Project 2

Nagamochi-Ibaraki Algorithm

CS 6385

Vyoma Trivedi (vmt160030)

Table of Content

Objective

- Create an implementation of the Nagamochi-Ibaraki algorithm for finding a minimum cut in an undirected graph.
- Run the program on randomly generated examples. Let the number of nodes be fixed at n = 21, while the number m of edges will vary between 20 and 200, in steps of 4. For any such value of m, the program creates 5 graphs with n = 21 nodes and m edges. The actual edges are selected randomly. Parallel edges and self-loops are not allowed in the original graph generation. Note, however, that the Nagamochi-Ibaraki algorithm allows parallel edges in its internal working, as they may arise due to the merging of nodes.
- Experiment with your random graph examples to find an experimental connection between the number of edges in the graph and its connectivity $\lambda(G)$. (If the graph happens to be disconnected, then take $\lambda(G) = 0$.) Show the found relationship graphically in a diagram, exhibiting $\lambda(G)$ as a function of m, while keeping n = 21 fixed. Since the edges are selected randomly, therefore, we reduce the effect of randomness in the following way: run 5 independent experiments for every value of m, one with each generated example for this m, and average out the results. (This is why 5 graphs were created for every m.)
- For every connectivity value $\lambda = \lambda(G)$ that occurred in the experiments, record the largest and smallest number of edges with which this 1 λ value occurred. Let us call their difference the stability of λ, and let us denote it by $s(\lambda)$. Show in a diagram how $s(\lambda)$ depends on λ, based on your experiments.

Algorithm:

Nagamochi-Ibaraki algorithm makes use the maximum adjacency (MA) ordering of the vertices while computing the edge connectivity.

Consider the following graph for our reference.

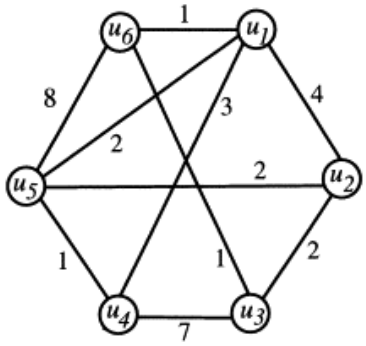


Figure 1: Sample graph

MA Ordering:

An ordering $u_1, u_2, \ldots, u_n$ of vertices is called an MA ordering if an arbitrary vertex is chosen as u_1 , and after choosing the first i vertices $u_1, \ldots, u_i$, the $(i + 1)^{th}$ vertex u_{i+1} is chosen from the vertices v that have the largest number of edges between $u_1, \ldots, u_i$ and v.

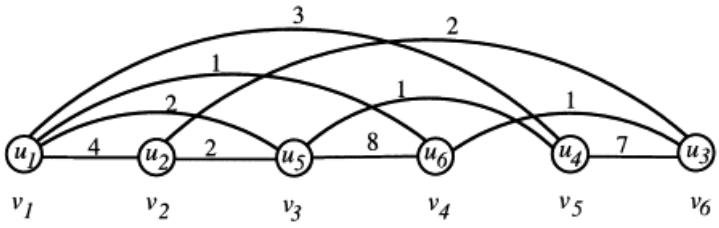


Figure 2: MA ordering for the graph in figure 1

Theorem:

Let G_{xy} be the graph obtained from G by contracting nodes x and y. In this operation we omit the possibility of arising loop (if x and y are connected in G), but keep the parallel edges. Then for any two nodes x and y, the following holds: **λ (G) = min λ(x, y), λ (G_{xy})**

Nagamochi and Ibaraki has proved the following theorem which helps us to use MA ordering in computing $\lambda(x, y)$ and hence use the above theorem.

Theorem: In any MA ordering $v_1, \ldots, v_n$ of the nodes

$$\lambda (v_{n-1}, v_n) = d(v_n)$$

holds, where d (.) denotes the degree of the node.

Pseudo code:

1. Initialize the graph of size 21*21 to 0.
   Method: initialize ()

2. Randomly generate the edges using the Random function.
   Method: graph ()

3. Use BFS to see if the graph is connected.
   Method: BFS ()

   - If the graph is disconnected then the minimum cut is 0.
   - Else we find the minimum cut using the Nagamochi-Ibaraki Algorithm.
     It starts the MA ordering from node 0 and then it generates the contracted graph and continues through all the nodes till we find the minimum cut.
     Method: find_min_cut ()

Algorithm 1 Pseudocode for Nagamochi-Ibaraki algorithm

INPUT : $G := (V, E)$ //possibly a multi-graph
OUTPUT : $\lambda(G)$
Initialization:
G= MULTIGRAPH-TO-WEIGHTED-SIMPLE-GRAPH(G)
Nagamochi-Ibaraki-Recursive (G)
if $|V| == 2$ **then**
  return $weight(E_G(v_1, v_2))$
end if
vertList:=MA-ORDER(G)
$(x, y) := vertList.lastTwo()$
$\lambda_G(x, y) := d_G(y)$
G-contracted:=CONTRACT(G,x,y)
return $\min(\lambda_G(x, y)$, NAGAMOCHI-IBARAKI-RECURSIVE(G-contracted))

Results : (Number of Nodes = 21)

Number of Edges	Connectivity
20	1
24	2
28	1
32	4
36	2
40	3
44	4
48	4
52	5
56	3
60	4
64	5
68	6
72	6
76	6
80	7
84	8
88	8
92	9
96	7
100	8
104	11
108	9
112	10
116	8
120	11
124	10
128	10
132	12

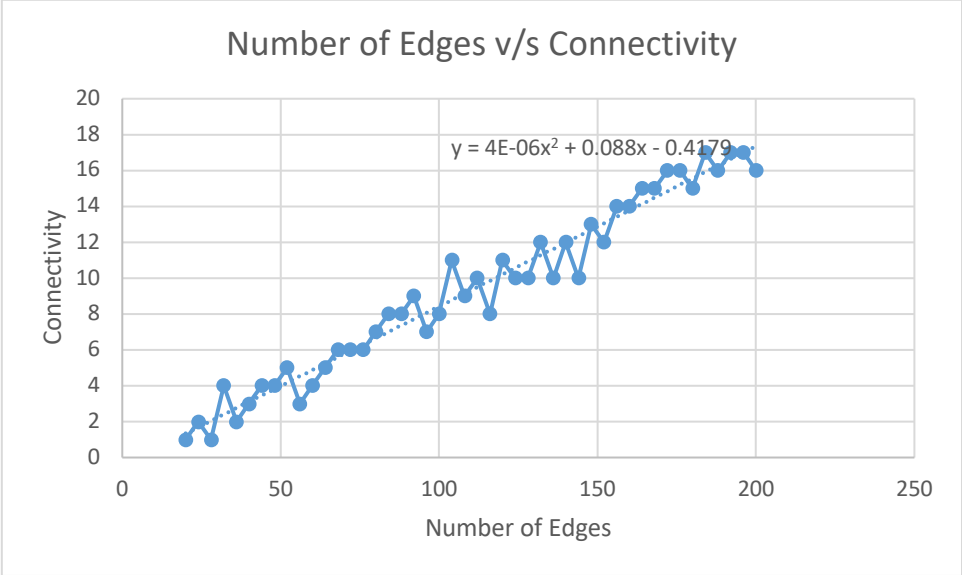
136	10
140	12
144	10
148	13
152	12
156	14
160	14
164	15
168	15
172	16
176	16
180	15
184	17
188	16
192	17
196	17
200	16

Table 1: The experimental values obtained for number of edges and connectivity

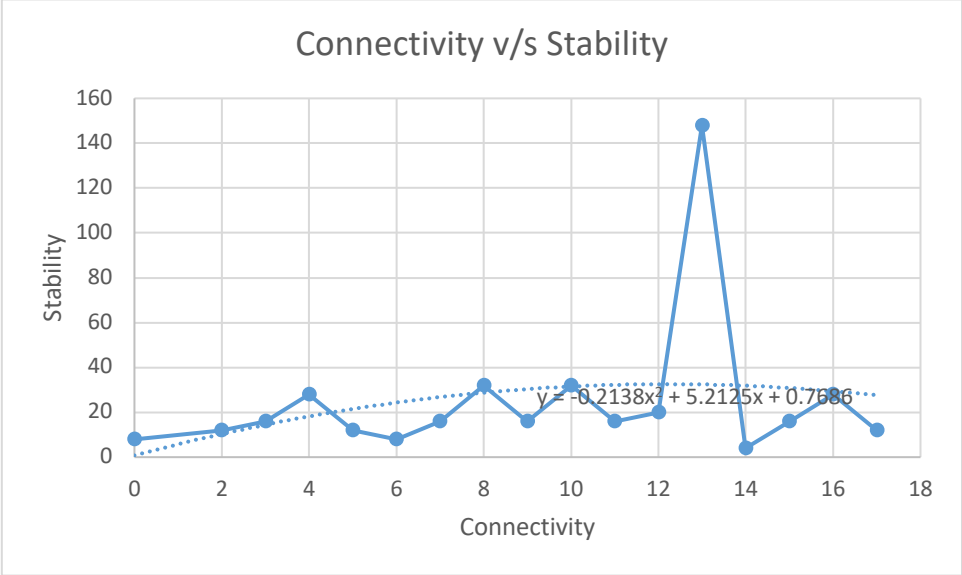
Connectivity	Stability
0	8
2	12
3	16
4	28
5	12
6	8
7	16
8	32
9	16
10	32
11	16
12	20
13	148
14	4
15	16
16	28
17	12

Table 2: The experimental values obtained for connectivity and stability

Charts



As we can see from the graph there is no specific relation between number of edges and Connectivity but taking a general view we can say that Connectivity increases with increase in number of edges. This is because of the Random function we used for generating the graph.



Generally the stability increases with increase in connectivity but for some values it decreases with the connectivity. This is because of the Random function we used for generating the graph.

Conclusion:

There is no linear relation between (number of edges, connectivity) and (connectivity, stability). Some quadratic relation might exist that can be found by solving some complex equations and they are shown in the graph.

Appendix

```
//An implementation of the Nagamochi-Ibaraki algorithm
//for finding a minimum cut in an undirected graph.

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package nagamochi_ibaraki;

import java.nio.file.FileVisitResult;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;

/**
 *
 * @author VYOMA
 */
public class Nagamochi_Ibaraki {

    /**
     * @param args the command line arguments
     */
```

```
    //global variables

    int nodes = 21,tnodes=21;
    int g[][] = new int[21][21];
    int temp[][]=new int[21][21];
    int MA_order[] = new int[21];
    int min_cut[] = new int[21];
    public Queue<Integer> q = new LinkedList<Integer>();
    boolean flag=false;

//initialize both the graph matrices

    public void initialize(){
       for(int i=0;i<nodes;i++){
          for(int j=0;j<nodes;j++){
             g[i][j]=0;
             temp[i][j]=0;
          }
          MA_order[i]=Integer.MAX_VALUE;
          min_cut[i] = 0;
       }
    }

    //randomly generating the graph of 21 nodes

    public void graph(int a){
//        int i=0;
//        while(i<a){
//           Random r = new Random();
//           int b=r.nextInt(nodes);
```

```
//            int c=r.nextInt(nodes);
//            if(b<c){
//               g[b][c]++;
//               temp[b][c]++;
//               i++;
//            }
//        }
        int i=0;
        while(i<a){
           Random r = new Random();

           int flag = 0;
           while(flag!=1){
              int b=r.nextInt(nodes);
              int c=r.nextInt(nodes);
              if(g[b][c]==0 && g[c][b]!=1){
              g[b][c]++;
              temp[b][c]++;
              i++;
              flag = 1;
              }

           }
        }
    }

    //check if the graph is disconnected using BFS

    public boolean BFS(int start){
```

```
int el,tem,count=0;

int visited[]=new int[21];
visited[start] = 1;
q.add(start);
while(!q.isEmpty()){
   el=q.remove();
   tem=1;
   while(tem < nodes){
      if(g[start][tem] == 1 && visited[tem] == 0){
         q.add(tem);
         visited[tem] = 1;
      }
      tem++;
   }
}
for(int j=0;j<nodes;j++){
   if(visited[j]==1)
      count++;

}
if(count==nodes)
   return true;
else
{
   return false;

}
```

```
}

    //calculate the MA order and get mincut value

    public int find_min_cut(int a){
        MA_order[0]=0;
        int vertex=1,vertex_checked=0,max_connectivity=0,min_vertex=0;
        int contracted[]=new int[21];
        for(int i=0;i<nodes;i++)
            contracted[i]=0;



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

                    while(vertex < tnodes){
                        for(int j=0;j<vertex;j++){
                            for(int k=0;k<nodes;k++){
                                for(int m=0;m<vertex;m++){
                                    if(MA_order[m]==k)
                                    {
                                        vertex_checked=1;
                                        break;
                                    }
                                }
                                if(vertex_checked==1){
                                    vertex_checked=0;
                                    continue;
                                }
```

```
            else{
                if(((g[k][MA_order[j]])!=0)||((g[MA_order[j]][k])!=0)){
                    if(k < MA_order[i])
                        contracted[k] += g[k][MA_order[j]] ;
                    else
                        contracted[k] += g[MA_order[j]][k] ;
                    if(contracted[k] >= contracted[max_connectivity])
                        max_connectivity = k ;

                }

            }
        }

    }
    MA_order[vertex]=max_connectivity;
    vertex++;
    for(int t=0;t<nodes;t++)
        contracted[t]=0;
}
for(int j=0;j<nodes;j++){
    min_cut[i] = min_cut[i]+g[j][MA_order[vertex-1]];
    min_cut[i] = min_cut[i]+g[MA_order[vertex-1]][j];
    contracted[j]=0;

}
//selecting the minimum vertex

if(min_cut[i]<min_cut[min_vertex])
    min_vertex=i;
```

```
                for(int j=0;j<nodes;j++){
                   if(j < MA_order[vertex-1]){
                      g[j][MA_order[vertex-2]] = g[j][MA_order[vertex-2]] +
g[j][MA_order[vertex-1]];
                      g[j][MA_order[vertex-1]]=0;
                   }
                   else if(vertex > 1){
                      g[MA_order[vertex-2]][j] = g[MA_order[vertex-2]][j] +
g[MA_order[vertex-1]][j];
                      g[MA_order[vertex-1]][j]=0;
                      }
                   }
                if(vertex > 1){
                   g[MA_order[vertex-2]][MA_order[vertex-2]]=0;
                   g[MA_order[vertex-2]][MA_order[vertex-1]]=0;
                   g[MA_order[vertex-1]][MA_order[vertex-2]]=0;
                }
                g[MA_order[vertex-1]][MA_order[vertex-1]]=0;
                for(int j=0;j<nodes;j++)
                   MA_order[j]=j;
                MA_order[0]=0;
                max_connectivity=0;
                vertex=1;
                tnodes=tnodes-1;
             }
             return min_cut[min_vertex];
//       }
//
//       else
//          return 0;
```

```
    }
    public static void main(String[] args) {
        // TODO code application logic here
        Nagamochi_Ibaraki nm=new Nagamochi_Ibaraki();
        int edge_start=20,average_edge=0,minimum_cut,critical_edge,sum=0;
        ArrayList<Integer> al=new ArrayList<Integer>();
        ArrayList<Integer> min=new ArrayList<Integer>();
        ArrayList<Integer> max=new ArrayList<Integer>();
        Boolean isConnected,flag=true;
        System.out.println("edge"+"\t"+"minimum_cut");

        while(edge_start <= 200){

            sum=0;
            for(int p=0;p<5;p++){
            nm.initialize();
            nm.graph(edge_start);
            //check if the graph is connected

            for(int i=0;i<nm.nodes;i++){
                isConnected=nm.BFS(i);
                if(!isConnected){
//                  minimum_cut=0;
                    flag=false;

                    break;
                }

            }
            if(flag=true)
```

```
        sum=sum+nm.find_min_cut(edge_start);
    else
        sum=sum+0;
    }
    for(int h=0;h<nm.nodes;h++){
        max.add(0);
    }
    int pos=0;
    minimum_cut=sum/5;
    if(!al.contains(minimum_cut)){
        al.add(minimum_cut);
        min.add(edge_start);
    }
    else{
        for(int l=0;l<al.size();l++)
        {
            if(al.get(l)==minimum_cut)
                pos=l;
        }
        max.set(pos, edge_start);
    }

    System.out.println(edge_start+"\t\t"+minimum_cut);
    edge_start=edge_start+4;
}
//int output[][]=new int [][2];
for(int i=0;i<al.size();i++){
    System.out.print("Mincut:"+al.get(i)+"  ");
    System.out.println("");
    System.out.println("Stability: "+Math.abs(min.get(i)-max.get(i)));
```

```
    }
    }

}
```

References:

1) Lecture Notes
2) http://www.sciencedirect.com/science/article/pii/S0166218X01003493
3) Wikipedia
4) Sanfoundary - BFS