CS 6385.001 - Algorithmic Aspects of Telecommunication Networks - F15
Project 3 Report

Peng Li - pxl141030@utdallas.edu

November 30, 2015

1 Introduction

This project is to develop an efficient algorithm for calculating the network reliability using the method of exhaustive enumeration, and then to implement it. Further more, it is to study experimentally how the network reliability depends on the individual link reliabilities; and how the network reliability changes as a number of component states flip occurs.

The network configuration of this project is as follows:

- *Network topology:* A complete undirected graph on $n = 5$ nodes, parallel edges and self-loops are excluded. As a result, this graph has $m = n(n-1)/2 = 10$ edges, representing the links of the network;
- *Components that may fail:* The links of the network may fail, the nodes are always up. The reliability of each links is p, the same for every link;
- *Reliability configuration:* The system is considered operational, if the network is connected.

The specific tasks in the project are as follows:

1. *Algorithm development:*
   Develop an algorithm to calculate network reliability using exhaustive enumeration for the network configuration described above.
2. *Algorithm implementation:*
   Implement the developed algorithm using the programming language and OS of any choice.
3. *Experiments:*
   (a) ***Experiment 1:***
   Calculate the network reliability of the given network configuration with different values of link reliability p for p ranges over $[0, 1]$ in step of 0.04. Study how obtained network reliability values depends on values of p.
   (b) ***Experiment 2:***
   For the given network configuration, fix the link reliability $p = 0.9$. Flip the network component states for k randomly selected component states, and calculate the system reliability after flipping. Parameter k ranges from $[0, 1, 2, ..., 30]$. Study how the reliability of the system changes due to this alternation and how it depends on the value of k.

2 Algorithm

2.1 Exhaustive Enumeration

Exhaustive enumeration is an exact method to calculate network reliability by listing all possible states of the given system and assign "UP" and "DOWN" system condition to each state. Then the system reliability can be obtained by summing up the probability of all the "UP" states.

The number of states has an exponential growth rate, this algorithm therefore is not a scalable solution. However, in this project, as our network configuration is relatively small in size ($n = 5, m = 10$), this approach is still practical.

2.2 Description

In the configuration of this project, the network is represented with a complete undirected graph with no parallel edges and self-loops. The number of nodes is $n = 5$, and therefore the number of links is $m = n(n-1)/2 = 10$. Nodes are always functioning, each link may be functioning or fail, therefore, there are $2^m = 2^{10} = 1024$ possible combinations of component states of the system.

According to the exhaustive enumeration method, calculating system reliability can be broken into the 3 sub tasks below:

1. **Generate all possible system states:**
   In order to generate all possible states of the system, we can treat the 10 links as a set of links $A = \{links\}$, any subset S of A can represent a unique system state by defining that all links in S are functioning and all links not in S fail.
   Therefore, finding all possible system states is converted to the problem of finding all subsets of the set of links(A). This can be easily archived using a simple recursion algorithm.
   Please refer to the pseudocode in Algorithm 1 of Section 2.3.

2. **Assign system condition to each state:**
   After generating all 1024 subsets of the set of links, we can construct an undirected graph for each of the subset of links. Each of the graph represents a system state, in order to assign system condition for each state, we need to find if the graph representing the system state is a connected graph.
   We do this by traversing the graph using functioning links only (Depth-First Search or Breath-First Search), if the traversal visits all nodes in the graph, that means this graph is connected, therefore we assign "UP" condition to this state; otherwise, we assign "DOWN" condition to this state.
   Please refer to the pseudocode in Algorithm 2 of Section 2.3.

3. **Calculate overall system reliability:**
   Overall system reliability R_{system} can be calculated iterating all "UP" states of the 1024 system states, and summing up the reliability R_s of each states.

   $$R_{system} = \sum_{UPstates} R_s$$

   For each "UP" state, the reliability is calculated by multiplying the link reliability p of all "functioning" links and the link failure probability $(1-p)$ of all "failed" links.

   $$R_s = \prod_{functioning} p \prod_{failed} (1-p)$$

   Please refer to the pseudocode in Algorithm 3 of Section 2.3.

2.3 Pseudocode

Algorithm 1 Generate All Possible System States

```
procedure GenerateLinkSubSets(E)                ▷ input - E represents all links of the network
    result ← new List(Set(links))               ▷ result to return represent all possible subsets of E
    subSet ← new Set(links)                     ▷ an empty subset to start with
    GenerateLinkSubSetsHelper(result, subSet, E, 0)   ▷ call the helper function recursively
return result
end procedure

procedure GenerateLinkSubSetsHelper(result, subSet, E, index)   ▷ a helper recursive function to generate all subsets of E
    Add subSet into result
    for (int i = index, i < E.size(), i = i + 1) do
        Add E[i] into subSet
        GenerateLinkSubSetsHelper(result, subSet, E, i + 1)
        Remove E[i] from subSet
    end for
end procedure
```

Algorithm 2 Calculate System Condition

```
procedure IsGraphConnected(G(V, E))     ▷ Use DFS to calculate if G(V, E) is a connected graph
    startNode ← pick a node from V      ▷ start with any node in G(V, E)
    stack ← new Stack(nodes)            ▷ stack stores nodes to be traversed
    stack.push(startNode)
    while stack is not empty do
        currentNode ← stack.pop()
        set currentNode to visited
        neighbors ← all neighboring nodes of currentNode that are connected with a "functioning" link
        for each node in neighbors do
            if (node is not visited) then
                stack.push(node)
            end if
        end for
    end while

    if (all nodes in V are visited) then return true   ▷ G(V, E) is connected, system condition is "UP"
    else return false                   ▷ G(V, E) is not connected, system condition is "DOWN"
    end if
end procedure
```

Algorithm 3 **Network Reliability Calculation**

```
procedure RELIABILITY(G(V, E))                              ▷ G(V, E) is the input network
    statesMap ← new HashMap(Graph, Boolean)                 ▷ Map to store network states and conditions
    result ← 0                                              ▷ Calculated network reliability
    linkSubSets ← GenerateLinkSubSets(E)

    for each linkSubSet ∈ linkSubSets do                    ▷ Find all system states and its corresponding conditions
        Create a new graph G_s(V, linkSubSet)
        isConnected ← IsGraphConnected(G_s)
        put (G_s, isConnected) into statesMap
    end for

    for each Entry(G_s, isConnected) ∈ statesMap do         ▷ Calculate system reliability
        if (isConnected == true) then
            result = result + reliability of G_s            ▷ Calculate single state reliability
        end if
    end for
return result
end procedure
```

2.4 Implementation

The project is implemented using Java JDK 1.7.
Note that the implementation requires dependencies on the following third party libraries:

- JUNG 2 Java Universal Network / Graph Framework, this library is used for graph representation and visualization.
  download at http://sourceforge.net/projects/jung/files/
- Opencsv, this library is used to write the output of the experiment into a .csv file for easier manipulation.
  download at http://sourceforge.net/projects/opencsv/

2.4.1 NetworkReliability Class

The implementation of the proposed exhaustive enumeration algorithm is formulated in *NetworkReliability* class. This class has the following instance variables:

- *network* is a undirected graph that represents the network topology;
- *networkStates* is a HashMap that maps all possible network states to its corresponding conditions.

NetworkReliability class provides 3 public method:

- *NetworkReliability* is the class constructor, during the instantiation, it calls several private method to generate network topology, generate all system states and assign system conditions for each states;
- *networkReliability* is the function to calculate network reliability and return it to the caller;
- *networkReliabilityFlipped* is used for *Experiment 2* of this project to flip k randomly selected system conditions and return system reliability after flipping.

The following methods for carry out the algorithm are defined private:

- *constructCompleteNetwork* construct and return a complete undirected graph that has no parallel links and selfloops;
- *linksSubSet* generate and return all possible subsets of a given set of elements;

- *isConnected* calculate and return if a given graph is connected by traversing the graph using DFS;
- *reliability* calculate and return the reliability of a single network state.

The implementation follows the *Incremental Development* method, where all sub-routine or function are designed, implemented and tested incrementally. Errors and bugs are found and eliminated at early stage, this also makes the code re-useable for future implementation of similar functionality or trying out algorithm changes.
For source code, please refer to Appendix A.1.

2.4.2 Project3 Class

The *Project3* class is the driving class, it creates an instance of *NetworkReliability* class and perform the required experiments with the created instance.

Note that in *Experiment 2* to reduce the effect of randomness, the experiment is carried out independently 1000 times for each k. For source code, please refer to Appendix A.2.

2.4.3 NetworkLink Class

The *NetworkLink* class is used to represent the graph edge in the software implementation. In this project, all edges of the graph are undirected and have identical reliability value. For source code, please refer to Appendix A.3.

2.5 How To Run

Before running the program, make sure to have the third party Java libraries as described in section included in the class path.

- Create package named "project3" and copy all source code files into that package;
- Create a folder named "data" in your project root folder;
- In terminal, run "javac Project3.java" and then "java Project3";

The program should print out the experiment result to terminal as well as output it to csv file in ".data/" folder.

3 Experiment Result

3.1 Network Reliability and p

The result of network reliability obtained from the given configuration with different link reliability p is shown in Figure 1 and Table 1.

3.2 Network Reliability and k

The result of network reliability obtained from the given configuration with different number of random condition flips k is shown in Figure 2 and Table 2.

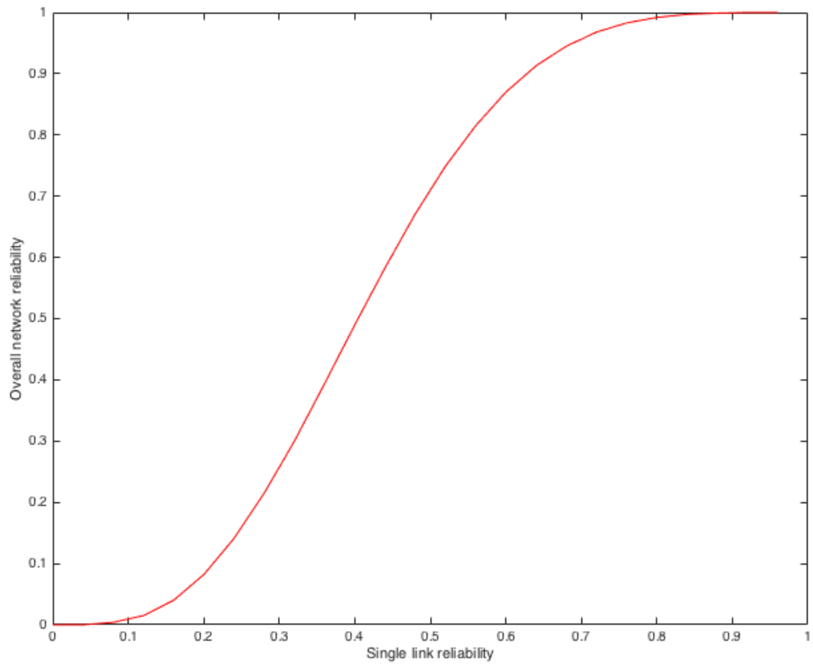

Figure 1: Network reliability and p

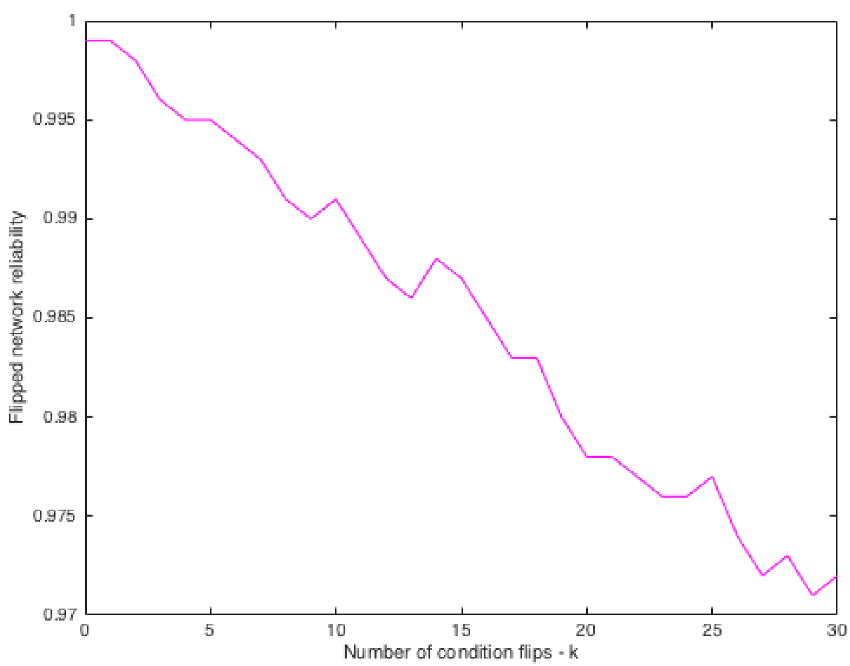

Figure 2: Network reliability and k

Table 1: Network reliability and p

link reliability p	network reliability
0.00	0.000
0.04	0.000
0.08	0.004
0.12	0.015
0.16	0.040
0.20	0.082
0.24	0.141
0.28	0.215
0.32	0.300
0.36	0.394
0.40	0.490
0.44	0.583
0.48	0.671
0.52	0.749
0.56	0.815
0.60	0.870
0.64	0.913
0.68	0.945
0.72	0.968
0.76	0.983
0.80	0.992
0.84	0.997
0.88	0.999
0.92	1.000
0.96	1.000
1.00	1.000

3.3 Explanation

From the result of Experiment 1, we could see that the network reliability increases as the single link reliabiliy increases. This is quite intuitive because given the same newtork topology, the more reliable the links are the more reliable the network is.

From the result of Experiment 2, we could see that the network reliability decreases as the number of random flips increases. This is because that among the 1024 possible system states, there are 728 "UP" states and 296 "DOWN" states. As the k state flips are picked up randomly, it is more likely to alter an "UP" state to "DOWN" state, and as a consequence this state will be excluded from the network reliability calculation. In other words, it is more likely to exclude some states from network reliability calculation than to include some states into it, therefore yields in a lower system reliability value.

Table 2: Network reliability and k

number of random flips k	network reliability
0	0.999
1	0.999
2	0.998
3	0.996
4	0.995
5	0.995
6	0.994
7	0.993
8	0.991
9	0.990
10	0.991
11	0.989
12	0.987
13	0.986
14	0.988
15	0.987
16	0.985
17	0.983
18	0.983
19	0.980
20	0.978
21	0.978
22	0.977
23	0.976
24	0.976
25	0.977
26	0.974
27	0.972
28	0.973
29	0.971
30	0.972

A Source Code

A.1 NetworkReliability.java

```
package project3;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Paint;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.Set;
import java.util.Stack;

import javax.swing.JFrame;

```

```java
import org.apache.commons.collections15.Transformer;

import edu.uci.ics.jung.algorithms.layout.CircleLayout;
import edu.uci.ics.jung.algorithms.layout.Layout;
import edu.uci.ics.jung.graph.UndirectedSparseGraph;
import edu.uci.ics.jung.graph.util.Pair;
import edu.uci.ics.jung.visualization.BasicVisualizationServer;
import edu.uci.ics.jung.visualization.decorators.ToStringLabeller;
import edu.uci.ics.jung.visualization.renderers.Renderer.VertexLabel.Position;

/**
 * Used to be instantiated to perform required network reliability experiments
 *
 * @author LiP
 *
 */
public class NetworkReliability {

  // network representation (a complete undirected graph)
  private UndirectedSparseGraph<Integer, NetworkLink> network;
  // network states representation (each entry in the map is a single state of
  // the network, key is the network representation, value indicates the
  // network status (UP or DOWN))
  private Map<UndirectedSparseGraph<Integer, NetworkLink>, Boolean> networkStates;

  /**
   * constructor
   *
   * @param numOfNodes
   *            : number of nodes in the network
   * @param reliability
   *            : reliability of each link assumed that all links have
   *            identical reliability
   */
  public NetworkReliability(int numOfNodes, double reliability) {
    this.network = constructCompleteNetwork(numOfNodes, reliability);

    List<NetworkLink> allLinks = new ArrayList<NetworkLink>(
        network.getEdges());
    List<List<NetworkLink>> allLinksSubSet = linksSubSet(allLinks);

    networkStates = new HashMap<UndirectedSparseGraph<Integer, NetworkLink>, Boolean>();

    for (List<NetworkLink> links : allLinksSubSet) {
      UndirectedSparseGraph<Integer, NetworkLink> event = constructCompleteNetwork(
          numOfNodes, reliability);
      for (NetworkLink link : links) {
        Pair<Integer> nodes = network.getEndpoints(link);
        event.findEdge(nodes.getFirst(), nodes.getSecond())
            .setLinkDown();
      }

      networkStates.put(event, isConnected(event));
    }
  }

  /**
   * calculate network reliability
   *
   * @return
   */
  public double networkReliability() {
    double result = 0;
    for (UndirectedSparseGraph<Integer, NetworkLink> graph : networkStates
        .keySet()) {
      if (networkStates.get(graph)) {
        result += reliability(graph);
      }
```

```java
    }
    return result;
  }

  /**
   * flip network states and calculate reliability of flipped network states
   *
   * @param k
   * @return
   */
  public double networkReliabilityFlipped(int k) {

    Set<UndirectedSparseGraph<Integer, NetworkLink>> flippedStates = new
        HashSet<UndirectedSparseGraph<Integer, NetworkLink>>();

    Random random = new Random();
    List<UndirectedSparseGraph<Integer, NetworkLink>> keys = new
        ArrayList<UndirectedSparseGraph<Integer, NetworkLink>>(
        networkStates.keySet());

    while (flippedStates.size() < k) {
      UndirectedSparseGraph<Integer, NetworkLink> randomKey = keys
          .get(random.nextInt(keys.size()));

      if (!flippedStates.contains(randomKey)) {
        flippedStates.add(randomKey);
        networkStates.put(randomKey, !networkStates.get(randomKey)); // flip
      }
    }

    double result = networkReliability();

    // flip it back
    for (UndirectedSparseGraph<Integer, NetworkLink> graph : flippedStates) {
      networkStates.put(graph, !networkStates.get(graph)); // flip back
    }

    return result;
  }

  /**
   * Construct a complete graph represent the network
   *
   * @param numOfNodes
   *            - number of nodes in the graph
   * @param reliability
   *            - reliability of network links
   * @return the constructed complete graph
   */
  private UndirectedSparseGraph<Integer, NetworkLink> constructCompleteNetwork(
      int numOfNodes, double reliability) {

    UndirectedSparseGraph<Integer, NetworkLink> result = new UndirectedSparseGraph<Integer,
        NetworkLink>();

    for (int i = 1; i <= numOfNodes; i++) {
      result.addVertex(i);
    }

    for (int i = 1; i <= numOfNodes; i++) {
      for (int j = 1; j <= numOfNodes; j++) {
        if (i != j && !result.isNeighbor(i, j)) {
          result.addEdge(new NetworkLink(reliability), i, j);
        }
      }
    }

    return result;
```

```
  }

  /**
   * find out all states
   *
   * @param links
   * @return
   */
  private List<List<NetworkLink>> linksSubSet(List<NetworkLink> links) {
    List<List<NetworkLink>> result = new ArrayList<List<NetworkLink>>();
    List<NetworkLink> list = new ArrayList<NetworkLink>();

    if (links == null || links.size() == 0) {
      return result;
    }

    subsetHelper(links, result, list, 0);
    return result;
  }

  /*
   * helper function to find all possible states
   */
  private void subsetHelper(List<NetworkLink> links,
      List<List<NetworkLink>> result, List<NetworkLink> list, int index) {
    result.add(new ArrayList<NetworkLink>(list));

    for (int i = index; i < links.size(); i++) {
      list.add(links.get(i));
      subsetHelper(links, result, list, i + 1);
      list.remove(list.size() - 1);
    }
  }

  /**
   * use DFS to tell if the network is connected
   *
   * @param graph
   * @return
   */
  private boolean isConnected(
      UndirectedSparseGraph<Integer, NetworkLink> graph) {
    List<Integer> allNodes = new ArrayList<Integer>(graph.getVertices());
    Integer startNode = allNodes.get(0);

    Set<Integer> visited = new HashSet<Integer>();
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(startNode);

    while (!stack.empty()) {
      Integer current = stack.pop();
      if (!visited.contains(current)) {
        visited.add(current);
      }

      List<Integer> neighbors = new ArrayList<Integer>(
          graph.getNeighbors(current));

      for (Integer neighbor : neighbors) {
        if (!visited.contains(neighbor)
            && graph.findEdge(current, neighbor).isUp()) {
          stack.push(neighbor);
        }
      }
    }

    return (visited.size() == graph.getVertexCount());

```

```
  }

  /*
   * calculate reliability of a single network state
   */
  private static double reliability(
      UndirectedSparseGraph<Integer, NetworkLink> graph) {
    double result = 1;
    List<NetworkLink> allLinks = new ArrayList<NetworkLink>(
        graph.getEdges());
    for (NetworkLink link : allLinks) {
      if (link.isUp()) {
        result *= link.getReliability();
      } else {
        result *= (1 - link.getReliability());
      }
    }

    return result;
  }

  /**
   * network reliability instance reporting
   */
  public void print() {
    int upCount = 0;
    int downCount = 0;

    System.out.println("Number of nodes: " + network.getVertexCount()
        + "; Number of links: " + network.getEdgeCount()
        + "; Number of network states: " + networkStates.size() + ".");

    for (UndirectedSparseGraph<Integer, NetworkLink> graph : networkStates
        .keySet()) {
      if (networkStates.get(graph)) {
        upCount++;
        // System.out.println(graph);
        // System.out.println(networkStates.get(graph));
        // visualizeGraph(graph);
      } else {
        downCount++;
      }
    }

    System.out.println("Network up states count: " + upCount
        + ", network down states count: " + downCount + ".");
  }

  /*
   * graph visualization
   */
  private void visualizeGraph(
      UndirectedSparseGraph<Integer, NetworkLink> graph) {
    Layout<Integer, NetworkLink> layout = new CircleLayout<Integer, NetworkLink>(
        graph);
    layout.setSize(new Dimension(200, 200));

    BasicVisualizationServer<Integer, NetworkLink> vv = new BasicVisualizationServer<Integer,
        NetworkLink>(
        layout);
    Transformer<Integer, Paint> vertexPaint = new Transformer<Integer, Paint>() {
      public Paint transform(Integer i) {
        return Color.YELLOW;
      }
    };

    vv.setPreferredSize(new Dimension(200, 200));
    vv.getRenderContext().setVertexFillPaintTransformer(vertexPaint);
```

```
    vv.getRenderContext().setVertexLabelTransformer(
        new ToStringLabeller<Integer>());
    vv.getRenderContext().setEdgeLabelTransformer(
        new ToStringLabeller<NetworkLink>());
    vv.getRenderer().getVertexLabelRenderer().setPosition(Position.CNTR);

    JFrame frame = new JFrame("Network Reliability Experiment");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(vv);
    frame.pack();
    frame.setVisible(true);
  }
}
```

A.2 Project3.java

```java
package project3;

import java.io.FileWriter;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

import com.opencsv.CSVWriter;

/**
 * Driver class for Project 3 of CS6385.001 Algorithmic Aspects of
 * Telecommunication Networks
 *
 * Experiment 1 calculate the given network reliability using exhaustive
 * enumeration for single link reliability ranging from 0 to 1.
 *
 * Experiment 2 calculate the network reliability after flipping randomly k
 * system states, k ranges from 0 to 30. To reduce the effect of randomness, for
 * each value of k, the experiment is carried out 1000 times and the resulting
 * reliability is averaged out.
 *
 * @author LiP
 *
 */
public class Project3 {
  /*
   * project configuration: network is represented with a complete undirected
   * graph with 5 nodes; In experiment 2, each link has the same reliability
   * of 0.9, and for each k the experiment is carried out 1000 times to be
   * average out.
   */
  public static final int NUM_OF_NODES = 5;
  public static final double FIXED_RELIABILITY = 0.9;
  public static final int FLIP_REPEATS = 1000;

  public static void main(String[] args) throws IOException {
    // output to csv file
    String csv_experiment1 = "data/project3_experiment1_result.csv";
    String csv_experiment2 = "data/project3_experiment2_result.csv";
    NumberFormat formatter = new DecimalFormat("0.000");

    // experiment 1
    CSVWriter writer = new CSVWriter(new FileWriter(csv_experiment1));
    String[] header = "Single link reliability, Network reliability"
        .split(",");
    writer.writeNext(header);

    for (double r = 0; r <= 1.01; r = r + 0.04) {
      NetworkReliability experiment1 = new NetworkReliability(
          NUM_OF_NODES, r);

      writer.writeNext((formatter.format(r) + "," + formatter
          .format(experiment1.networkReliability())).split(","));

      System.out.println("Link reliability is: " + formatter.format(r)
          + ", Network reliability is: "
          + formatter.format(experiment1.networkReliability()));
    }
    writer.close();

    // experiment 2
    writer = new CSVWriter(new FileWriter(csv_experiment2));
    header = "Flip k, Single link reliability, Flipped Network reliability"
        .split(",");
    writer.writeNext(header);
    NetworkReliability experiment2 = new NetworkReliability(NUM_OF_NODES,
```

```
        FIXED_RELIABILITY);

    System.out.printf(
        "Link reliability is: %5.2f, Network reliability is: %5.3f \n",
        FIXED_RELIABILITY, experiment2.networkReliability());

    for (int k = 0; k <= 30; k++) {
      double result = 0;
      for (int i = 0; i < FLIP_REPEATS; i++) {
        result += experiment2.networkReliabilityFlipped(k);
      }
      result = result / FLIP_REPEATS;
      writer.writeNext((k + "," + FIXED_RELIABILITY + "," + formatter
          .format(result)).split(","));
      System.out
          .printf("Flip k = %3d, Network reliability is: %5.3f, Flipped Network reliability is:
              %5.3f\n",
              k, experiment2.networkReliability(), result);
    }
    writer.close();
    experiment2.print();
  }
}
```

A.3 NetworkLink.java

```
package project3;

/**
 * Class to represent network links for the given task
 *
 * @author LiP
 *
 */
public class NetworkLink {

  // link reliability of a NetworkLink instance
  private double reliability;
  // indicates if the link instance is up or down
  private boolean status;

  /*
   * constructor, all NetworkLinks are up by default
   */
  public NetworkLink(double d) {
    this.reliability = d;
    this.status = true;
  }

  /*
   * Deep copy of a NetworkLink instance
   */
  public NetworkLink(NetworkLink o) {
    this.reliability = o.reliability;
    this.status = o.status;
  }

  /*
   * getters and setters
   */
  public double getReliability() {
    return reliability;
  }

  public boolean isUp() {
    return status;
  }

  public void setReliability(double d) {
    this.reliability = d;
  }

  public void setLinkUp() {
    this.status = true;
  }

  public void setLinkDown() {
    this.status = false;
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Object#toString() used for reporting and visualization
   * purposes
   */
  @Override
  public String toString() {
    if (status) {
      return "UP";
    } else {
      return "DOWN";
```

```
        }
    }
}
```