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 Exhaustive Enumeration algorithm for finding the reliability of a network
- Give an input network with n nodes , m edges, and component reliability p, our program outputs the reliability of the network
- We do this for range of values of p
- We can also flip the system condition for k states
- We also analyze how the reliability varies with the value of k when the value of p is fixed
- We discuss the possible reasons for the above characteristics.

2 Design Decisions

- We implement the solution in the **Java** programming language
- We use the **Eclipse** IDE since it has a lot of help for debugging
- 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 *Module 2* 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 number 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 in a particular state 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

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 network

- 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

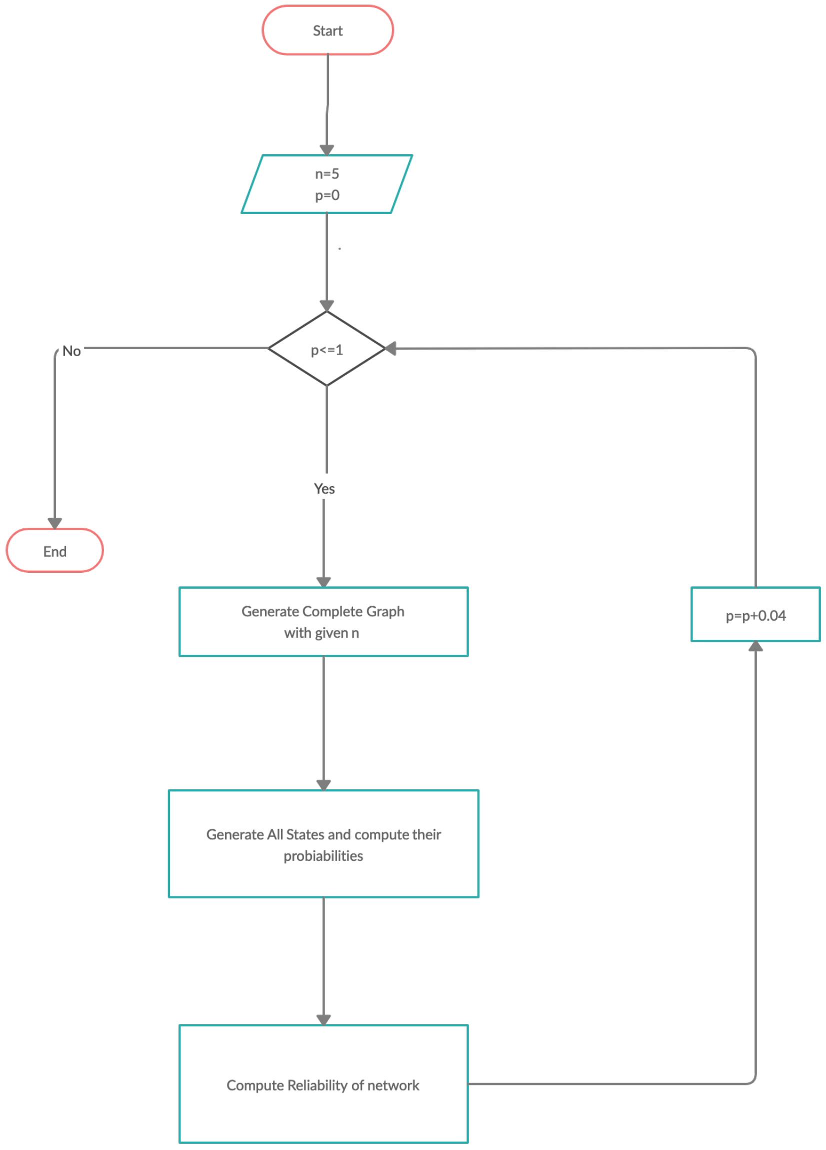
$$prob(s) = p * u + (1 - p) * d \tag{1}$$

  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

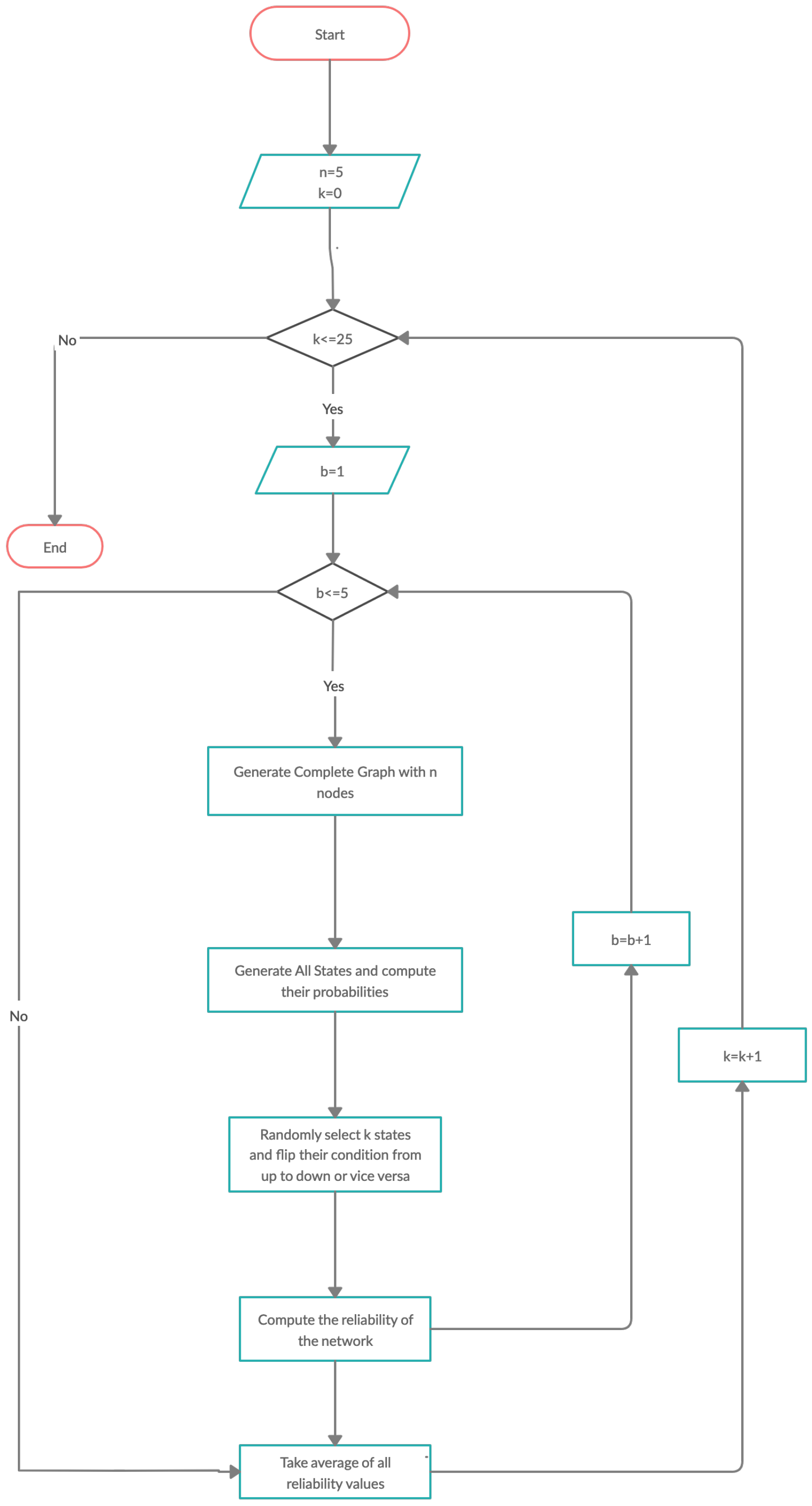
- We vary the value of p from 0 to 1 in steps of 0.04


Start
n=5
p=0
p<=1
No
Yes
End
Generate Complete Graph with given n
Generate All States and compute their probiabilities
Compute Reliability of network
p=p+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.


Start
n=5
k=0
k<=25
No
Yes
End
b=1
b<=5
Yes
No
Generate Complete Graph with n nodes
Generate All States and compute their probabilities
Randomly select k states and flip their condition from up to down or vice versa
Compute the reliability of the network
Take average of all reliability values
b=b+1
k=k+1

4 Exhaustive Enumeration Algorithm - Explanation

- The goal is to find the *reliability* of a network where nodes are 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

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

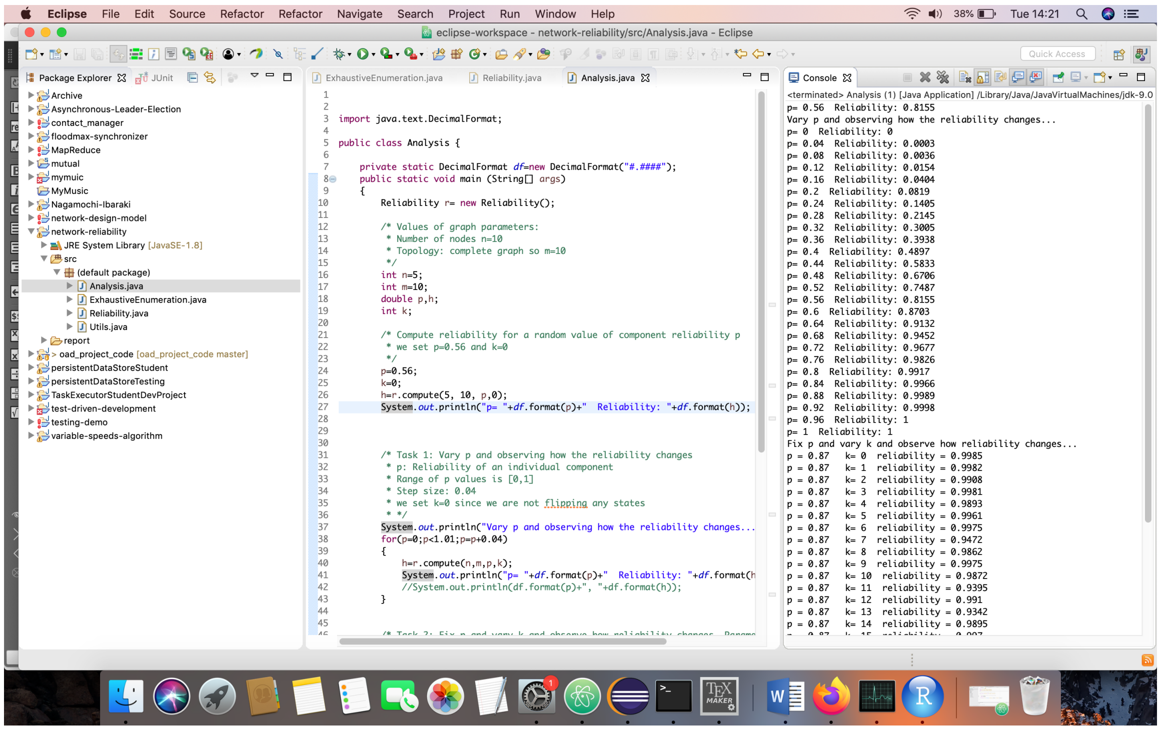
  where **u** is the number of links which are up and **d** is the number of links which are down
- The **reliability** of the network is the sum of probabilities of all *up* states

Algorithm 1 ExhaustiveEnumeration

```
1: procedure ExhaustiveEnumeration($V, E, p$)
2:     $m = \mid E \mid$
3:     $states[][] = generateStates(m)$
4:     $t = 2^m$
5:     $reliability = 0$
6:     for $i = 1$ to $t$ do
7:         $s = states[i]$
8:         $fail = 0$
9:         for $j = 1$ to $m$ do
10:            if $s[j] = 1$ then
11:                $fail = fail + 1$
12:            end if
13:        end for
14:        $prob = p * (m - fail) + (1 - p) * fail$
15:        $G' = transform(V, E, s)$
16:        if $isConnected(G')$ then
17:            $reliability = reliability + prob$
18:        end if
19:    end for
       return reliability
20: end procedure
```

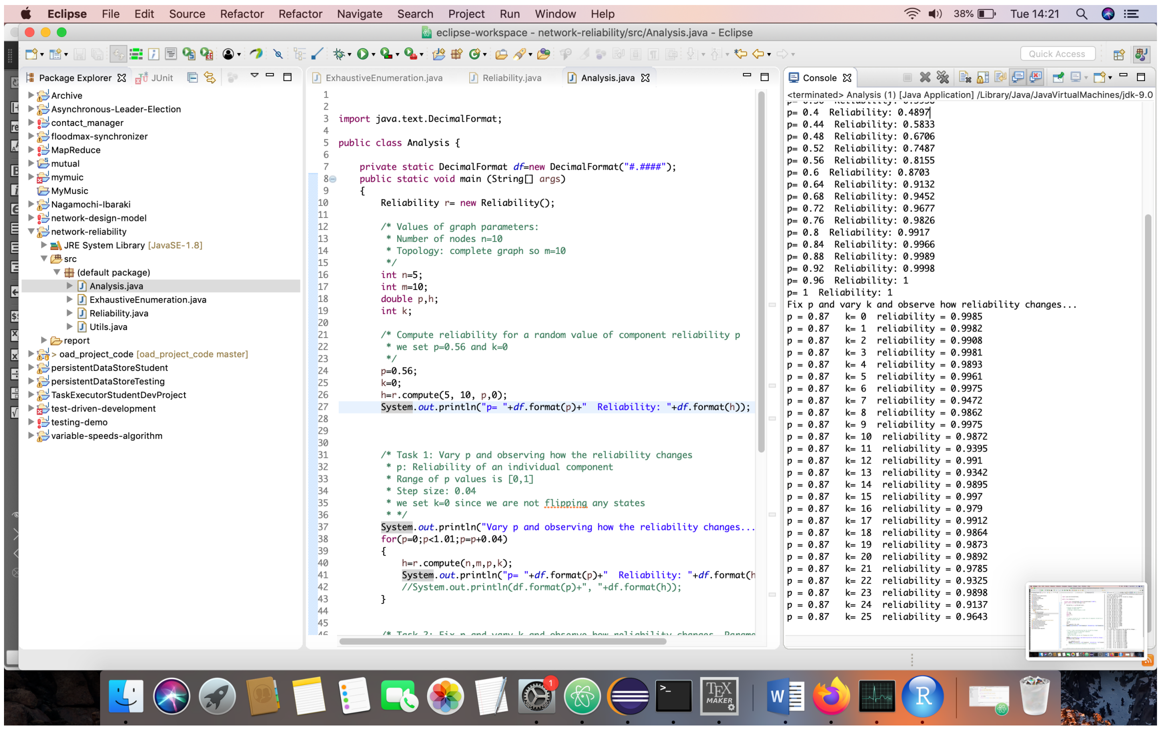
5 Observations and Analysis

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


eclipse-workspace - network-reliability/src/Analysis.java - Eclipse
Package Explorer
Archive
Asynchronous-Leader-Election
contact_manager
floodmax-synchronizer
MapReduce
mutual
mymuic
MyMusic
Nagamochi-Ibaraki
network-design-model
network-reliability
JRE System Library [JavaSE-1.8]
src
(default package)
Analysis.java
ExhaustiveEnumeration.java
Reliability.java
Utils.java
report
oad_project_code [oad_project_code master]
persistentDataStoreStudent
persistentDataStoreTesting
TaskExecutorStudentDevProject
test-driven-development
testing-demo
variable-speeds-algorithm
ExhaustiveEnumeration.java
Reliability.java
Analysis.java
Console

```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));
        }
```

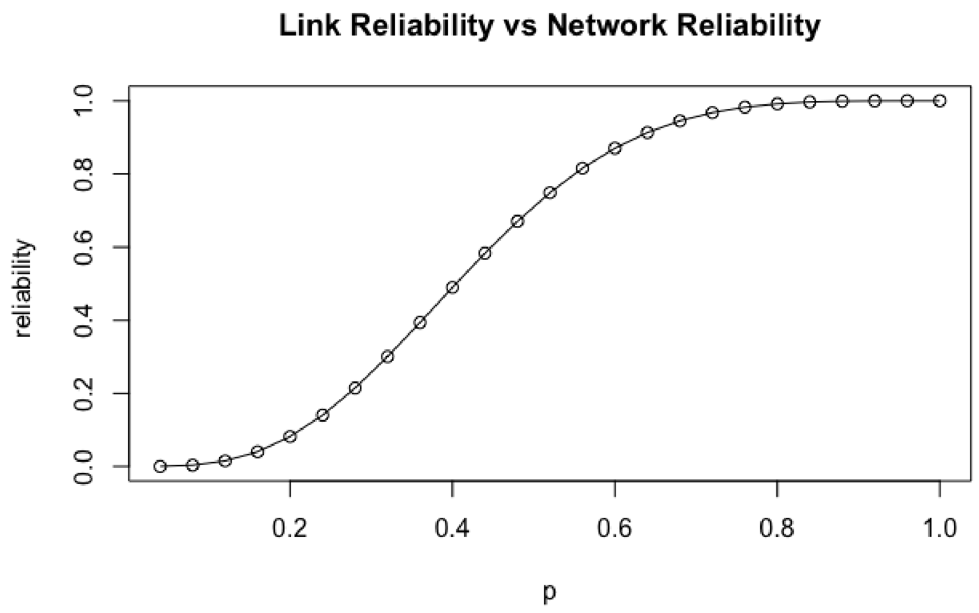
```
<terminated> Analysis (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk-9.0
p= 0.56  Reliability: 0.8155
Vary p and observing how the reliability changes...
p= 0  Reliability: 0
p= 0.04  Reliability: 0.0003
p= 0.08  Reliability: 0.0036
p= 0.12  Reliability: 0.0154
p= 0.16  Reliability: 0.0404
p= 0.2  Reliability: 0.0819
p= 0.24  Reliability: 0.1405
p= 0.28  Reliability: 0.2145
p= 0.32  Reliability: 0.3005
p= 0.36  Reliability: 0.3938
p= 0.4  Reliability: 0.4897
p= 0.44  Reliability: 0.5833
p= 0.48  Reliability: 0.6706
p= 0.52  Reliability: 0.7487
p= 0.56  Reliability: 0.8155
p= 0.6  Reliability: 0.8703
p= 0.64  Reliability: 0.9132
p= 0.68  Reliability: 0.9452
p= 0.72  Reliability: 0.9677
p= 0.76  Reliability: 0.9826
p= 0.8  Reliability: 0.9917
p= 0.84  Reliability: 0.9966
p= 0.88  Reliability: 0.9989
p= 0.92  Reliability: 0.9998
p= 0.96  Reliability: 1
p= 1  Reliability: 1
Fix p and vary k and observe how reliability changes...
p = 0.87   k= 0  reliability = 0.9985
p = 0.87   k= 1  reliability = 0.9982
p = 0.87   k= 2  reliability = 0.9908
p = 0.87   k= 3  reliability = 0.9981
p = 0.87   k= 4  reliability = 0.9893
p = 0.87   k= 5  reliability = 0.9961
p = 0.87   k= 6  reliability = 0.9975
p = 0.87   k= 7  reliability = 0.9472
p = 0.87   k= 8  reliability = 0.9862
p = 0.87   k= 9  reliability = 0.9975
p = 0.87   k= 10  reliability = 0.9872
p = 0.87   k= 11  reliability = 0.9395
p = 0.87   k= 12  reliability = 0.991
p = 0.87   k= 13  reliability = 0.9342
p = 0.87   k= 14  reliability = 0.9895
```



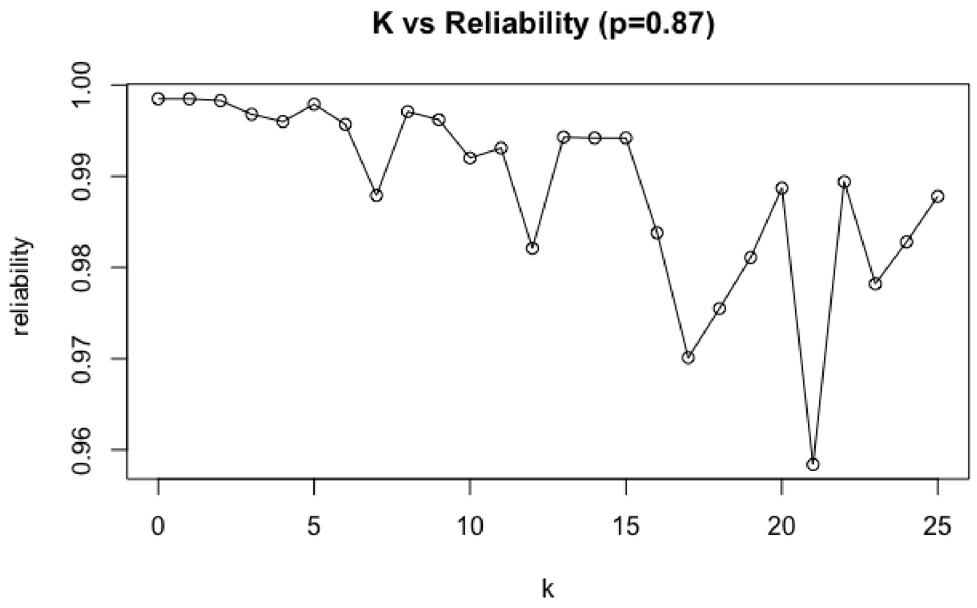
```java
import java.text.DecimalFormat;

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));
        }
```

```
<terminated> Analysis (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk-9.0
p= 0.4  Reliability: 0.4897
p= 0.44  Reliability: 0.5833
p= 0.48  Reliability: 0.6706
p= 0.52  Reliability: 0.7487
p= 0.56  Reliability: 0.8155
p= 0.6  Reliability: 0.8703
p= 0.64  Reliability: 0.9132
p= 0.68  Reliability: 0.9452
p= 0.72  Reliability: 0.9677
p= 0.76  Reliability: 0.9826
p= 0.8  Reliability: 0.9917
p= 0.84  Reliability: 0.9966
p= 0.88  Reliability: 0.9989
p= 0.92  Reliability: 0.9998
p= 0.96  Reliability: 1
p= 1  Reliability: 1
Fix p and vary k and observe how reliability changes...
p = 0.87   k= 0  reliability = 0.9985
p = 0.87   k= 1  reliability = 0.9982
p = 0.87   k= 2  reliability = 0.9908
p = 0.87   k= 3  reliability = 0.9981
p = 0.87   k= 4  reliability = 0.9893
p = 0.87   k= 5  reliability = 0.9961
p = 0.87   k= 6  reliability = 0.9975
p = 0.87   k= 7  reliability = 0.9472
p = 0.87   k= 8  reliability = 0.9862
p = 0.87   k= 9  reliability = 0.9975
p = 0.87   k= 10  reliability = 0.9872
p = 0.87   k= 11  reliability = 0.9395
p = 0.87   k= 12  reliability = 0.991
p = 0.87   k= 13  reliability = 0.9342
p = 0.87   k= 14  reliability = 0.9895
p = 0.87   k= 15  reliability = 0.997
p = 0.87   k= 16  reliability = 0.979
p = 0.87   k= 17  reliability = 0.9912
p = 0.87   k= 18  reliability = 0.9864
p = 0.87   k= 19  reliability = 0.9873
p = 0.87   k= 20  reliability = 0.9892
p = 0.87   k= 21  reliability = 0.9785
p = 0.87   k= 22  reliability = 0.9325
p = 0.87   k= 23  reliability = 0.9898
p = 0.87   k= 24  reliability = 0.9137
p = 0.87   k= 25  reliability = 0.9643
```

- The output results are stored in a *csv* file. The graphs are generated in **R**.
- We plot the graph of the *component reliability* p vs. the *network reliability*.



- We can clearly see that the reliability of the network **increases** with increase in the value of p
- We plot the graph of k values against network reliabilities.



- We can clearly see that the reliability of a network shows **no distinct behaviour** with respect to the k values

6 Discussion

- As the component reliability **p** in the graph increases, the probabilities contribution of the *up* states to the network reliability tends to increase, hence the reliability of the graph also **increases**
- When we fix the value of p to **0.87** and **vary k**, the reliability of the network shows no distinct behavior. Most of the reliability values are close to the reliability value without flipping any states i.e. k=0
- The reason is that flipping a few states (in our case, a maximum of 25 states out of 1024) does not affect the reliability of the network. This shows that the Exhaustive Enumeration algorithm gives us an accurate estimate of the network reliability even if some states are wrongly assigned system condition.

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

8 Code

Module 1: ExhaustiveEnumeration.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

```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

```


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

```
data <- read.csv(file="/users/psprao/eclipse-workspace/network-reliability/
    pVsR.csv")

# Getting K, cost and density  data
p<-data[,1]
r<-data[,2]


# Scatterplot of K vs Cost of Network
```

```
plot(p,r,xlab="p",ylab="reliability ",main="Link Reliability vs Network
    Reliability")
lines(r~p)


data <- read.csv(file="/users/psprao/eclipse-workspace/network-reliability/
    kVsR.csv")

# Getting K, cost and density  data
k<-data[,1]
r<-data[,2]


# Scatterplot of K vs Cost of Network
plot(k,r,xlab="k",ylab="reliability ",main="K vs Reliability (p=0.87) ")
lines(r~k)
```