Project 2

Algorithmic Aspects of Telecommunication Networks (CS 6385)

Nagamochi-Ibaraki Algorithm

Rajan Jhaveri
rjj160330

Table of Content

OBJECTIVE

- Create an implementation of the Nagamochi-Ibaraki algorithm for finding a minimum cut in an undirected graph, and experiment with it. Explain the implementation of the algorithm and pseudo code for the description, with sufficient comments to make it readable and understandable by a human.
- A computer program that implements the algorithm.
- 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.
- For every connectivity value $\lambda = \lambda(G)$ that occurred in the experiments, record the largest and smallest number of edges with which this 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.
- A short explanation why the functions found in items 4 and 5 look the way they look.
- A readme file that provides description on how to run the program.

ALGORITHM

- Nagamochi and Ibaraki developed an algorithm to find the minimum cut in an entire graph by using the nodes and the connectivity between them.
- Nagamochi-Ibaraki algorithm makes use of the maximum adjacency (MA) ordering of the vertices while computing the edge connectivity. So, first we should clearly understand the definition of Connectivity and Maximum Adjacency Ordering.

Connectivity

- A graph is connected when there is a path between every pair of vertices. In a connected graph, there are no unreachable vertices. A graph that is not connected is disconnected.

- Edge connectivity between nodes x and y, $\lambda(x,y)$, is the minimum number of edges that is needed to be deleted to disconnect x and y, where $x \neq y$. Edge connectivity of a graph, $\lambda(G)$, is the minimum number of edges that is needed to be deleted to disconnect the graph. Note that there is the following relation between $\lambda(x,y)$ and $\lambda(G)$,

$$\lambda(G) = \min \lambda(x, y)$$

Maximum Adjacency 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.

- Consider the following example

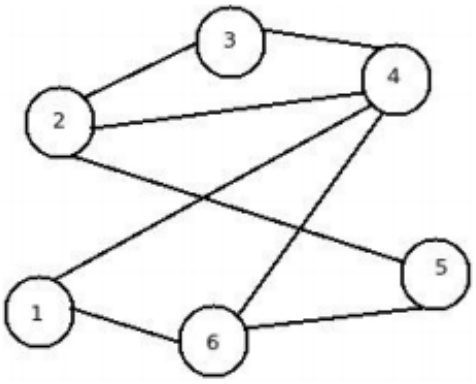


Figure 1: Sample graph

- **MA** Ordering for the sample graph will be as follows :

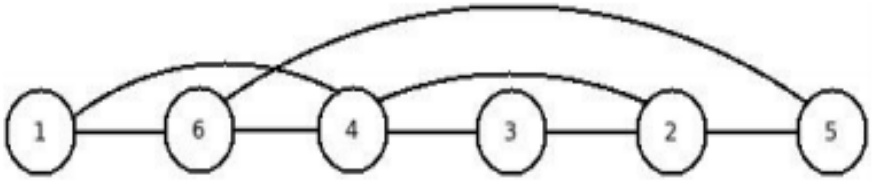


Figure 2: MA ordering for the graph in figure 1

PSEUDOCODE

- Create graph G with n nodes and m edges.
- Initialize all the edges to 0.
- Randomly assign weight 1 to the number of edges decided, here 20 to 200 in steps incrementing by 4.
- Using the graph created find the minimum cut of the graph.
- We check if the graph is connected or disconnected using DFS.
- The minimum cut is 0 if graph is disconnected.
- While number of nodes > 2
    - If G is connected then Maximum Adjacency Ordering of $v_1, v_2, \ldots, v_n$ of G is created.
    - Edge connectivity is calculated as degree of v_n.
    - Nodes v_{n-1} and v_n are contracted into v_n and v_n is added back.
    - $\lambda(G)$ is updated each time 2 nodes in MAO are contracted. If edge connectivity is smaller, the minimum of both the values is taken.
    - Minimum cut is, the minimum connectivity calculated so far.
    - Make the necessary changes and calculate the minimum cut for new graph.
    - Method used will be get_min_cut
- Repeat procedure for all the edges and find total number of critical edges

RESULTS

- NUMBER OF NODES =21.
- Values of Number of Edges and their corresponding Connectivity are shown in the table :

Number of Edges	Connectivity
20	0
24	0
28	3
32	3
36	2
40	4
44	3
48	4
52	5
56	5
60	5
64	5
68	5
72	7
76	7
80	6
84	8
88	6
92	7
96	9
100	10
104	9
108	9
112	11
116	10
120	10
124	11
128	11
132	13
136	12
140	13
144	13
148	14
152	12

156	14
160	15
164	13
168	14
172	15
176	17
180	15
184	16
188	16
192	17
196	17
200	17

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

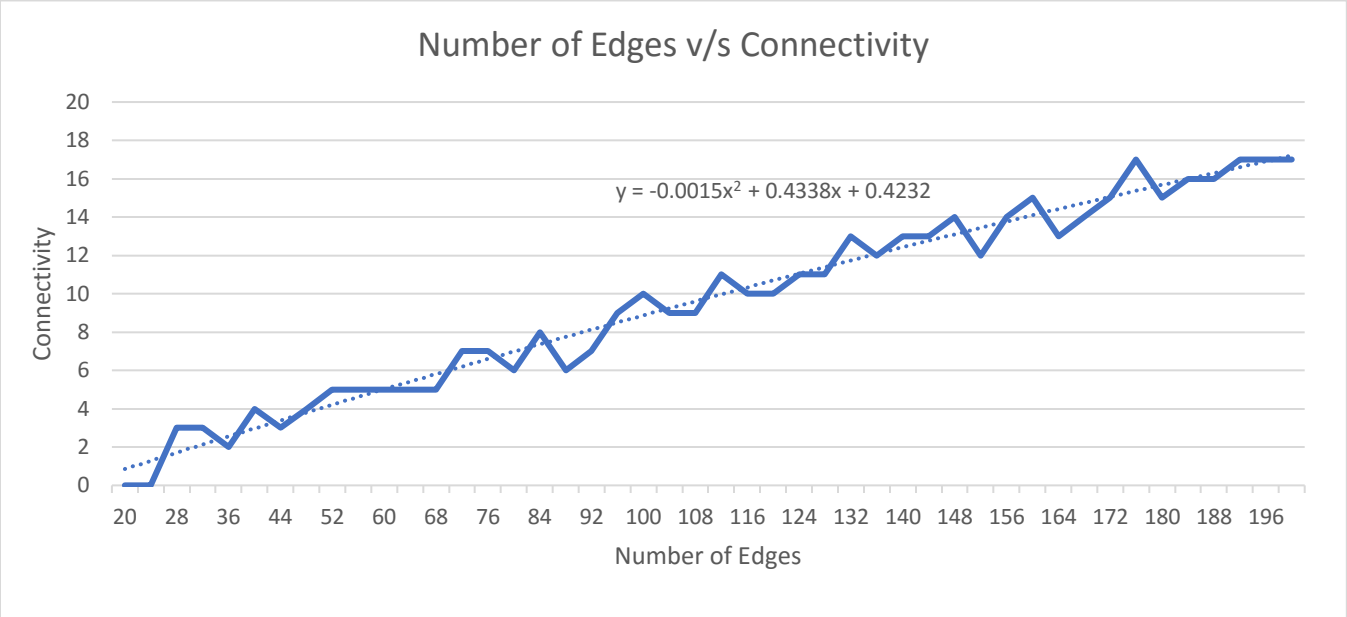
- One of the important findings here is the value of Stability with respect to the value of Connectivity.
- We show a table for the stability of the graph based on the Connectivity.

Connectivity	Stability
0	4
2	12
3	16
4	8
5	16
6	8
7	20
8	84
9	12
10	20
11	16
12	16
13	32
14	20
15	20
16	4
17	24

Table 2: The experimental values obtained for Connectivity and Stability

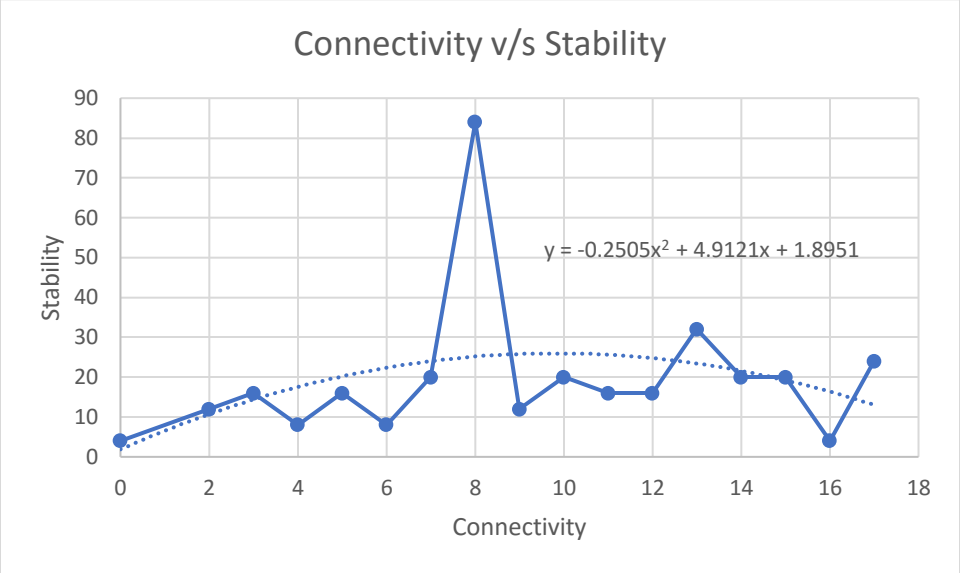
CHARTS

- The graph for Number of Edges v/s Connectivity



- Here, as seen in the graph, there is no specific linear relationship between the values of Number of Edges and Connectivity but a general view suggests that with an increase in the Number of Edges, there is an increase in the connectivity. But if we try to dig in deeper and generate quadratic equations, we get to see that there is a direct relation and can be shown using the equation.

- The graph for Connectivity v/s Stability



- Generally the stability increases with increase in connectivity upto a certain value of Connectivity but then it decreases with the connectivity. There can be some spikes in the center because we know that if there are d nodes, we get maximum stability near d/2 connectivity value but a general trendline is as described above. Here also we can't develop an accurate linear relation. Even a quadratic equation can't truly justify the relation but still it can give a better idea than the Linear Equation.

CONCLUSION

- There is no linear relation between number of edges and connectivity.
- There is no linear relation between connectivity and stability.
- Some quadratic relations exist that are shown using the equations.

APPENDIX

```
package nagamochi;

import java.util.ArrayList;
import java.util.Random;
import java.util.Stack;


public class Nagamochi {

   // Denotes Minimum cut
   int mcut[] = new int[21];

   // Denotes Maximum Adjacency Ordering
   int MA_Order[] = new int[21];
   int graph[][] = new int[21][21];
   int tmp[][]=new int[21][21];
   int nodes = 21;
   int tnodes=21;

   public Stack<Integer> stack = new Stack<Integer>();
   boolean flag=false;


   public static void main(String[] args) {

      Nagamochi nmi=new Nagamochi();
      int edge_start=20,average_edge=0,minimum_cut,critical_edge,sum=0,pos=0;

      ArrayList<Integer> vals=new ArrayList<Integer>();
      ArrayList<Integer> min=new ArrayList<Integer>();
      ArrayList<Integer> max=new ArrayList<Integer>();

      Boolean isConnected,flag=true;
      System.out.println("edge"+"\t\t"+"minimum_cut");
      while(edge_start <= 200){
                        sum=0;
                        pos=0;
                        //Run the loop 5 times to minimize the randomness effect
                        for(int iteration=0;iteration<5;iteration++){
                        //Call to initialize the graph
                        nmi.init();
                        //Generate random weights for the graph
                        nmi.rand_graph(edge_start);

                        for(int i=0;i<nmi.nodes;i++){
                           isConnected=nmi.DFS(i);

                           if(!isConnected){
                              flag=false;
                              break;
                           }
                         }

                        //Increase sum based on Min_Cut
                        if(flag=true)
                           sum=sum+nmi.get_min_cut(edge_start);
                        else
```

```
                sum=sum+0;
            }

            for(int h=0;h<nmi.nodes;h++){
                max.add(0);
            }

            minimum_cut=sum/5;
            if(!vals.contains(minimum_cut)){
                vals.add(minimum_cut);
                min.add(edge_start);
            }
            else{
                for(int l=0;l<vals.size();l++)
                {
                    if(vals.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;
        }

        for(int i=0;i<vals.size();i++){
            System.out.print("Mincut = "+vals.get(i)+"  ");
            System.out.println("\t Stability = "+Math.abs(min.get(i)-max.get(i)));


        }
    }



//initialize both the matrices
public void init(){
    for(int i=0;i<nodes;i++){
        for(int j=0;j<nodes;j++){
            graph[i][j]=0;
            tmp[i][j]=0;
        }
        MA_Order[i]=Integer.MAX_VALUE;
        mcut[i] = 0;
    }
}









//generate random graph of 21 nodes with number of edges = a
        // Here the code is written such that we avoid both self loops and parallel edges
public void rand_graph(int a){
    int i=0;
    while(i<a){
        Random rd = new Random();
        int flag = 0;
        while(flag!=1){
            int x=rd.nextInt(nodes);
            int y=rd.nextInt(nodes);
                    if(graph[x][y]==0 && graph[y][x]!=1){
```

```
                        graph[x][y]++;
                        tmp[x][y]++;
                        i++;
                        flag = 1;
                        }
                }
    }
  }


  //check if the graph is disconnected using DFS
  public boolean DFS(int start){
            int number_of_nodes = graph[start].length - 1;
    int[] visited = new int[number_of_nodes + 1];
    int i, element,count=0;
    visited[start] = 1;
    stack.push(start);

    while (!stack.isEmpty())
    {
      element = stack.pop();
      i = 1;// element;
      while (i <= number_of_nodes)
      {
        if (graph[element][i] == 1 && visited[i] == 0)
        {
          stack.push(i);
          visited[i] = 1;
        }
        i++;

      }
    }

    for(int j=0;j<nodes;j++){
      if(visited[j]==1)
        count++;
    }

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

    }
}




  //calculate the Maximum Adjacency order and get mincut value
            // This is done multiple times in order to remove(minimize) the effect of randomness
  public int get_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(((graph[k][MA_Order[j]])!=0)||((graph[MA_Order[j]][k])!=0)){
                   if(k < MA_Order[i])
                      contracted[k] += graph[k][MA_Order[j]] ;
                   else
                      contracted[k] += graph[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++){
       mcut[i] = mcut[i]+graph[j][MA_Order[vertex-1]];
       mcut[i] = mcut[i]+graph[MA_Order[vertex-1]][j];
       contracted[j]=0;

    }
    //selecting the minimum vertex

    if(mcut[i]<mcut[min_vertex])
       min_vertex=i;

    for(int j=0;j<nodes;j++){
       if(j < MA_Order[vertex-1]){
          graph[j][MA_Order[vertex-2]] = graph[j][MA_Order[vertex-2]] +  graph[j][MA_Order[vertex-1]];
          graph[j][MA_Order[vertex-1]]=0;
       }
       else if(vertex > 1){
          graph[MA_Order[vertex-2]][j] = graph[MA_Order[vertex-2]][j] +  graph[MA_Order[vertex-1]][j];
          graph[MA_Order[vertex-1]][j]=0;
          }
       }
    if(vertex > 1){
       graph[MA_Order[vertex-2]][MA_Order[vertex-2]]=0;
       graph[MA_Order[vertex-2]][MA_Order[vertex-1]]=0;
       graph[MA_Order[vertex-1]][MA_Order[vertex-2]]=0;
    }
    graph[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 mcut[min_vertex];
}

}
```

REFERENCES

- Lecture Notes
- Wikipedia
- Sanfoundary – DFS
- http://www.diss.fu-berlin.de/docs/servlets/MCRFileNodeServlet/FUDOCS_derivate_000000000270/1994_12.pdf
- http://www.orsj.or.jp/~archive/pdf/e_mag/Vol.47_04_199.pdf