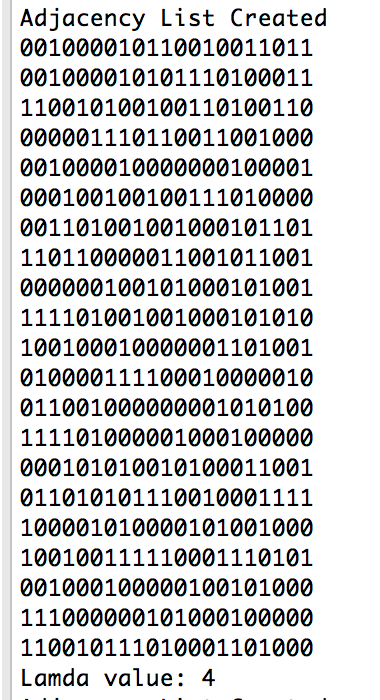
**Graph 1 Adjacency List:**

****

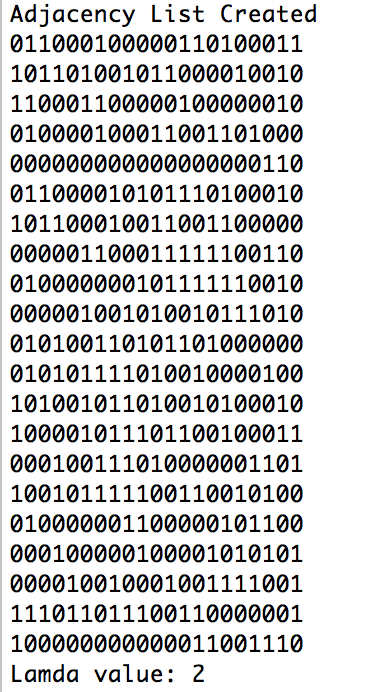
**Graph 2 Adjacency List:**

****

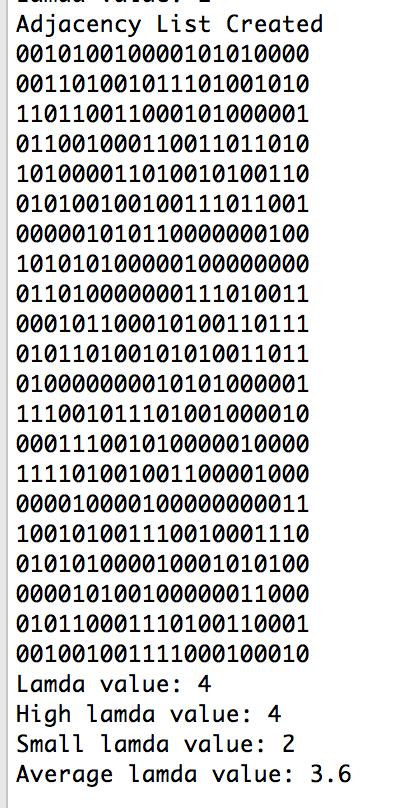
**Graph 3 Adjacency List:**

****

**Graph 4 Adjacency List:**

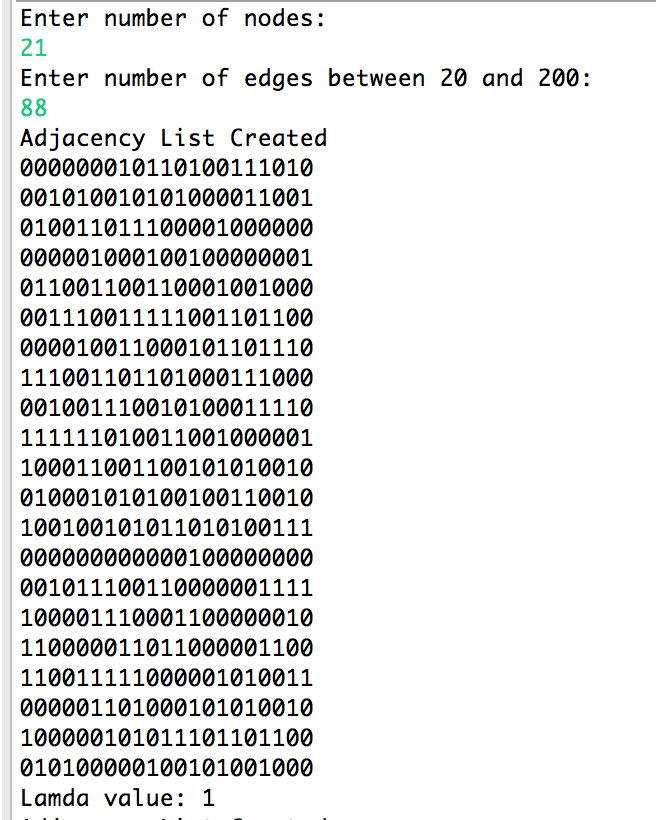
****

**Graph 5 Adjacency List:**

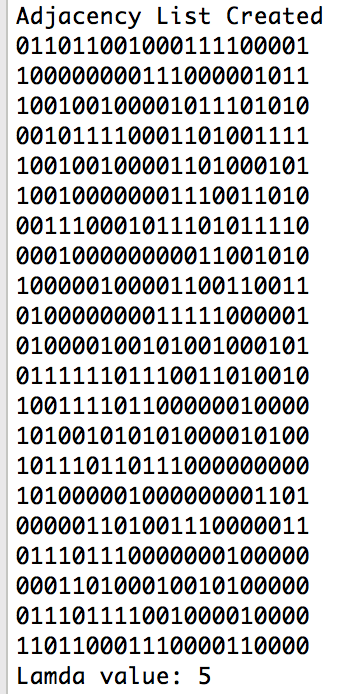
****

**m=88**

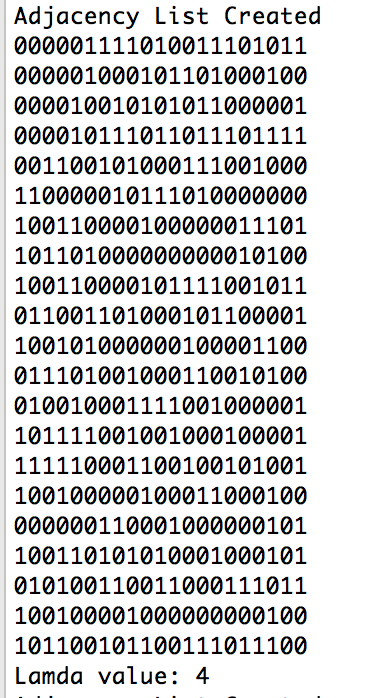
**Graph 1 Adjacency List:**

****

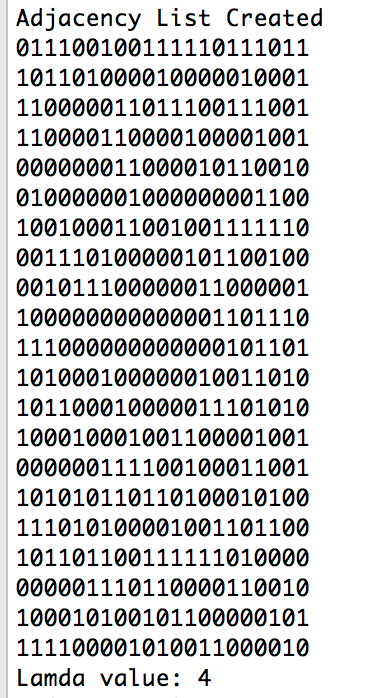
**Graph 2 Adjacency List:**

****

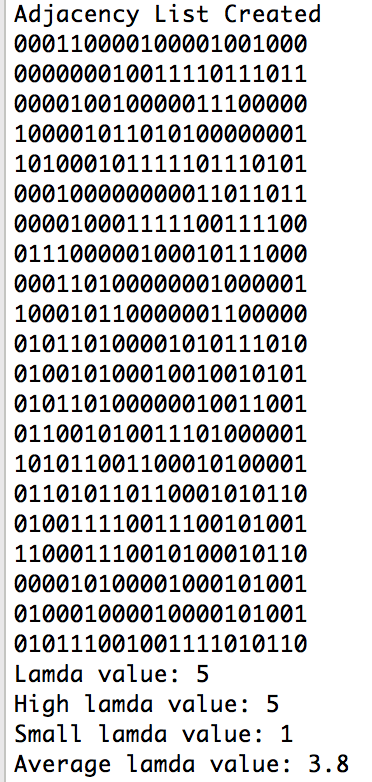
**Graph 3 Adjacency List:**

****

**Graph 4 Adjacency List:**

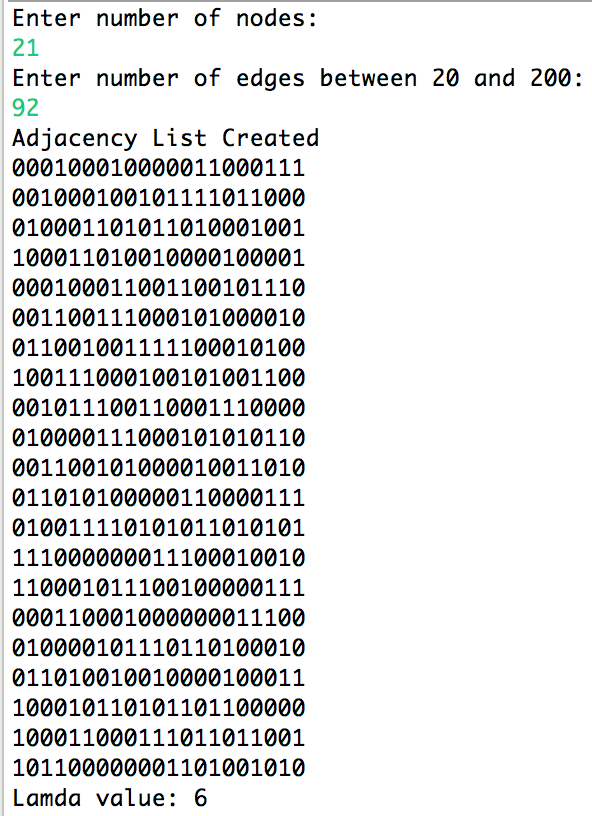
****

**Graph 5 Adjacency List:**

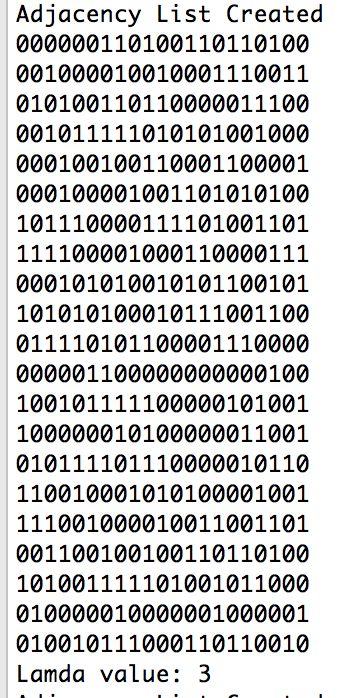
****

**m=92**

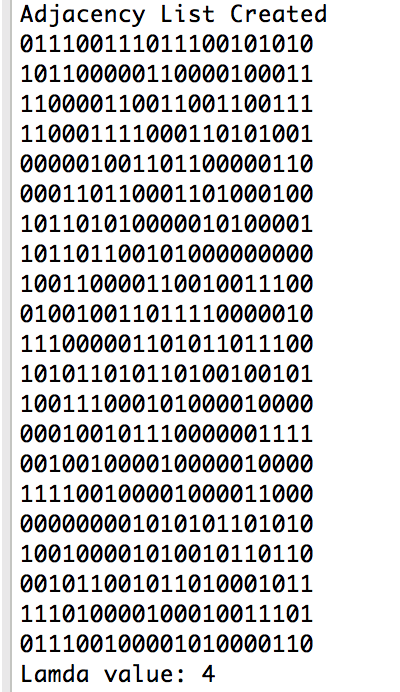
**Graph 1 Adjacency List:**

****

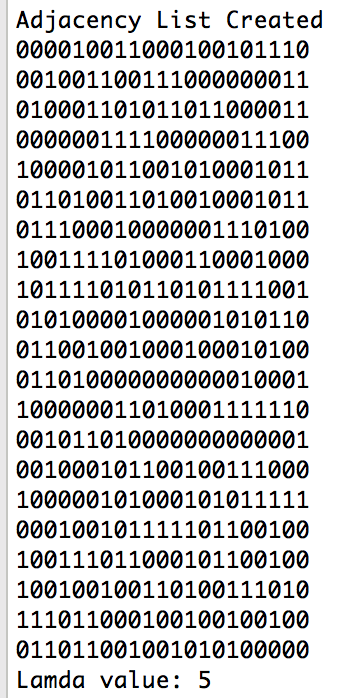
**Graph 2 Adjacency List:**

****

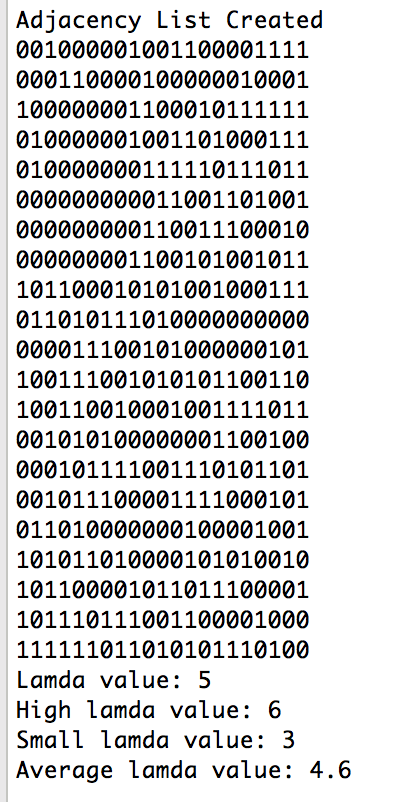
**Graph 3 Adjacency List:**

****

**Graph 4 Adjacency List:**

****

**Graph 5 Adjacency List:**

****

1. **Graphs**

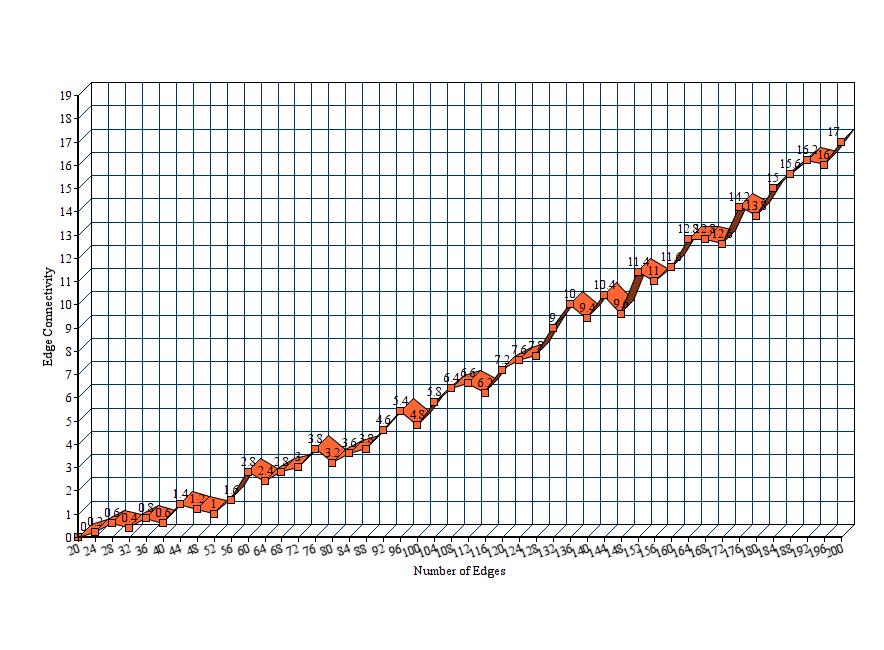
n – number of nodes = 21

m- number of edges = 20, 24, 28, 32… 200

**Graph 1 Data:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SNo** | **Number of edges** | **Edge Connectivity Or Lamda value** | | | | | | | |
| **Graph 1** | **Graph 2** | **Graph 3** | **Graph 4** | **Graph 5** | **Small Value** | **Large Value** | **Average Value** |
| 1 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 24 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0.2 |
| 3 | 28 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0.6 |
| 4 | 32 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0.4 |
| 5 | 36 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0.8 |
| 6 | 40 | 2 | 1 | 0 | 0 | 0 | 0 | 2 | 0.6 |
| 7 | 44 | 2 | 1 | 1 | 2 | 1 | 1 | 2 | 1.4 |
| 8 | 48 | 1 | 0 | 1 | 2 | 2 | 0 | 2 | 1.2 |
| 9 | 52 | 1 | 1 | 1 | 2 | 0 | 0 | 2 | 1.0 |
| 10 | 56 | 3 | 2 | 1 | 2 | 0 | 0 | 3 | 1.6 |
| 11 | 60 | 3 | 3 | 3 | 2 | 3 | 2 | 3 | 2.8 |
| 12 | 64 | 1 | 2 | 3 | 3 | 3 | 1 | 3 | 2.4 |
| 13 | 68 | 2 | 4 | 2 | 3 | 3 | 2 | 4 | 2.8 |
| 14 | 72 | 3 | 2 | 2 | 5 | 3 | 2 | 5 | 3.0 |
| 15 | 76 | 4 | 4 | 4 | 4 | 3 | 3 | 4 | 3.8 |
| 16 | 80 | 4 | 3 | 3 | 3 | 3 | 3 | 4 | 3.2 |
| 17 | 84 | 4 | 4 | 4 | 2 | 4 | 2 | 4 | 3.6 |
| 18 | 88 | 1 | 5 | 4 | 4 | 5 | 1 | 5 | 3.8 |
| 19 | 92 | 6 | 3 | 4 | 5 | 5 | 3 | 6 | 4.6 |
| 20 | 96 | 6 | 5 | 5 | 4 | 7 | 4 | 7 | 5.4 |
| 21 | 100 | 4 | 6 | 4 | 4 | 6 | 4 | 6 | 4.8 |
| 22 | 104 | 7 | 7 | 6 | 5 | 4 | 4 | 7 | 5.8 |
| 23 | 108 | 5 | 6 | 6 | 8 | 7 | 5 | 8 | 6.4 |
| 24 | 112 | 5 | 7 | 8 | 6 | 7 | 5 | 8 | 6.6 |
| 25 | 116 | 4 | 6 | 7 | 8 | 6 | 4 | 8 | 6.2 |
| 26 | 120 | 6 | 8 | 7 | 9 | 6 | 6 | 9 | 7.2 |
| 27 | 124 | 8 | 8 | 6 | 9 | 7 | 6 | 9 | 7.6 |
| 28 | 128 | 6 | 8 | 8 | 9 | 8 | 6 | 9 | 7.8 |
| 29 | 132 | 8 | 9 | 9 | 10 | 9 | 8 | 10 | 9.0 |
| 30 | 136 | 9 | 10 | 9 | 11 | 11 | 9 | 11 | 10.0 |
| 31 | 140 | 11 | 9 | 10 | 9 | 8 | 8 | 11 | 9.4 |
| 32 | 144 | 10 | 11 | 11 | 10 | 10 | 10 | 11 | 10.4 |
| 33 | 148 | 11 | 8 | 9 | 9 | 11 | 8 | 11 | 9.6 |
| 34 | 152 | 12 | 12 | 12 | 11 | 10 | 10 | 12 | 11.4 |
| 35 | 156 | 11 | 12 | 11 | 10 | 11 | 10 | 12 | 11.0 |
| 36 | 160 | 13 | 13 | 10 | 11 | 11 | 10 | 13 | 11.6 |
| 37 | 164 | 13 | 13 | 13 | 13 | 12 | 12 | 13 | 12.8 |
| 38 | 168 | 13 | 13 | 13 | 12 | 13 | 12 | 13 | 12.8 |
| 39 | 172 | 13 | 13 | 14 | 11 | 12 | 11 | 14 | 12.6 |
| 40 | 176 | 14 | 14 | 14 | 15 | 14 | 14 | 15 | 14.2 |
| 41 | 180 | 13 | 13 | 14 | 15 | 14 | 13 | 15 | 13.8 |
| 42 | 184 | 15 | 15 | 14 | 16 | 15 | 14 | 16 | 15.0 |
| 43 | 188 | 16 | 15 | 16 | 16 | 15 | 15 | 16 | 15.6 |
| 44 | 192 | 16 | 16 | 17 | 16 | 16 | 16 | 17 | 16.2 |
| 45 | 196 | 14 | 16 | 17 | 17 | 16 | 14 | 17 | 16.0 |
| 46 | 200 | 17 | 17 | 16 | 18 | 17 | 16 | 18 | 17.0 |

**Graph 1: Number of edges Vs Edge connectivity**

****

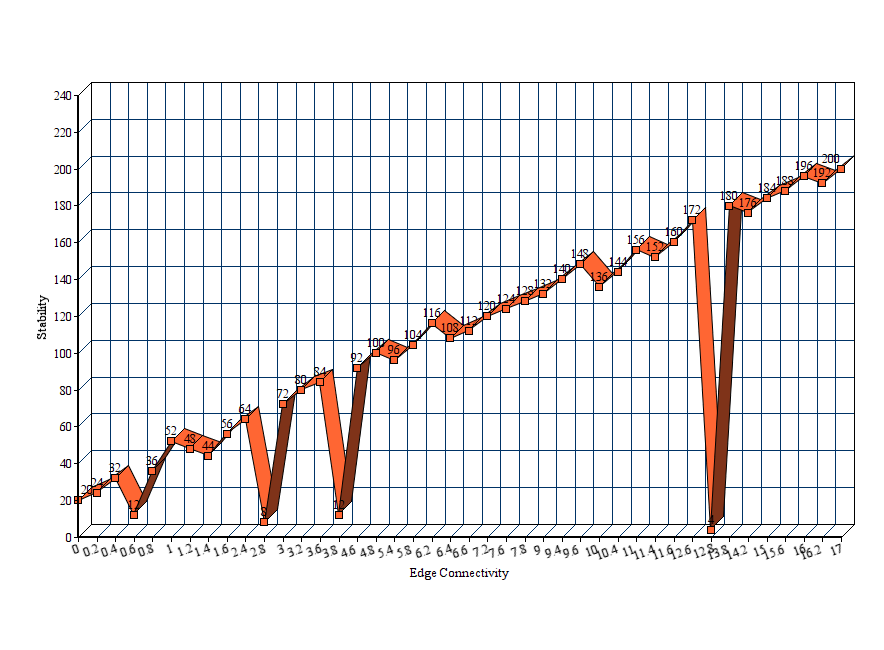
**Note: I chose average value of lamda(Edge connectivity) vs number of edges.**

**Graph 2 Data:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sno** | **Small Value m** | **Large Value m** | **Average Lamda** | **Stability** |
| 1 | 0 | 20 | 0 | 20 |
| 2 | 0 | 24 | 0.2 | 24 |
| 3 | 0 | 32 | 0.4 | 32 |
| 4 | 28 | 40 | 0.6 | 12 |
| 5 | 0 | 36 | 0.8 | 36 |
| 6 | 0 | 52 | 1 | 52 |
| 7 | 0 | 48 | 1.2 | 48 |
| 8 | 0 | 44 | 1.4 | 44 |
| 9 | 0 | 56 | 1.6 | 56 |
| 10 | 0 | 64 | 2.4 | 64 |
| 11 | 60 | 68 | 2.8 | 8 |
| 12 | 0 | 72 | 3 | 72 |
| 13 | 0 | 80 | 3.2 | 80 |
| 14 | 0 | 84 | 3.6 | 84 |
| 15 | 76 | 88 | 3.8 | 12 |
| 16 | 0 | 92 | 4.6 | 92 |
| 17 | 0 | 100 | 4.8 | 100 |
| 18 | 0 | 96 | 5.4 | 96 |
| 19 | 0 | 104 | 5.8 | 104 |
| 20 | 0 | 116 | 6.2 | 116 |
| 21 | 0 | 108 | 6.4 | 108 |
| 22 | 0 | 112 | 6.6 | 112 |
| 23 | 0 | 120 | 7.2 | 120 |
| 24 | 0 | 124 | 7.6 | 124 |
| 25 | 0 | 128 | 7.8 | 128 |
| 26 | 0 | 132 | 9 | 132 |
| 27 | 0 | 140 | 9.4 | 140 |
| 28 | 0 | 148 | 9.6 | 148 |
| 29 | 0 | 136 | 10 | 136 |
| 30 | 0 | 144 | 10.4 | 144 |
| 31 | 0 | 156 | 11 | 156 |
| 32 | 0 | 152 | 11.4 | 152 |
| 33 | 0 | 160 | 11.6 | 160 |
| 34 | 0 | 172 | 12.6 | 172 |
| 35 | 164 | 168 | 12.8 | 4 |
| 36 | 0 | 180 | 13.8 | 180 |
| 37 | 0 | 176 | 14.2 | 176 |
| 38 | 0 | 184 | 15 | 184 |
| 39 | 0 | 188 | 15.6 | 188 |
| 40 | 0 | 196 | 16 | 196 |
| 41 | 0 | 192 | 16.2 | 192 |
| 42 | 0 | 200 | 17 | 200 |

**Note**: If for a particular average value of lamda there are only one value of number of edges, then that has chosen as large value and the small value is taken as 0. As the average lamda value is not rounded, there are not many edges with same value of lamda.

**Graph 2: Stability Vs Edge connectivity(Average value)**

****

1. **Observation and Conclusion**

* As the number of edges increases edge connectivity or lamda value increases. Even though for small differences of edges it seems to varies due to randomness of edge values. But considering overall graph lamda value increases when number of edges increasing.
* For small differences in number of edges lamda value may increase or decrease with small values. This is because of the random selection of edges each time.
* Stability is considered as the difference between large and small value of number of edges for a particular lamda value. From the stability vs lamda value graph it is clear that the stability is not depending on lamda values.

1. **Appendix : Source code**

**Driver.java**

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.Random;

**import** java.util.Scanner;

**public** **class** Driver {

**public** **static** **void** main(String args[]) **throws** FileNotFoundException{

System.***out***.println("Enter number of nodes: ");

Scanner in;

**if**(args.length>0){

File inputFile = **new** File(args[0]);

in = **new** Scanner(inputFile);

}

**else**{

in = **new** Scanner(System.***in***);

}

**int** n;

**int** m1;

n=in.nextInt();

**if**(n<0){

n=21;

}

**if**(n%2==1){

m1=n\*(n-1)-n+1;

m1=m1/2;

}

**else**{

m1=n\*(n-1)-n;

m1=m1/2;

}

**int** n1=n-1;

System.***out***.println("Enter number of edges between "+n1+" and "+m1+": ");

**int** m;

m=in.nextInt();

**if**(m<n-1 || m>m1){

System.***out***.println("Invalid input, taking a random number between "+n1+" and "+m1+": ");

Random r = **new** Random();

**int** Low = n1;

**int** High = m1;

m=r.nextInt(High-Low) + Low;

}

**int**[] lamda = **new** **int**[5];

**for**(**int** i=0;i<5;i++){

lamda[i]=Graph.*adjacencyListCreation*(n, m);

System.***out***.println("Lamda value: "+lamda[i]);

}

**int** highlamda=0,smalllamda=Integer.***MAX\_VALUE***;

**float** avglamda=0;

**for**(**int** i=0;i<lamda.length;i++){

**if**(highlamda<lamda[i]){

highlamda=lamda[i];

}

**if**(smalllamda>lamda[i]){

smalllamda=lamda[i];

}

avglamda+=lamda[i];

}

avglamda=avglamda/lamda.length;

System.***out***.println("High lamda value: "+highlamda);

System.***out***.println("Small lamda value: "+smalllamda);

System.***out***.println("Average lamda value: "+avglamda);

}

}

**Graph.java**

**import** java.util.Random;

**public** **class** Graph {

**public** **static** **int** adjacencyListCreation(**int** n, **int** m){

Random r = **new** Random();

**int** Low=0;

**int** High=n;

//int count=0;

**int**[][] adj = **new** **int**[n][n];

**for**(**int** i=0;i<m;i++){

**int** from=r.nextInt(High-Low)+Low;

**int** to=r.nextInt(High-Low)+Low;

**if**(to==from){

i--;

}

**else** **if**(adj[to][from]==1){

i--;

}

**else**{

adj[to][from]=1;

adj[from][to]=1;

//count++;

}

**if**(i==-1) {

i++;

}

}

**int** flag=0;

System.***out***.println("Adjacency List Created");

**for**(**int** i=0;i<n;i++){

**int** c=0;

**for**(**int** j=0;j<n;j++){

System.***out***.print(adj[i][j]);

c+=adj[i][j];

}

**if**(c==0){

flag=1;

}

System.***out***.println();

}

//System.out.println("Count:"+count);

**if**(flag==1){

**return** 0;

}

**else**{

NagamochiIbarakiAlg Ni=**new** NagamochiIbarakiAlg(0);

**int** res=Ni.*NIAlg*(adj,n);

//System.out.println("Result:"+res);

**return** res;

}

}

}

**NagamochiIbarakiAlg.java**

**public** **class** NagamochiIbarakiAlg {

**private** **static** **int** *res*;

**public** NagamochiIbarakiAlg(**int** res) {

**this**.*res*=res;

}

**public** **static** **int** NIAlg(**int**[][] adj,**int** n){

**if**(n==2){

**int** r=0;

**int** i=0;

**for**(**int** j=0;j<n;j++){

r+=adj[i][j];

}

**return** r;

}

**int** flag=0;

//System.out.println("Adjacency List Created");

**for**(**int** i=0;i<n;i++){

**int** c=0;

**for**(**int** j=0;j<n;j++){

//System.out.print(adj[i][j]);

c+=adj[i][j];

}

**if**(c==0){

flag=1;

}

//System.out.println();

}

**if**(flag==1){

**return** 0;

}

**int**[] a=*MAOrdering*(adj,n);

**int**[][] adj1=*Merge*(adj,a[0],a[1],n); //Merge completed n tested

*res*=*Min*(*NIAlg*(adj1,n-1),a[2]);

**return**(*res*);

//System.out.println(a);

//return 0;

}

**public** **static** **int** Min(**int** n1, **int** n2){

**if**(n1<n2){

**return** n1;

}

**else**{

**return** n2;

}

}

**public** **static** **int**[] MAOrdering(**int**[][] adj,**int** node){

**int**[] a = **new** **int**[3];

**int**[] q=**new** **int**[node];

**int** i=0;

q[i]=0;

**while**(i<node-1){

//int[] c=null;

**int**[] c=*count*(adj,q,node);

**int** m=*max*(c,node);

q[++i]=m;

}

a[0]=q[node-2];

a[1]=q[node-1];

i=a[1];

**for**(**int** j=0;j<node;j++){

a[2]+=adj[i][j];

}

**return** a;

}

**public** **static** **int**[] count(**int**[][] adj,**int**[] q,**int** node){

**int**[] c=**new** **int**[node];

**for**(**int** j=0;j<node;j++)

{

**int** flag=0;

**for**(**int** i=0;i<q.length;i++){

**if**(q[i]==j){

flag=1;

**break**;

}

}

**if**(flag==1)

**continue**;

**else**{

**for**(**int** i=0;i<q.length;i++){

c[j]+=adj[i][j];

}

}

//c[j]+=adj;

}

**return** c;

}

**public** **static** **int** max(**int**[] c,**int** node){

**int** a=c[0];

**int** ret=0;

**for**(**int** i=1;i<node;i++){

**if**(a<c[i]){

a=c[i];

ret=i;

}

}

**return** ret;

}

**public** **static** **int**[][] Merge(**int**[][] adj,**int** x,**int** y, **int** node){

**int**[][] a =**new** **int**[node-1][node-1];

**for**(**int** i=0;i<node;i++){

**if**(i==x){

**for**(**int** j=0;j<node;j++){

**int** k=adj[x][j];

adj[x][j]=k+adj[y][j];

adj[y][j]=k+adj[y][j];

}

**break**;

}

}

/\*

System.out.println("Adjacency test");

for(int i=0;i<node;i++){

for(int j=0;j<node;j++){

System.out.print(adj[i][j]);

}

System.out.println();

}

\*/

**for**(**int** j=0;j<node;j++){

**if**(j==x){

**for**(**int** i=0;i<node;i++){

**int** k=adj[i][x];

adj[i][x]=k+adj[i][y];

adj[i][y]=k+adj[i][y];

}

}

}

/\*

System.out.println("Adjacency test2");

for(int i=0;i<node;i++){

for(int j=0;j<node;j++){

System.out.print(adj[i][j]);

}

System.out.println();

}

\*/

//new adjacency list creation

**int** p=0,q;

**for**(**int** i=0;i<node;i++){

q=0;

**if**(i==y)

**continue**;

**for**(**int** j=0;j<node;j++){

**if**(j==y)

**continue**;

**if**(p==q)

a[p][q++]=0;

**else**

a[p][q++]=adj[i][j];

}

p++;

}

/\*

int flag=0;

//System.out.println("New Adjacency test");

for(int i=0;i<node-1;i++){

for(int j=0;j<node-1;j++){

//System.out.print(a[i][j]);

}

//System.out.println();

}

\*/

**return** a;

}

}

1. **Readme**

**Step 1:** Keep the programs in different files as Driver.java, NagamochiIbarakiAlg.java and Graph.java

**Step 2:** Compile all the files. Javac \*.java

**Step 3:** Run the driver.java file.

**Step 4:** It will prompt for inputs. Give number of nodes and number of edges as inputs.

**Step 5:** Program will output smallest lamda value, largest lamda value, and average lamda value. It will also output adjacency list of the input graphs.

**Step 6:** Repeat the steps for 4 different values of m.

1. **References**

* Lecture notes Professor Andras Farago