Algorithmic Aspects of Telecommunication Networks
CS 6385.001: Project #3

Due on Wednesday November 27, 2019 at 11:59am

Instructor: Prof. Andras Farago

Shyam Patharla (sxp178231)

Contents

1 Introduction

- In this project, we try to implement the Nagamochi Ibaraki algorithm for finding the edge connectivity of a connected graph
- Give an input graph with n nodes and m edges, our program outputs the edge connectivity of the graph
- We do this for range of values of m
- We also analyze how the edge connectivity varies with the value of m and assess the spread of the edge connectivity values
- We discuss the possible reasons for the above characteristics.

2 Design Decisions

- We implement the solution in the **Java** programming language
- The program modules were run on a **Mac** operating system

3 Solution Approach

3.1 Generating Input Examples

We first generate input graphs for simulating the algorithm. These graphs are then passed on to second *Module 1* module which runs the Exhaustive Enumeration algorithm on them. The graphs are generated as follows.

- For all examples, we set the number of nodes in the network to 5 i.e $n = 5$
- The topology of the graph for all examples is a complete graph, hence the numbe of edges m=10

3.2 Exhaustive Enumeration Algorithm

Module 2 receives input parameters from the first module (graph). It has the following functions.

- The n nodes in our network are always up
- We have m links each of which may fail
- The minimum number of links that may fail is zero, while the maximum number is m

- We can generate 2^m possible states, each of which may contain some links which fail and some links do not
- We proceed step by step, generating states with k failures, k=0,1,2,...m and store them
- We generate

Module 2 passes the generated states it on to Module 3, which computes the raliability for the network.

3.3 Reliability

Module 3 takes the output parameters of *Module 2* i.e. the **generated states** for an input graph.

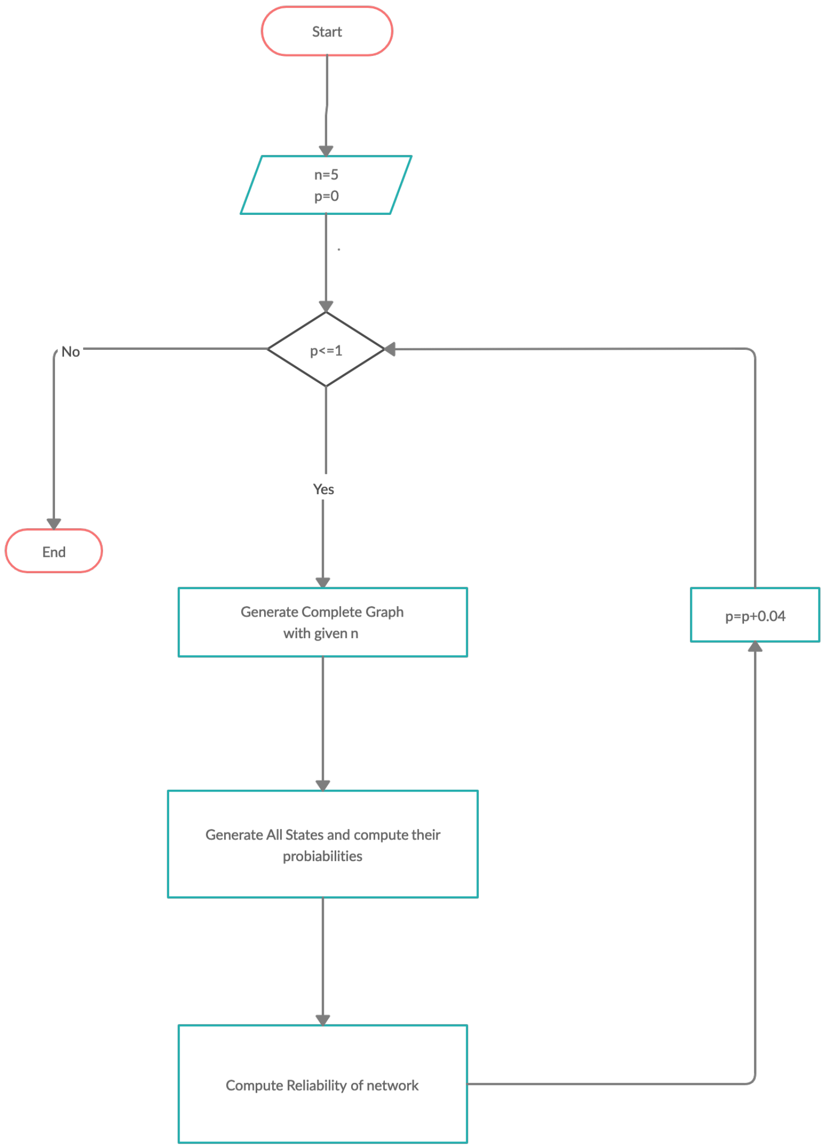
- We iterate through the states received
- For each state we check if the network is *connected* using a depth first search
- If it is connected, then the system condition is **up**, else the system condition is *down*
- We compute the **probability** of the state s using the formula

$$prob(s) = p * u + (1 - p) * d \tag{1}$$

  where u is the number of links which are up and d is the number of links which are down
- The **reliability** of the network is the sum of probabilities of the **up** states

3.4 Presentation of Results

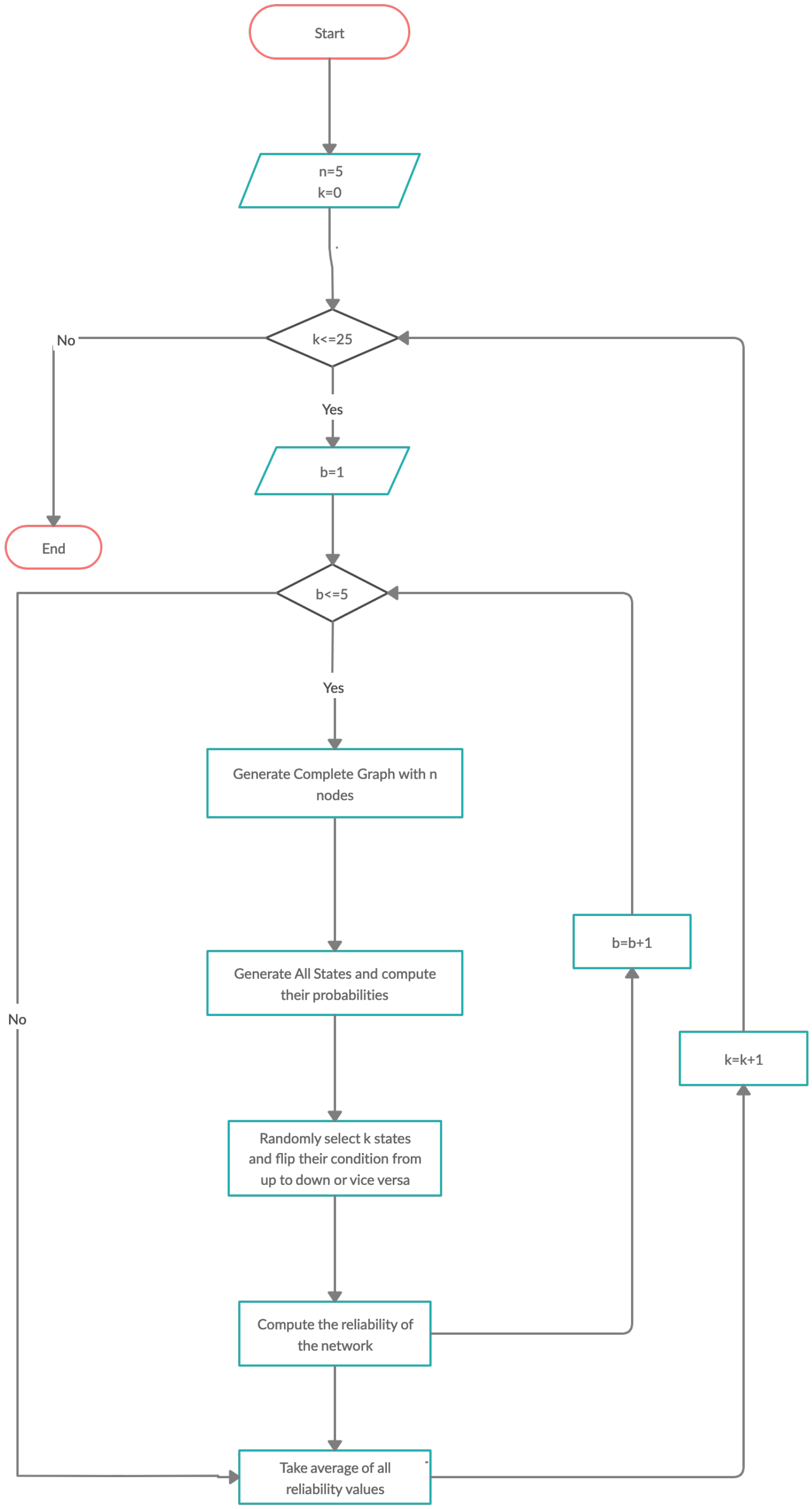
- We use a value **p** to denote the reliability of a component (link)
- We also use a parameter **k** which denotes the number of states (randomly chosen) whose condition is flipped i.e. from up to down or vice versa
- *Task 1* consists of varying the value of **p** and seeing how the reliability of the network changes
- We vary the value of p from 0 to 1 in steps of 0.04



- *Task 2* consists of fixing the value of **p**, varying **k** and seeing how the reliability of the network varies
- We fix the value of p to 0.87 and vary the value of k from 0 to 25 in steps of 1

- We take 5 trials for each value of k and take the average of the reliabilities

The flowchart for Task 2 is shown below.



4 Exhaustive Enumeration Algorithm - Explanation

- The goal is to find the *reliability* of a network where nodes anre always up but links may fail
- We generate **all possible states** for the network where each component may be *up* or *down*
- For each state, we check if the network is **connected**
- If it is connected we compute its probability using the formula

$$prob(s) = p * u + (1 - p) * d \tag{2}$$

  where **u** is the number of links which are up and **d** is the number of links which are down
- The **reliability** of the network is the sum of probabilities of all *up* states

Algorithm 1 ExhaustiveEnumeration

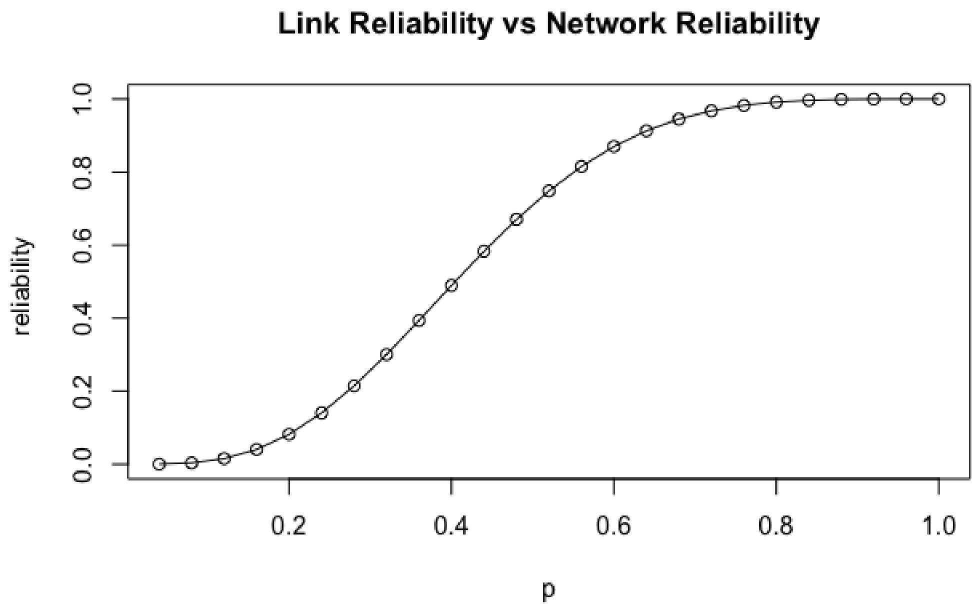
```
procedure ExhaustiveEnumeration(V, E, p)
    m =| E |
    states[][] = generateStates(m)
    t = 2^m
    reliability = 0
    for i = 1 to t do
        s = states[i]
        fail = 0
        for j = 1 to m do
            if s[i] = 1 then
                fail = fail + 1
            end if
        end for
        prob = p * (m − fail) + (1 − p) * fail
        G' = transform(V, E, s)
        if isConnected(G') then
            reliability = reliability + prob
        end if
    end for
     return reliability
end procedure
```

5 Observations and Analysis

- The program produces outputs as shown in the figures below.

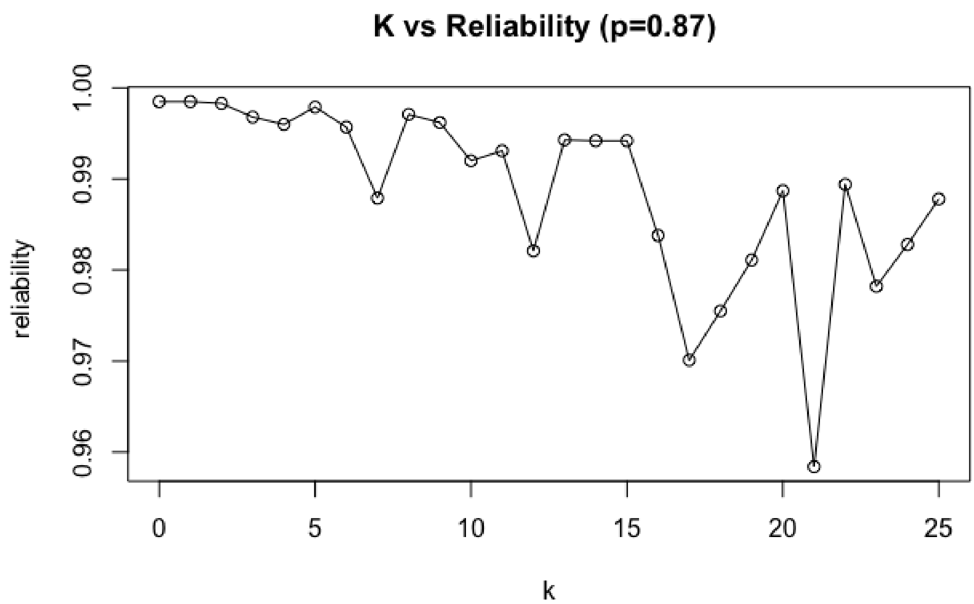
- The output results are stored in a *csv* file. The graphs are generated in **R**.

- We plot the graph of the *number of edges* m vs. the *edge connectivity* $\lambda(G)$



- We can clearly see that the edge connectivity of the network **increases** with increase in the value of m

- We plot the graph of edge connectivity values **lambda(G)** against their **spread**



- We can clearly see that the **spread** of edge connectivity values increases, saturates in the middle and then decreases

6 Discussion

- As the number of edges in the graph increases, the number of edge disjoint paths between any two nodes tends to increase, hence the edge connectivity of the graph also **increases**
- In our observations, the highest values of spread occur for $\lambda(G) = 2, 3, 4$ i.e middle values with decreasing values on either side
- The reason is that a given value of edges connectivity (especially the ones in the middle) tends to persist for a range of m values, especially since for each graph we choose edges randomly, which sort of randomises our progression as m increases

7 ReadMe File

This section shows how to run the project files.

- Downloads the project files and store them in a folder
- Open the project folder in Eclipse
- Open the file **Presentation.java**

- Right Click – > Run as – > Java Application
- Alternatively,navigate to the folder in **terminal** and run the following commands
  - javac Presentation.java
  - java Presentation

8 Code

Module 1: ExhaustiveEnumeration.java

```java


import java.util.ArrayList;
import java.util.Collections;

public class ExhaustiveEnumeration {

  public int numStates=0;
  private ArrayList<ArrayList<Integer>> states;

  /* Generates all possible states for a graph. Parameters
   * m: number of links
   * */
  public ArrayList<ArrayList<Integer>> generateStates(int m)
  {
    ArrayList<Integer> arr=new ArrayList<Integer>(Collections.nCopies(10, 1));
    this.states=new ArrayList<ArrayList<Integer>>();
    states.add(arr);
    for(int i=1;i<=m;i++)
    {
      //System.out.println("Generating states with "+i+" link failures...");
      getStatesWithKFails(i,arr);
      //System.out.println();
      }

    //System.out.println("Total States Generated: "+states.size());
    return this.states;
  }

  /* Generates states with k link failures */
  public void getStatesWithKFails(int k,ArrayList<Integer> arr)
  {
    for(int i=0;i<arr.size();i++)
    {
      getStates(arr,k,0,i);
    }
  }
```

```



  /* recursive function to generate states */
  public void getStates(ArrayList<Integer> arr, int maxFail,int curFail, int i
   )
  {
    arr.set(i,0);
    if(curFail+1==maxFail)
    {
      numStates++;
      ArrayList<Integer> a =new ArrayList<Integer>(arr);
      this.states.add(a);
      //Utils.printList(arr);
      arr.set(i, 1);
      return;
    }

    for(int j=i+1;j<arr.size();j++)
    {
      getStates(arr,maxFail,curFail+1,j);
      arr.set(j, 1);
    }
    arr.set(i, 1);
  }


  /* generates a complete graph with n nodes */
  public int[][] getCompleteGraph(int n)
  {
    int arr[][] = new int[n][n];
    for(int i=0;i<n;i++)
    {
      for(int j=0;j<n;j++)
      {
        if(i!=j)
        {
          arr[i][j]=1;
        }
      }
    }
    //Utils.printMatrix(arr, n, "input graph");
    return arr;

  }
}
```

Module 2: Reliability.java

```


import java.util.ArrayList;

public class Reliability {

  /* storing the reference to the 10 edges in our graph */
  public static final int[] aSet={0,0,0,0,1,1,1,2,2,3};
  private static final int [] bSet={1,2,3,4,2,3,4,3,4,4};


  /* computes the reliability of a network with
   * n : number of nodes
   * m: number of edges:
   * p: probability that a link does not fail
   * k: number of states that must be flipped
   */
  public double compute(int n,int m,double p, int k)
  {
    ExhaustiveEnumeration e=new ExhaustiveEnumeration();

    ArrayList<ArrayList<Integer>>res=e.generateStates(10);
    int[][] graph;


    double reliability=0.00;
    int fail;

    /* select k random states to flip */
    ArrayList<Integer> flipSet=new ArrayList<Integer>();
    double range=Math.pow((int)2, (int)m);
    while(flipSet.size()!=k)
    {
      int v = (int)(Math.random() * (range-1));
      if(!flipSet.contains(v))
      {
        flipSet.add(v);
      }
    }

    //System.out.print("Flipset: ");
    //Utils.printList(flipSet);

    for(int i=0;i<res.size();i++)
    {

      fail=0;
      ArrayList<Integer> a =res.get(i);
```

```
      graph=e.getCompleteGraph(n);
      for(int j=0;j<m;j++)
      {
        if(a.get(j)==0)
        {
          graph[aSet[j]][bSet[j]]=0;
          graph[bSet[j]][aSet[j]]=0;
          fail++;
        }
      }


      if(isConnected(graph,n) && !flipSet.contains(i) ||
          !isConnected(graph,n) && flipSet.contains(i))
      {
        //System.out.println("Connected");
        reliability+=Math.pow(p, m-fail)*Math.pow(1-p, fail);
        continue;
      }

    }

    return reliability;
  }


  /* checks if a graph is connected */
  public static boolean isConnected(int graph[][],int n) {
        ArrayList<Integer>nodes = new ArrayList < Integer > ();


        DFS(0,graph,n,nodes);
        if (nodes.size() != n) {
            return false;
        }
        return true;

    }


  /* runs a recursive dfs traversal on a graph */
  public static void DFS(int v,int[][] graph, int n,ArrayList<Integer> nodes)
   {
        nodes.add(v);
        for (int i = 0; i < n; i++) {
            if (graph[v][i] != 0 && !nodes.contains(i)) {
                DFS(i,graph,n,nodes);
            }
```

```
            }
    }
}
```

Module 3: Presentation.java

```


import java.text.DecimalFormat;

public class Analysis {

  private static DecimalFormat df=new DecimalFormat("#.####");
  public static void main (String[] args)
  {
    Reliability r= new Reliability();

    /* Values of graph parameters:
     * Number of nodes n=10
     * Topology: complete graph so m=10
     */
    int n=5;
    int m=10;
    double p,h;
    int k;

    /* Compute reliability for a random value of component reliability p
     * we set p=0.56 and k=0
     */
    p=0.56;
    k=0;
    h=r.compute(5, 10, p,0);
    System.out.println("p= "+df.format(p)+"  Reliability: "+df.format(h));



    /* Task 1: Vary p and observing how the reliability changes
     * p: Reliability of an individual component
     * Range of p values is [0,1]
     * Step size: 0.04
     * we set k=0 since we are not flipping any states
     * */
    System.out.println("Vary p and observing how the reliability changes...");
    for(p=0;p<1.01;p=p+0.04)
    {
      h=r.compute(n,m,p,k);
      System.out.println("p= "+df.format(p)+"  Reliability: "+df.format(h));
      //System.out.println(df.format(p)+", "+df.format(h));
    }


    /* Task 2: Fix p and vary k and observe how reliability changes.
   Parameters:
     * k: number of flipped states
```

```
     * b: Number of trials for a given value of k to minimize randomness
     * Fixing the value of p to 0.87
     * Range of k is [0,25]
     * Step size=1
     */
    System.out.println("Fix p and vary k and observe how reliability changes
...");
    p=0.87;
    int b=5;
    for(k=0;k<=25;k++)
    {
      double sum=0;
      for(int i=0;i<b;i++) {
      sum+=r.compute(n, m, p, k);

      }
      System.out.println("p = "+df.format(p)+"   k= "+k+"  reliability = "+df.
format(sum/b));
      //System.out.println(k+","+df.format(sum/b));
    }
  }
}
```

Utils.java

```java


import java.util.ArrayList;

public class Utils {
  public static void printMatrix(int[][] arr, int n, String s)
  {
    System.out.println(s);

    for(int i=0;i<n;i++)
    {
      System.out.print("row "+i+" : ");
      for(int j=0;j<n;j++)
      {
        System.out.print(arr[i][j]+" ");
      }
      System.out.println("\n");
    }
  }

  public static void printList(ArrayList<Integer> arr)
  {
    int n=arr.size();

    for(int i=0;i<n;i++)
    {
      System.out.print(arr.get(i)+" ");
    }
      System.out.println("");

  }





}
```

Visualization.R.java

```r
data <- read.csv(file="/users/psprao/eclipse-workspace/Nagamochi-Ibaraki/
    output1.csv")

# Getting K, cost and density  data
m<-data[,1]
lambda<-data[,2]


# Scatterplot of K vs Cost of Network
```

```
plot(m,lambda,xlab="m",ylab="lambda(G) ",main="Number of Edges vs Edge")
lines(lambda~m)


data <- read.csv(file="/users/psprao/eclipse-workspace/Nagamochi-Ibaraki/output.csv")

# Getting K, cost and density  data
lambda<-data[,1]
spread<-data[,2]


# Scatterplot of K vs Cost of Network
plot(lambda,spread,xlab="lambda(G)",ylab="spread ",main="Spread of Edge Connectivities  ")
lines(spread~lambda)
```