PROJECT 2

An Implementation of the Nagamochi and Ibaraki Algorithm

Work done

by

Arun Prakash Themothy Prabu Vincnet

NET ID : AXT161330

SPRING'17 CS/TE6385

ALGORITHMIC ASPECTS OF TELECOMMUNICATION NETWORKING

CONTENTS

Chapter 1

Introduction

Often we want to find a minimum cut in the entire graph, not just between a specified source and destination. The size of this minimum cut characterizes the connectivity of the graph.

Let us consider an undirected graph, which is also unweighted, that is, no capacities assigned to the edges. If it models the topology of a network, then the size of the minimum cut tells us that minimum how many links have to fail to disconnect the network (assuming, of course, that it was originally connected). This is the edge-connectivity. We can similarly define node connectivity, as well. These connectivity values characterize the vulnerability of the network, that is, how easy it is to disconnect it. We mainly focus on the (simpler) edge-connectivity. Therefore, often we just call it connectivity, for short.

Definitions

• Edge-connectivity between two nodes: $\lambda(x, y)$ = minimum number of edges that need to be deleted to disconnect nodes $x \neq y$. $\lambda(x, y)$ is the size of a minimum cut between x and y.

• Edge-connectivity (of the graph): $\lambda(G)$ = minimum number of edges that need to be deleted to disconnect G.

$\lambda(G) = \min_{x,y} \lambda(x, y)$.

$\lambda(G)$ is the size of a minimum cut.

• Node-connectivity of the graph: $\kappa(G)$ = minimum number of nodes (vertices) that need to be deleted, such that the remaining graph is either disconnected, or has no edge at all.

$\kappa(G)$ is the size of a minimum vertex-cut.

Naming convention: If a graph has $\lambda(G) = k$, we call it k-connected. Simi- larly, if $\kappa(G) = k$, then the graph is called k-node-connected. Occasionally

we also use the phrase that the graph is $\geq$k-connected, to express that $\lambda(G) \geq k$.

Chapter 2

Project Description

The subject of this project is to create an implementation of the NagamochiIbaraki algorithm for finding a minimum cut in an undirected graph, and experiment with it.

The program runs on randomly generated samples. The number of nodes is fixed at n = 21, while the number m of edges is varied 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. The parallel edges are allowed since Nagamochi-Ibaraki algorithm may generate it in its internal working due to the merging of nodes.

The connection between the number of edges in the graph and its connectivity $\lambda(G)$ is analysed. (If the graph happens to be disconnected, then $\lambda(G)$ is taken to be 0). The relationship between $\lambda(G)$ as a function of m is shown graphically in a diagram, while keeping n = 21 fixed. Since the edges are selected randomly, therefore, we reduce the effect of randomness in the following way: the program runs 5 independent experiments for every value of m, one with each generated example for this m, and averages out the results.

For every connectivity value $\lambda = \lambda(G)$ that occurred in the experiments, largest and smallest number of edges with which this λ value occurred are recorded. Their difference is known as the stability of λ, and is denoted by $s(\lambda)$. A diagram on how $s(\lambda)$ depends on λ is drawn .

Chapter 3

Description of Nagamochi and Ibaraki Algorithm

Consider the edge-connectivity of a graph G is denoted by $\lambda(G)$ and $\lambda(x; y)$ denotes the connectivity between two different nodes x and y.

Let Gxy be the graph obtained from G by contracting (merging) nodes x and y. In this operation we omit the possibly arising loop (if x; y are connected in G), but keep the parallel edges.

We can now use the following result that is not hard to prove: For any two nodes x; y

$$\lambda(G) = \min\{ \lambda(x, y), \lambda(Gxy)\}$$

Thus, if we can find two nodes x and y for which λ (x, y) can be computed easily then the above formula yields a simple recursive algorithm. One can indeed easily find such a pair (x, y) of nodes. This is done via the so-called Maximum Adjacency (MA) ordering.

An MA ordering (v1... Vn) of the nodes is generated recursively the by the following algorithm: 1. Take any of the nodes for v1. 2. Once(v1...Vn)isalreadychosen,takeanodeforvi+1thathasthemaximumnumberof

edges connecting it with the set {v1... Vn } In any MA ordering (V_1... V_n) of the nodes,

$\lambda(V_{n-1},V_n) = d(V_n)$ holds, where d(:) denotes the degree of the node.

Therefore, it is enough to create an MA ordering, as described above, and take the last 2 nodes in this ordering for x and y. Then the connectivity between them will be equal to the degree of the last node in the MA ordering. Then one can apply formula (3) recursively to get the minimum cut. Here we make use of the fact that it reduces the problem to computing (Gxy) and Gxy is already a smaller graph.

Chapter 4

Algorithm Implementation Logic

4.1 Pseudo Code

Inputs :

- Number of Nodes (N=21)
- Number of edges m (Varied from 20 to 200 in the step of 4)

Minimum Cut Calculation :

1. Forming a Network Graph : The program establishes a random edge connection between the nodes.(No loops and no parallal edges)
2. This topology is stored in the adjacency matrix.
3. Calculating MA order
    a. A node is picked randomly as first node and is added into the MA order array.
    b. Node having maximum number of edge degree with the first is selected as next node and added to the MA order array, the adjacency matrix is altered to make the rows and columns of the next node zero.
    c. Node having maximum number of edge degree with the combined node in MA order array is selected as next node and is added to the MA order array.
    d. The process is continued till the last node.
    e. This gives the MA order Array.
    f. Degree of edge between last two nodes i.e. (V_{n-1}, V_n) is the $\lambda(x,y)$.
4. Graph Contraction to find minimum cut
    a. Last two nodes in the MA order array is combined (contracted) and adjacency is matrix is updated by adding the rows of both nodes and reflecting it on the second last node.
    b. Last node is removed; we call the MA order function for this contracted graph.
    c. We check weather $\lambda(G_{xy})$ is less than the $\lambda(x,y)$. (i.e. $\lambda(G_{xy})$ is equal to $\lambda(x,y)$ of the previous iteration if $\lambda(x,y)$ calculated from the current contracted graph is more than $\lambda(x,y)$ calculated in the previous iteration.)

5. Step 4 is repeated till we have two nodes left. After the final iteration we will have the Minimum cut value.

6. Calculate Average Minimum Cut
   a. Repeat Steps 1-5 for five times for each value of m (Varied from 20 to 200 in the step of 4) .

   b. Calculate the average MinCut from the five MinCuts generated for each value of m.

7. Calculate the Stability

   a. The largest and smallest number of edges for which this $\lambda = \lambda(G)$ value occurred are recorded. Their difference is known as the stability of λ, and is denoted by $s(\lambda)$.

Chapter 5

Results and Analysis

5.1 Program Output:

The software runs with given Number of nodes and edges as inputs. Number of Nodes is taken as 21 and number of edges is varied from 20 to 200 by step of 4.

Edges	AverageMinCut
20	0
24	0
28	0
32	1
36	1
40	1
44	1
48	1
52	2
56	2
60	2
64	2
68	2
72	2
76	2
80	3
84	3
88	3
92	4

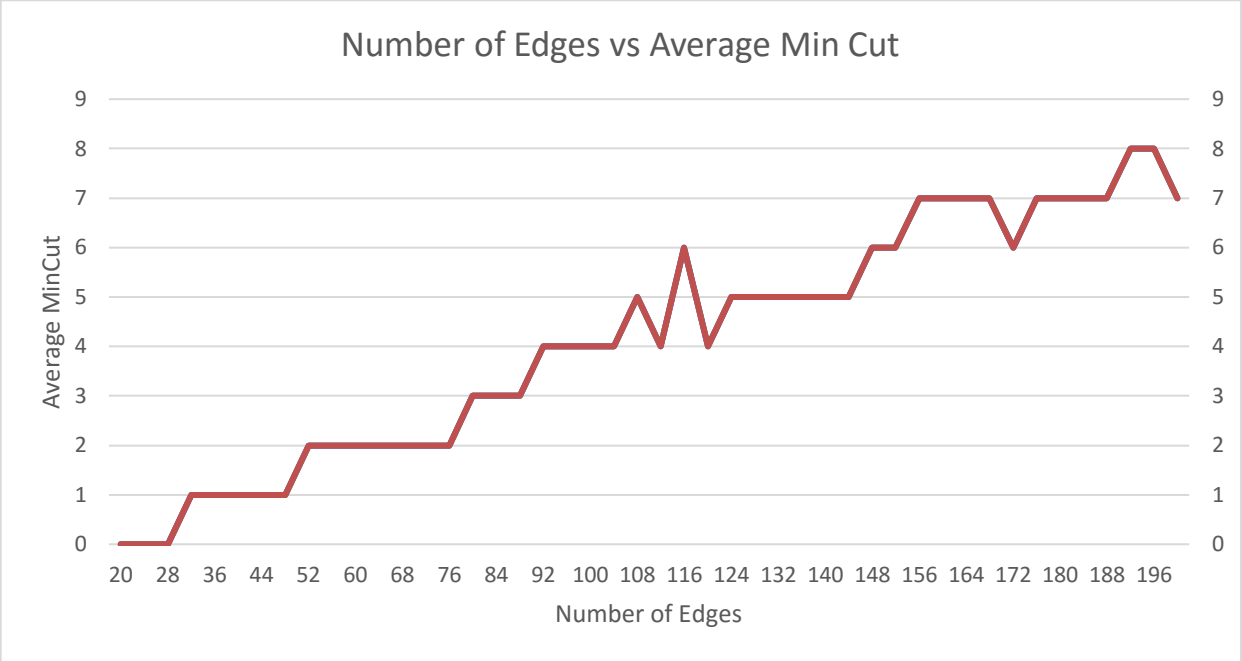
96	4
100	4
104	4
108	5
112	4
116	6
120	4
124	5
128	5
132	5
136	5
140	5
144	5
148	6
152	6
156	7
160	7
164	7
168	7

172	6
176	7
180	7
184	7
188	7
192	8
196	8
200	7

AverageMinCut	Stability
0	8
1	16
2	24
3	8
4	28
5	36
6	56
7	44
8	4

5.2 Graphs :

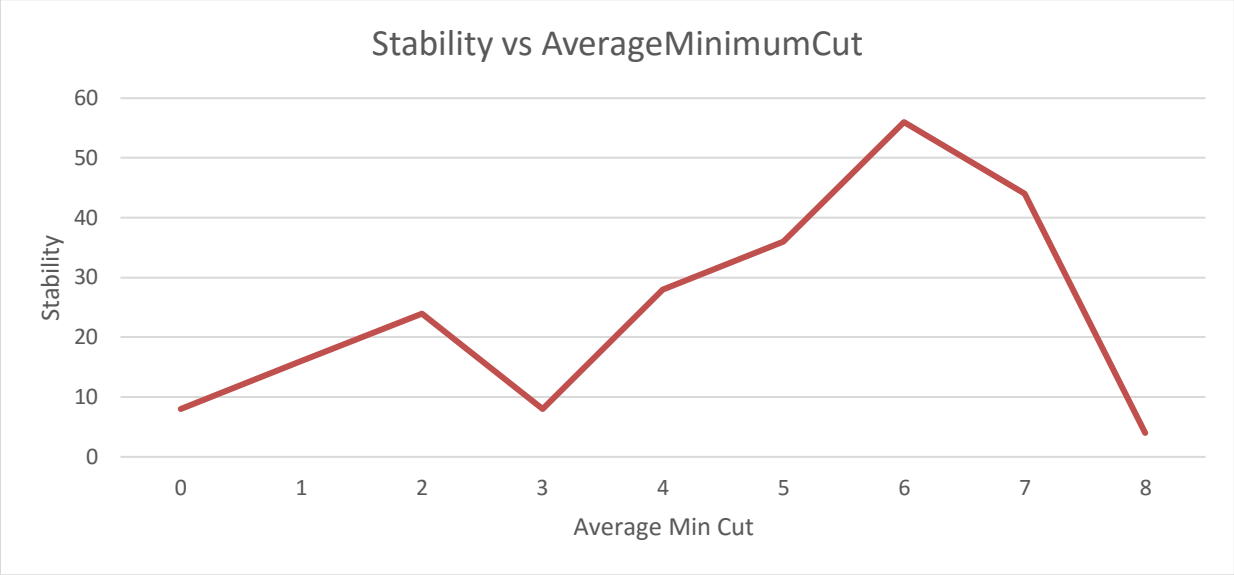
A. AverageMinCut vs Edges



Analysis:

From the above graph we can conclude that the Connectivity (AverageMinimumCut) is proportional to E (Number of Edges). That's because with the increase in number of edges the nodes may get multiple disjoint paths to reach every other node. Therefore the connectivity Increases. However ,the connectivity does not increase linearly with the edges and also decreases for some value of edges .This is because the adjacency matrix generation is random and therefore the generated graphs are random.

B. Stability Vs AverageMinCut



Analysis:

This graph shows impact of Average Minimum Cut on Stability. The Stability becomes zero for some value of AverageMiniumCut , this is because the maximum edge and minimum edge for the particular value of lamba (AverageMin Cut) is same. The stability factor is random since the graph for any value of edge is generated randomly.

Appendices

Appendix A – Source Code

```
package Atn;

import java.awt.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Random;

public class NagamochiAndIbarakiAlgorithm {


        //Graph Generator Function
        public static int[][] graphGenerator(int[][] graph, int nodes ,int edges){

          Random ranNum = new Random();//Random Variable to generate edges

          //Initialization of Graph Matrix
          for(int i=0; i<nodes ;++i){
                    for(int j=0; j<nodes; ++j)
                              graph[i][j] = 0;
          }

          //Creating a network graph
          for(int i=0; i<edges; ++i){
                    int s = ranNum.nextInt(nodes);
                    int d = ranNum.nextInt(nodes);
                    while(s==d || graph[s][d] !=0 ){ // condition for no-loop and unparallel edges
                              s = ranNum.nextInt(nodes);
                              d = ranNum.nextInt(nodes);
                    }
               graph[s][d] += 1;
               graph[d][s] += 1;
          }
          return graph;
        }


        public static int MinCut(int[][] Graph ,int nodes,int edges){

          Random random = new Random();
          int currentNode;
```

```
int nextNode =0;
int maxEdges = 0;
ArrayList<Integer> MA = new ArrayList();
int count = 0;
int delXY = 0;
int delG = Integer.MAX_VALUE;
int minCut = 0;

while(nodes >=2) // checking if nodes greater than or equal to 2 for contraction
{
        count =0;
        currentNode = random.nextInt(nodes);
        currentNode = 0;
        MA.add(currentNode); //Updating the MaximumAdjacency (MA) ordering
        count++;
        int[][] graph = new int[nodes][nodes];
        for(int i=0; i<nodes; ++i)
                for(int j=0; j<nodes; ++j)
                        graph[i][j] = Graph[i][j];
        while(count < nodes){
   maxEdges = 0;
                for(int j=0;j<nodes ;++j){
                        if(graph[currentNode][j] >= maxEdges){
                                maxEdges = graph[currentNode][j];
                           nextNode = j; // Finding next node with maximum adjacency with
nodes in MA Order
                        }
                }
         MA.add(nextNode);// Updating the MaximumAdjacency (MA) ordering
        count++;
        if(count == (nodes-2)){
                delXY = graph[currentNode][nextNode];
           }
        for(int j=0; j<nodes ;++j){
                        graph[currentNode][j] += graph[nextNode][j];
                        graph[j][currentNode] += graph[nextNode][j];
                        graph[currentNode][currentNode] = 0;
                        graph[nextNode][j] =0;
                        graph[j][nextNode] =0;
                }
         }
         // Graph Contraction
          int lastnode, lastsecondnode;
         lastnode = MA.get(MA.size() - 1);
          lastsecondnode = MA.get(MA.size() - 2);
          for (int j = 0; j < nodes; j++) {
                  Graph[lastsecondnode][j] += Graph[lastnode][j];
```

```
                Graph[j][lastsecondnode] += Graph[j][lastnode];
                Graph[lastsecondnode][lastsecondnode] = 0;
                Graph[lastnode][j] = 0;
                Graph[j][lastnode] =0;
                }

            // Updating the minimum connectivity if lamda(X,Y) < lamda(G)
            if (delG >= delXY) {
                    delG = delXY;
            }
            nodes--;
            int m =0, n=0;
            int[][] newGraph = new int[nodes][nodes];
            for (int i = 0; i < Graph.length; i++) {
                    if (i != lastnode) {
                    n = 0;
                    for (int j = 0; j < Graph.length; j++) {
                    if (j != lastnode) {
                    newGraph[m][n] = Graph[i][j];
                    n++;
                    }
                 }
               m++;
            }
                    else
                continue;
            }
            Graph = newGraph;
            minCut = delG; // updating the minCut

          }
          return minCut;
        }


        public static void main(String[] args) {

          int nodes = 21; // number of Nodes
          int edges = 20;
          int count =0;
          int[][] graph = new int[nodes][nodes];
          System.out.println("Edges     AverageMinCut  ");
          int[] minimumCuts = new int[5];
          Map<Integer,Integer> LargestEdge = new HashMap<Integer,Integer>(); //Hashmap for
maxEdge for a MinCut
          Map<Integer,Integer> SmallestEdge = new HashMap<Integer,Integer>();//Hashmap for
minEgde for a MinCut
```

```
        Map<Integer,Integer> stability = new HashMap<Integer,Integer>();// Hashmap for
Stability of the MinCut

          while(edges <= 200){
                  // Five Iterations for every Edge
                  while(count<5){
             graph = graphGenerator(graph, nodes, edges);
            int minCut = MinCut(graph, nodes,edges);
            minimumCuts[count] = minCut;
            count++;
           }

           int sum =0;
           int AverageMinCut = 0;
           for(int i=0; i<5; ++i){
             sum += minimumCuts[i];
           }

           AverageMinCut = sum/5;// Calculating Average Min cut

           //Caculating Stability - updating the maxEdge and MinEgde for a given Min Cut
         if(LargestEdge.containsKey(AverageMinCut))
         {
        if(LargestEdge.get(AverageMinCut) < edges)
          LargestEdge.put(AverageMinCut, edges);
         }
         else
         LargestEdge.put(AverageMinCut, edges);

         if(SmallestEdge.containsKey(AverageMinCut))
         {
        if(SmallestEdge.get(AverageMinCut) > edges)
          SmallestEdge.put(AverageMinCut, edges);
         }
         else
         SmallestEdge.put(AverageMinCut, edges);

         // calculating Stability - Difference of MaxEdge and MinEdge
         int Stability = LargestEdge.get(AverageMinCut) - SmallestEdge.get(AverageMinCut) ;
         stability.put(AverageMinCut, Stability);

         System.out.println("  "+ edges + "          " + AverageMinCut );
         System.out.println();
         edges += 4;
         count = 0;
    }
System.out.println("AverageMinCut     Stability ");
```

```
            for(Integer k : stability.keySet()){
                    System.out.println(k + "                    " + stability.get(k));
            }
           }
}
```