Project 2
Sudarshana Jagadeeshi

Contents

Objective

We are given a fully connected undirected graph with N nodes and no self-loops. We are also given a p-value, which is a base probability used to compute the reliability of the M edges.

The problem at hand is to use the reliabilities of the edges to compute the reliability of the overall network. We then further vary p and introduce random failures and examine their effects.

Approach

To initialize the p array, we perform the following:

$$p_i = p^{\lceil d_i/3 \rceil}$$

Where d_i is the dth digit of the student ID.

We first generate the various states of the graph as described in the "Generation of states" section.

After all the states are enumerated, we can assign up/down values to them. We want to assign an up state iff the graph is connected. Checking if the graph is connected or not is described in the "Finding Connectivity" section.

At last, we can compute the total reliability of the network. Refer to the section "Computing Network Reliability" in the algorithms section. The results are plotted using the matplotlib library.

Algorithms

Generation of states

The algorithm returns an bitarray, but this can easily be converted to a graph. A 0 indicates that an edge is down, and 1 that it is up.

The generation is done in the genstates() function by running a nested loop. In the outer loop, we increment i from 0 to 2^M, where M is the number of edges. Then, i is examined. Notice that the first digit stays 0 from 0 to 2^M-1, then flips to 1. The second digit flips twice as often, creating four sections where the digit is 0, 1, 0, 1. By converting i into a section number and checking if it is even, we know the value of the digit.

Pseudocode:

```
array=[] #holds the generated states

for i=0 to 2^M:
    digittoset =0 #this is an index from 0 to M-1
    sectionsize= 2 ^ M-1

    while digittoset < M:
        if i / sectionsize is even:
            array[i][digittoset]=0
        else:
            array[i][digittoset]=1

    sectionsize /= 2
    digittoset += 1
```

Finding connectivity

One can run a traversal algorithm of choice (BFS/DFS) between all pairs of nodes. If any of these runs fail, then the graph is not connected.

In the code, networkx.is_connected() is used. It calls plain_bfs under the hood to walk the graph.

Pseudocode:

```
for i in nodes:
    for j in nodes:
        if i == j: #no self loops
            skip
        else:
            if bfs(i,j) == inf: #i.e. bfs fails
                return false
    return true
```

Computing Network Reliability

Computing the total network reliability is done by summing over the reliability of the up-states. One only needs to sum over the up-states because down-states have 0 reliability by default.

To find the reliability of a single state, multiply the p-values of all the edges in that state that are up. Then take multiply that result by 1-p values for all the edges that are down.

Pseudocode:

```
total=0 #overall reliability

for each state:
    thisstate=1 #reliability of this state

    for edge in that state:
        if edge is up:
            thisstate *= edges' p
        else:
            thisstate *= 1- edges' p

    total += thisstate #running sum
```

Results

Figures

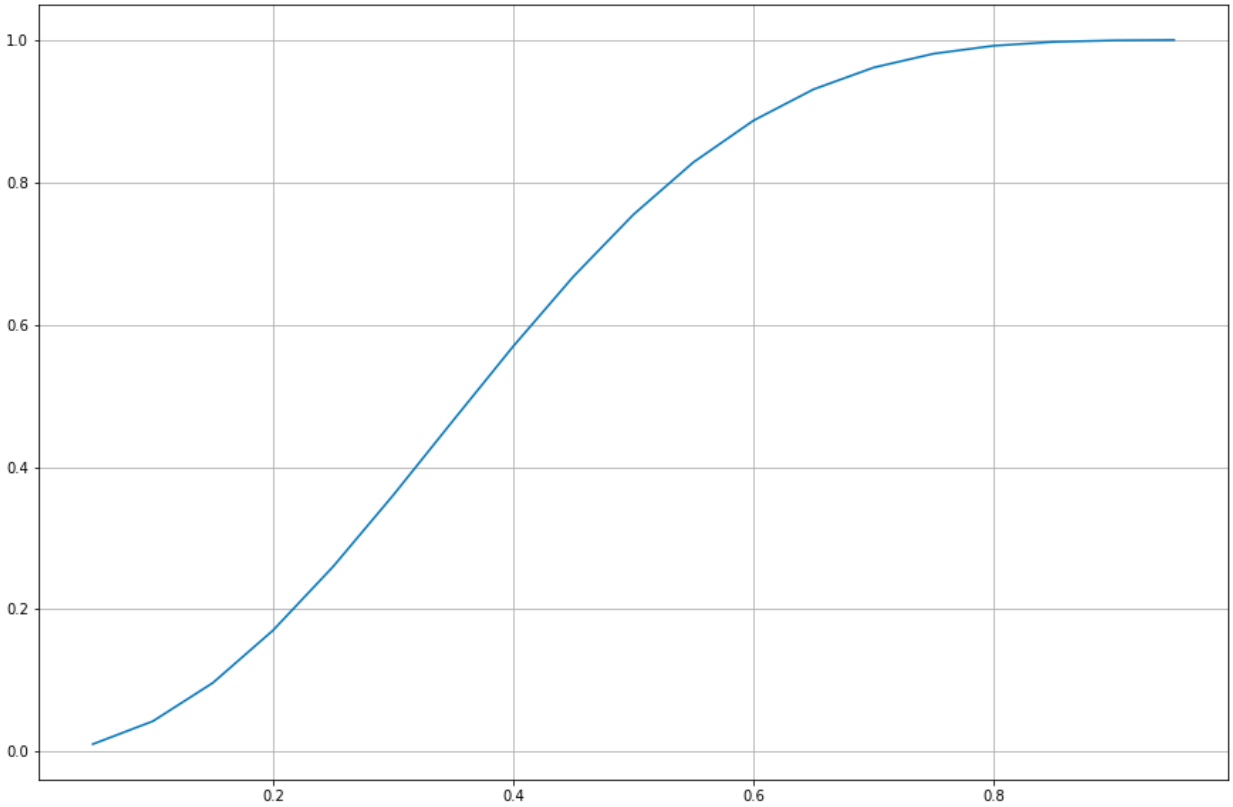

Fig 1.1 Overall Reliability graph for 0.05 <= p <= 1

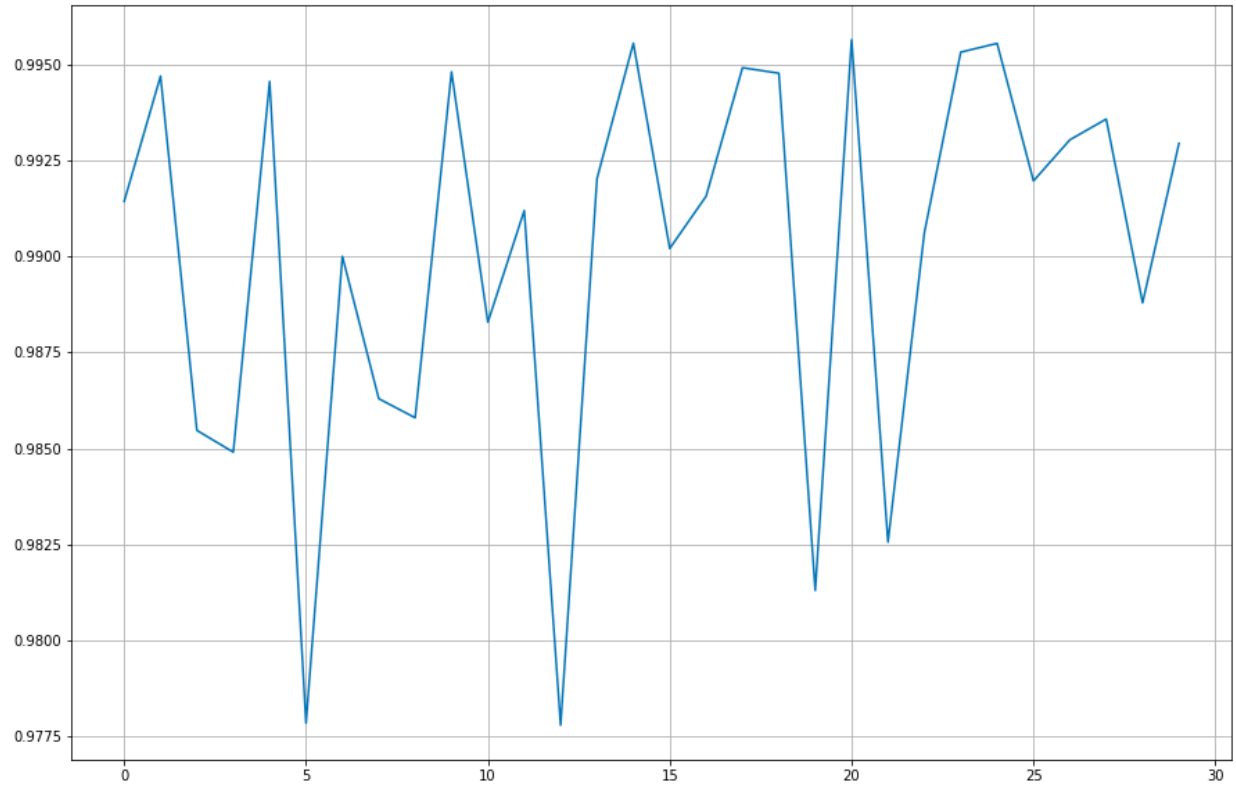

Fig 1.2 Reliability graphs for various k values, p=0.9, numiter= 30

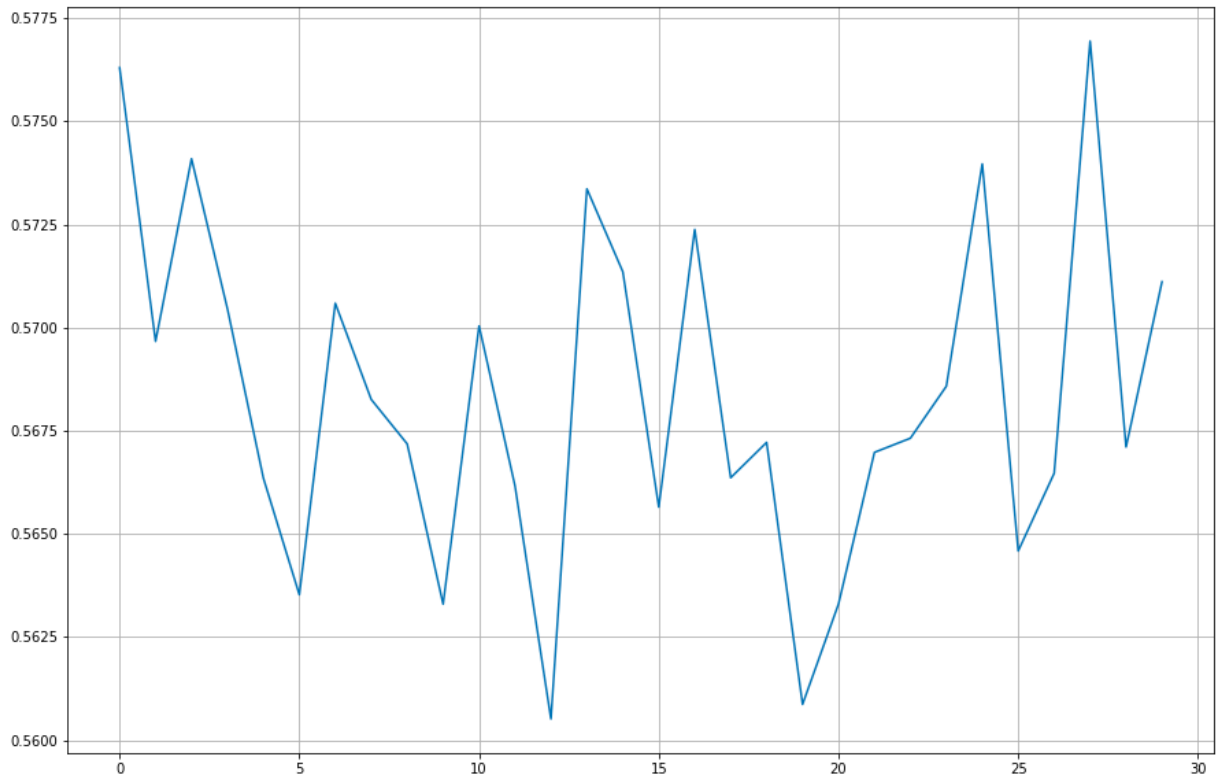

Fig 1.3 Reliability graphs for various k values, p=0.4, numiter= 30

Analysis

For the p-experiment, the reliability follows a sigmoid shape. Overall reliability increases with p as expected. 50% reliability is reached at a base p of only 0.35. Past p=0.8, the p values are already so high that few further gains can be made. This is because of my student ID, which has two zeros. That means that links 0 - 2 and 1 - 4 are always up. Also, there is only one digit from 7-9 in my ID. The low digits are good- higher numbers in the exponent will decrease the p-values of the respective links.

For the k-experiment, the main conclusion is that a pattern is hard to observe. It is not correct to say that the initial value of k=0, where no flipping is

done, has the most reliability. It is also not correct to say we observe more low outliers than high. When the experiment is repeated with a fixed p value of 0.4, we can see there is truly no trend.

Appendix

Support for Readability, Validation, Debugging, Parameterization

Readability is supported by the helpful comments, which help decipher the more confusing lines of code. The code is also separated into functions.

Validation is supported by the prompt error messaging, which exits the program with a custom message if parameters are supplied incorrectly or if a algorithmic error occurs.

Debugging is supported by the debugmsg function, which is an alternative to the print statement that allows the user to rapidly turn on and off debug messages in order to resolve issues.

Tha ability to parametrize is accommodate by the variables section near the bottom, allowing the user to change variables quickly and in one place.

Annotated Source Code

A link to the code can be found here: https://colab.research.google.com/drive/1RzBmLZkEjg9ihP_ucI2trdRRRFbVcvrp?usp=sharing

```python
import networkx as nx
import matplotlib.pyplot as plt
import random
import math
import copy
import sys
import numpy as np

N= 5
M= ( (N)* (N-1) ) //2 #numedges
p_generator= [2,0,2,1,4,6,0,2,4,9]; #my student ID
DEBUGMODE=0 #1 means you will see prints

#* args for number of args
def debugmsg(*args):
  if (DEBUGMODE):
    for arg in args:
      print(arg, end= " ") #no newline unless you ask for it using \n

#@title arrtograph
#input: a len M graph with arr[i] = 1 where
#means edge i exists

def arrtograph(state):
  tempG = nx.Graph()
  tempG.add_nodes_from(range(N))

  if (state[0] == 1):
    tempG.add_edge(0,1)
  if (state[1] == 1):
    tempG.add_edge(0,2)
  if (state[2] == 1):
    tempG.add_edge(0,3)
  if (state[3] == 1):
    tempG.add_edge(0,4)
  if (state[4] == 1):
    tempG.add_edge(1,2)
  if (state[5] == 1):
    tempG.add_edge(1,3)
  if (state[6] == 1):
    tempG.add_edge(1,4)
  if (state[7] == 1):
    tempG.add_edge(2,3)
  if (state[8] == 1):
    tempG.add_edge(2,4)
  if (state[9] == 1):
    tempG.add_edge(3,4)

  return tempG
```

```
    return tempG

#@title genstates

 def genstates():
  states= [[0 for i in range(M)] for j in range(2**M)] #create a M x 2^M array

  for i in range(len(states)):
    tempc=0 #the digit that we are currently setting
    tempM= 2 ** (M-1) #the width of the section for that digit

    while (tempM != 0.5): #i.e. after the last digit has been filled
      debugmsg(i, tempM, (i//tempM) % 2 == 0, "\n")
      if ( (i//tempM) % 2 == 0  ): #even section
        states[i][tempc]=0
      else:
        states[i][tempc]=1
      tempM /= 2
      tempc += 1

  return states

#@title run_expt

def run_expt(inputp, inputk, typeexp):

  if (typeexp != "K" and typeexp != "k" and typeexp != "P" and typeexp != "p"):
    sys.exit("The valid experiment types are K and P.")

  #this experiment runs
  #CONSTANTS
  BASEP= inputp

  G = nx.Graph()
  G.add_nodes_from(range(N))

  p = [0 for i in range(M)] #M-long zero array

  for i in range(len(p)):
    p[i] = BASEP ** math.ceil((p_generator[i]/3.0) )
  #print("p array", p)

  #make graph
  for i in range(N):
      for j in range(N):
          if (i != j):
            G.add_edge(i, j)

  debugmsg(G)
```

```
    debugmsg(G)

    #generate states
    states= genstates()
    debugmsg(len(states), len(states[0]))

    #assign up/down values
    isup = []

    for state in states:
      tempG=arrtograph(state); #convert to graph
      isup.append(nx.is_connected(tempG))

    debugmsg(len(isup))

    #simulate the flipping of certain edges for K experiment
    if (typeexp == "K" or typeexp == "k"):
      index= random.sample(range(0, 2 ** M), inputk);
      for i in index:
        isup[i] = abs(isup[i] - 1) #flip

    #find reliability of all the states
    totalr=0

    for i in range(len(states)):
      thisr = 1 #this states reliability
      if (isup[i]):
        for j in range(len(states[i])):
          edge= states[i][j]
          if (edge == 0): #state down
            thisr *= (1-p[j])
          elif (edge == 1): #state up
            thisr *= p[j]
        debugmsg(thisr, " ")
        totalr += thisr #add to running total

    return totalr

  #regular range doesn't accept float values
  def floatrange(low, high, step):
    arr=[]

    while (low <= high):
      arr.append(low)
      low += step

    return arr

  def main():
```

```
145 def main():
146     #======VARIABLES=======
147     STARTP= 0.05
148     ENDP= 1
149     STEPP= 0.05
150
151     STARTK= 0
152     ENDK= 20
153     STEPK= 1
154     FIXEDP= 0.4
155     NUMITER=30 #how many times to run the k-expt for each k-value
156     #SEED=2001
157
158     #======RUN P-EXPERIMENTS======
159     results=[]
160     for myp in floatrange(STARTP, ENDP, STEPP):
161       results.append(run_expt(myp, 0, "P", SEED))
162
163     print("RESULTS OF P-EXPERIMENT")
164     print(results)
165
166     #plot
167     plt.figure(figsize=(15, 10))
168     plt.plot(floatrange(STARTP, ENDP, STEPP), results)
169     plt.grid()
170
171     #======RUN K-EXPERIMENTS======
172     results2= []
173
174     for i in range(NUMITER):
175       results2.append([])
176       for k in floatrange(STARTK, ENDK, STEPK):
177         results2[i].append(run_expt(FIXEDP, k, "K"))
178
179     print(results2)
180     master =[]
181     master= np.average(results2, axis=1) #average on each element
182     print(len(master))
183     print("RESULTS OF K-EXPERIMENT", master)
184
185     #plot
186     plt.figure(figsize=(15, 10))
187     plt.plot(master)
188     plt.grid()
189
190
191     print(sum(master)/len(master)) #get the average of the k-experiments
192
193 if __name__ == "__main__":
194     main()
```

README

This program was written in python. Just type python3 <filename>.py

You may have to install the requisite libraries.

Sources

[1] “More Complex Configurations”, Class Handout
[2] https://networkx.org/, NetworkX documentation
[3] https://matplotlib.org/, matplotlib documentation
[4] https://numpy.org/doc/, numpy documentation