CS 6385: Project 2

Algorithmic Aspects of Telecommunication Networks

Submitted by:

Anunitya Alla (axa210108)

Task1:

Algorithm description:

1. There are 2^n possible states for a graph with n edges. To represent each state, we generate binary numbers from 0 to 2^n - 1. Each digit in the binary number represents the status of corresponding edge: 1 for UP and 0 for DOWN. When all edges are DOWN, the binary number is 0, and when all edges are UP, the binary number is (2^n) - 1.

2. For each possible state, we build a graph based on the edge status given in the state. The n-digit binary number represents the state, and each bit corresponds to an edge.

3. We perform a Depth-First Search (DFS) on the graph to check if the given state results in a connected graph. If the graph is connected, it means the state is valid and contributes to the network reliability calculation.

4. If the graph is connected for a given state, we calculate the reliability score of that graph. The reliability score is based on the given value of p (probability) and the number of UP and DOWN edges in the state. We use the formulas provided in the reliability class to calculate the reliability.

5. We repeat the process for all possible states, calculating the reliability for each connected state.

6. Finally, we sum up the reliabilities of all connected states to get the combined reliability score for the given value of p. This overall network reliability score represents the probability that the entire network is functioning correctly given the probability p of each edge being UP.

With this method, we exhaustively explore all possible states, evaluate their reliabilities, and combine them to obtain the network reliability.

Algorithm:

Step 1: Generating Possible States

1. Given a graph with n edges, the number of possible states is 2^n.
2. Generate all possible binary strings of length n, representing the status of each edge (UP = 1 or DOWN = 0).
3. For each binary string, it represents a potential state of the system. Store all these states in a list.

Step 2: Filter the connected states from all the possible states

1. Iterate through the list of possible states.
2. For each state, build a graph representation based on the edge status given in the state (UP or DOWN).
3. Use the Depth-First Search (DFS) algorithm to check if the graph is connected (all vertices are reachable from any starting vertex).
4. If the graph is connected, proceed to calculate the reliability of the state.

Step 3: Calculating Reliability for a State

1. Initialize the reliability score (Rscore) to 1.0.
2. Count the number of UP edges (nums1) and DOWN edges (nums0) in the current state.
3. If there are UP edges (nums1 > 0), multiply Rscore by p (the probability of an edge being UP) raised to the power of nums1.
4. If there are DOWN edges (nums0 > 0), multiply Rscore by (1 - p) raised to the power of nums0.
5. The resulting Rscore represents the reliability of the current state.

Step 4: Accumulating Network Reliability

1. Sum up the reliabilities of all connected states to get the overall network reliability.
2. Return the network reliability as the final result.

Task 2:
Description of code:

1. ### connectedComponents.py:

This connectedComponents.py defines a class *graph* that contains methods for working with graphs and finding connected components.

1. **buildG(state : str, numV : int, edges : list[tuple]) -> defaultdict(list):**
   This method builds a graph based on the given state, which represents the presence or absence of edges in the graph. It takes three arguments:
   - *state*: A binary string representing the presence or absence of edges (1: edge present, 0: edge absent).
   - *numV*: An integer representing the number of vertices in the graph.
   - *edges*: A list of tuples, where each tuple contains two integers representing the vertices that form an edge.
   The method returns a defaultdict(list) representing the graph as an adjacency list.

2. **isConn(grph : defaultdict(list), numV : int) -> bool:**
   This method checks whether the graph represented by the adjacency list grph is connected or not. It performs a Depth-First Search (DFS) starting from vertex 0 and checks if all vertices are visited. It takes two arguments:
   - *grph*: A defaultdict(list) representing the graph as an adjacency list.
   - *numV*: An integer representing the number of vertices in the graph.
  The method returns True if the graph is connected, and `False` otherwise.

3. **getConnectedComponents(numV : int, edges : list[tuple], PS: list[str]) -> list[str]:**
   This method finds the connected components by checking all possible states (PS) of the graph. It takes three arguments:
   - *numV*: An integer representing the number of vertices in the graph.
   - *edges*: A list of tuples, where each tuple contains two integers representing the vertices that form an edge.
   - *PS*: A list of binary strings representing all possible states of the graph.
  The method returns a list of binary strings representing the connected states.

4. **getPS(n : int):**
  This method generates all possible states of a graph with n edges. Since n edges can either be present or absent, there are 2^n possible states. It takes one argument:
   - *n*: An integer representing the number of edges in the graph.
   The method returns a list of binary strings representing all possible states of the graph.

2. **GetReliability.py**

This GetReliability.py defines a class `reliablity` that contains a method for calculating the reliability of all connected states for a given probability value p.

1. **calcReliablity(connstates : list[str], p: float) -> float:**
 This method calculates the reliability of all the connected states for a given probability p. It takes two arguments:
 - connstates: A list of binary strings representing the connected states of the graph.
 - p: A floating-point value representing the probability of an edge being "UP"

The method returns the sum of reliabilities of all connected states. Inside the method
 - The `Rsum` variable is initialized to 0, which will store the sum of reliabilities of all connected states.
 - For each state in connstates, the method calculates its reliability and adds it to the Rsum.
 - The reliability of a state is calculated as follows:
 - Rscore is initialized to 1.0, which will be used to accumulate the reliability score.
 - nums1 is the count of "UP" edges (edges with value '1') in the state.
 - nums0 is the count of "DOWN" edges (edges with value '0') in the state.
 - If there are "UP" edges (nums1 > 0), the method multiplies Rscore with p and raises it to the power of nums1.
 - If there are "DOWN" edges (nums0 > 0), the method multiplies Rscore with (1-p) and raises it to the power of nums0.
 - The Rscore now represents the reliability of the current state.
 - After calculating the reliability for each state, the method adds up all the reliabilities to get the combined reliability sum (Rsum).
 - Finally, the method returns the Rsum, which is the total reliability of all connected states.

This code is meant to calculate the reliability of a network with given states and probabilities. The higher the reliability, the more likely the network is to remain connected and operational under different conditions.

3. **main.py**

main.py is a Python script that calculates the reliability of a graph for different p-values and then plots the p-value vs. reliability graph using matplotlib. It uses custom functions from other modules (ConnectedComponents and GetReliability) to handle the graph calculations. Additionally, it uses the tabulate library to format and print the data to the terminal.

1. The script defines two functions: `plotG` for plotting the p-value vs. reliability graph and getE to extract the list of edges from the adjacency matrix.

2. . Inside the main block:

   - The input adjacency matrix aM represents the graph connections between vertices.
   - The number of vertices numV is determined from the size of the adjacency matrix.
   - The number of edges numE is calculated by summing all entries in the adjacency matrix and dividing by 2 (since each edge is counted twice in the matrix).
   - The list of edges edgeList is extracted using the getE function.
   - The list of all possible states in the graph possStates is obtained using a custom function graph.getPS.
   - The list of connected states connStates is calculated using another custom function graph.getConnectedComponents, which provides the states that form connected graphs.

3. The script then creates Xvalues and Yvalues lists to store the p-values and corresponding reliability scores, respectively.

4. It iterates through p-values from 1 to 0.05 (in increments of 0.05), calculates the reliability score using a custom function reliability.calcReliability with the connStates and the current p-value, and appends the result to the Yvalues list.

5. It calls the plotG function to create and display the p-value vs. reliability graph.

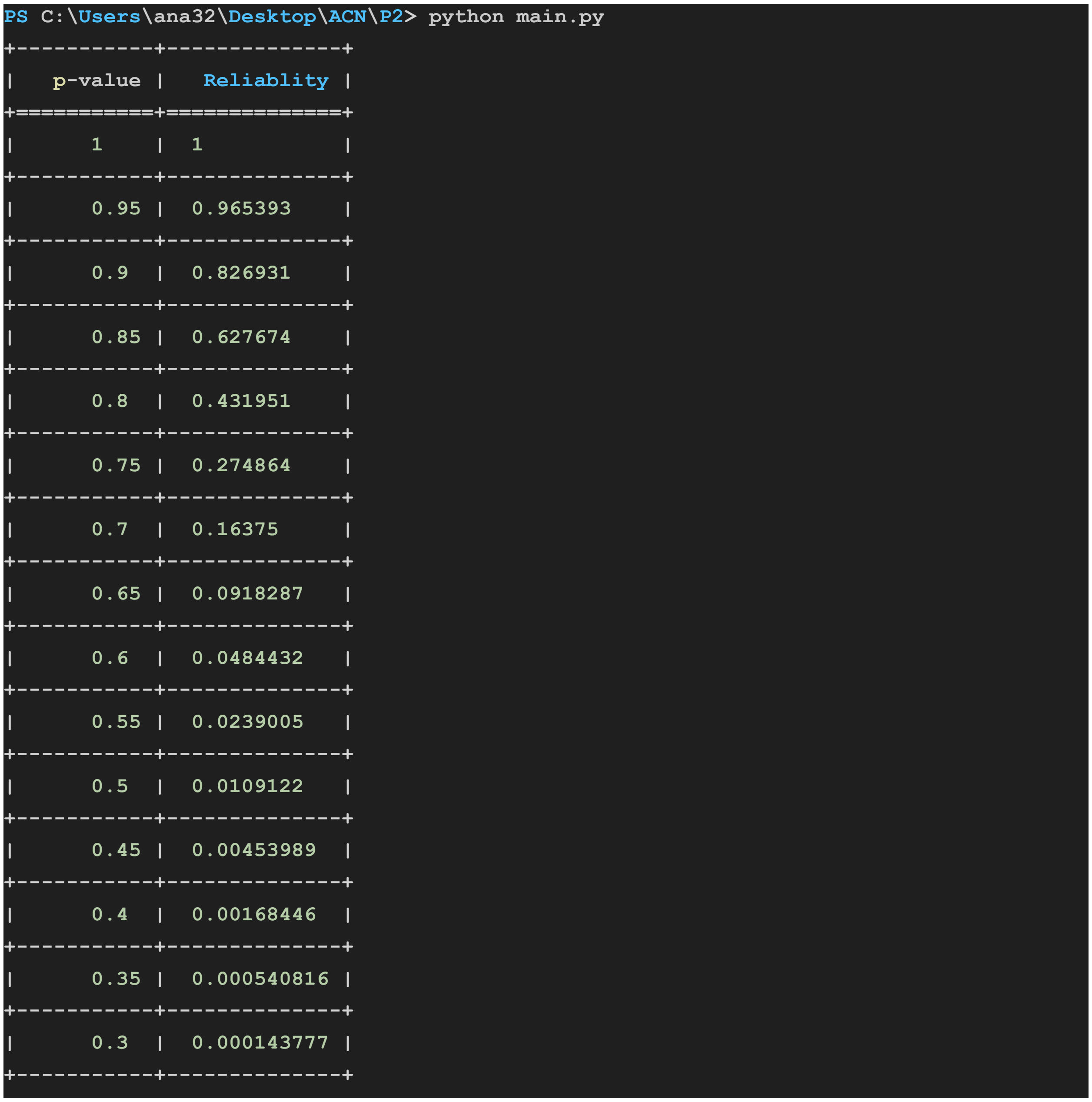
6. Finally, it creates a table (using `tabulate`) with 20 data points of p-values and their corresponding reliabilities, then prints the table to the terminal.

Task 3:

Output on the terminal:

The output shows the formatted table with p-values and their corresponding reliability scores. Each row in the table represents a p-value along with its reliability score.

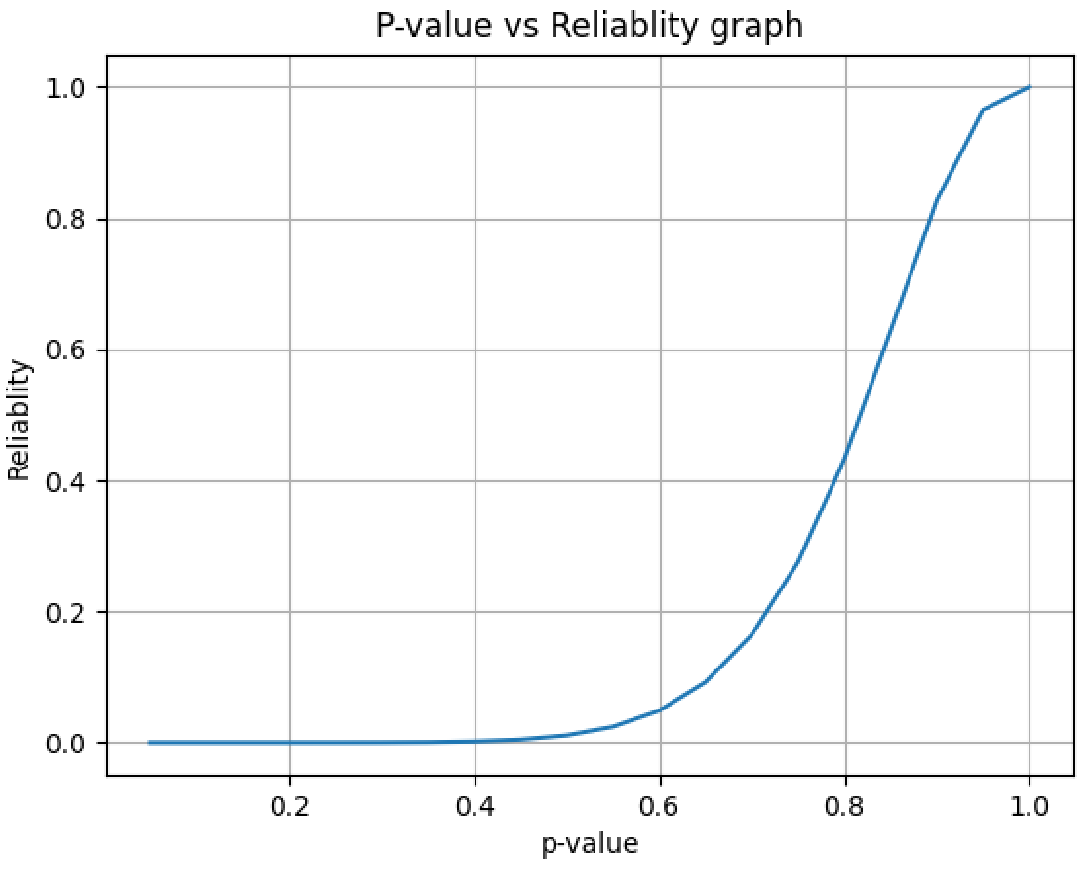
For example, the first row shows that for a p-value of 1.0, the reliability is 1.0, which means the graph is fully reliable (100% reliable). As the p-value decreases, the reliability score also decreases, indicating that the graph becomes less reliable.



```
PS C:\Users\ana32\Desktop\ACN\P2> python main.py
+-----------+--------------+
|   p-value |   Reliablity |
+===========+==============+
|      1    |  1           |
+-----------+--------------+
|      0.95 |  0.965393    |
+-----------+--------------+
|      0.9  |  0.826931    |
+-----------+--------------+
|      0.85 |  0.627674    |
+-----------+--------------+
|      0.8  |  0.431951    |
+-----------+--------------+
|      0.75 |  0.274864    |
+-----------+--------------+
|      0.7  |  0.16375     |
+-----------+--------------+
|      0.65 |  0.0918287   |
+-----------+--------------+
|      0.6  |  0.0484432   |
+-----------+--------------+
|      0.55 |  0.0239005   |
+-----------+--------------+
|      0.5  |  0.0109122   |
+-----------+--------------+
|      0.45 |  0.00453989  |
+-----------+--------------+
|      0.4  |  0.00168446  |
+-----------+--------------+
|      0.35 |  0.000540816 |
+-----------+--------------+
|      0.3  |  0.000143777 |
+-----------+--------------+
|       0.25 |   2.95695e-05 |
+-----------+--------------+
|       0.2  |   4.19858e-06 |
+-----------+--------------+
|       0.15 |   3.32537e-07 |
+-----------+--------------+
|       0.1  |   9.1e-09      |
+-----------+--------------+
|       0.05 |   1.86523e-11 |
+-----------+--------------+
```

P-values vs the reliability score graph:

As the p-value decreases, the reliability score also decreases, indicating that the graph becomes less reliable.



Readme:

The codebase consists of 3 files:

1. connectedComponents.py - contains methods for graph based calculations
2. GetReliablity.py - contains methods to calculate combined reliablity
3. Main.py - contains the python script

Dependencies:

Install the following dependencies using the commands below.

```
pip install tabulate
pip install matplotlib
```

To run:

In the terminal enter the following commands to run the code.

```
$ python main.py
```

CodeBase :

1. **connectedComponents.py**

```
from collections import defaultdict

class graph:

      # Builds a graph for the given state
    @staticmethod
    def buildG(state : str, numV : int, edges : list[tuple]) ->
defaultdict(list):

        def addEdge(u,v):   # Add an edge from vertex u to v
            grph[u].append(v)
            grph[v].append(u)

        grph = defaultdict(list)
        numE = len(edges)

        for i in range(numE):
            if state[i] == '1':
                u = edges[i][0]
                v = edges[i][1]
                addEdge(u,v)

        return grph

        # Input the graph built from the current stateand outputs if the
graph is connected
    @staticmethod
    def isConn(grph : defaultdict(list), numV : int) -> bool:

        traversed = [False for i in range(numV)] # Store if the vertex is
visited or not
        stack = []

        traversed[0] = True
        stack.append(0)

        while stack: # Do DFS
            V = stack.pop()

            for U in grph[V]:
                if traversed[U] == False:
                    traversed[U] = True
                    stack.append(U)

        if not (False in traversed):
            return True

        return False
```

```
 # Input all the possible states and return the states that give a
connected graph
    @staticmethod
    def getConnectedComponents(numV : int, edges : list[tuple], PS:
list[str]) -> list[str]:

        conn = []

        for state in PS:
            grph = graph.buildG(state, numV, edges) # Build a graph for
the current state

            if graph.isConn(grph, numV): # check if the graph is connected
                conn.append(state)

        return conn


    @staticmethod
    def getPS(n : int):

        numPS = int(2 ** n) # Number of all the possible states = 2 **
(number of edges)
        possibleStates = []

        binFormat = '{0:0' + str(n) + 'b}'

        for i in range(numPS):

            state = binFormat.format(i) # Convert the integer i to state
in binary value
            possibleStates.append(state) # Add te state to the all
possible states

        return possibleStates
```

2. **GetReliablity.py**

```python
class reliablity:

    @staticmethod
    def calcReliablity(connstates : list[str], p: float):   # Calculate
reliablity of all the connected states for a given P-value

        Rsum = 0    # store Reliablity sum

        for state in connstates:

            Rscore = 1.0

            nums1 = state.count('1') # Count number of edges are UP
            nums0 = state.count('0') # Count number of edges are down

            if nums1 > 0:   # If there exists "UP" edges, then multiply
p-value with Rscore and raise it to the power of number of "UP" edges
                Rscore = (Rscore *p) ** nums1

            if nums0 > 0:    # If there exists "DOWN" edges, then multiply
(1-p) with Rscore and raise it to the power of number of "DOWN" edges
                Rscore = (Rscore *(1-p))** nums0

            Rsum = Rsum  + Rscore   # Add the reliablity of each connected
state to get combined reliablity sum

        return Rsum
```

3. **main.py**

```python
import matplotlib.pyplot as Mplt
from tabulate import tabulate

from ConnectedComponents import graph
from GetReliablity import reliablity
```

```
def plotG(Xvalues, Yvalues): # Plots the p-value Vs Reliablity graph
    Mplt.plot(Xvalues, Yvalues)
    Mplt.xlabel('p-value')
    Mplt.ylabel('Reliablity')
    Mplt.title('P-value vs Reliablity graph')
    Mplt.grid(True)
    Mplt.show()

def getE(aM, numV): # Returns the list containing tuples (u,v) for rach
edge from vertex u to v
      edges = []
      for i in range(numV):
            for j in range(i+1, numV):
                  if aM[i][j] == 1:
                        edges.append((i,j))


if __name__ == "__main__":

      aM = [[0,1,1,0,0,1,0],
            [1,0,1,1,0,0,0],
            [1,1,0,0,1,1,0],
            [0,1,0,0,1,0,0],
            [0,0,1,1,0,0,1],
            [1,0,1,0,0,0,1],
            [0,0,0,0,1,1,0]] # Input adjacency matrix


      numV = len(aM)    # number of vertices

      edgeSUM = 0
      for i in range(numV):
            edgeSUM += sum(aM[i])

      numE = int(edgeSUM/2)   # number of edges

      edgeList = getE(aM, numV)     # get the list of edges as tuples
(u,v)
```

```
      possStates = graph.getPS(numE)      # get the list of all possible
states in the graph

      connStates = graph.getConnectedComponents( numV,
edgeList,possStates)   # get the list of states that give connected graph


      Xvalues = [p/100.0 for p in range(100, 0, -5)]
      Yvalues = []

            # Calculate reliablity for p-value in [0.05,1]
      for p in range(100, 0, -5):
            pvalue = p/100.0
            Rscore = reliablity.calcReliablity(connStates, pvalue)
            Yvalues.append(Rscore)


      plotG(Xvalues, Yvalues) # plot the p-value Vs Reliablity graph


            # Print the p-value and Reliablity sum to terminal
      data = []
      for i in range(20):
            data.append([Xvalues[i], Yvalues[i]])

      headers = ["p-value", "Reliablity"]
      print(tabulate(data, headers=headers, tablefmt="grid"))
```