Prasanna Bartakke
EE19B106
Electrical Engineering
@prasanna648:matrix.org
ee19b106@smail.iitm.ac.in

Module 1 Assignment Version 2

CS 6858 : Jul – Nov, 2022 : John Augustine
Due : 2 PM on Friday, September 16, 2022
(Submission via turnitin)

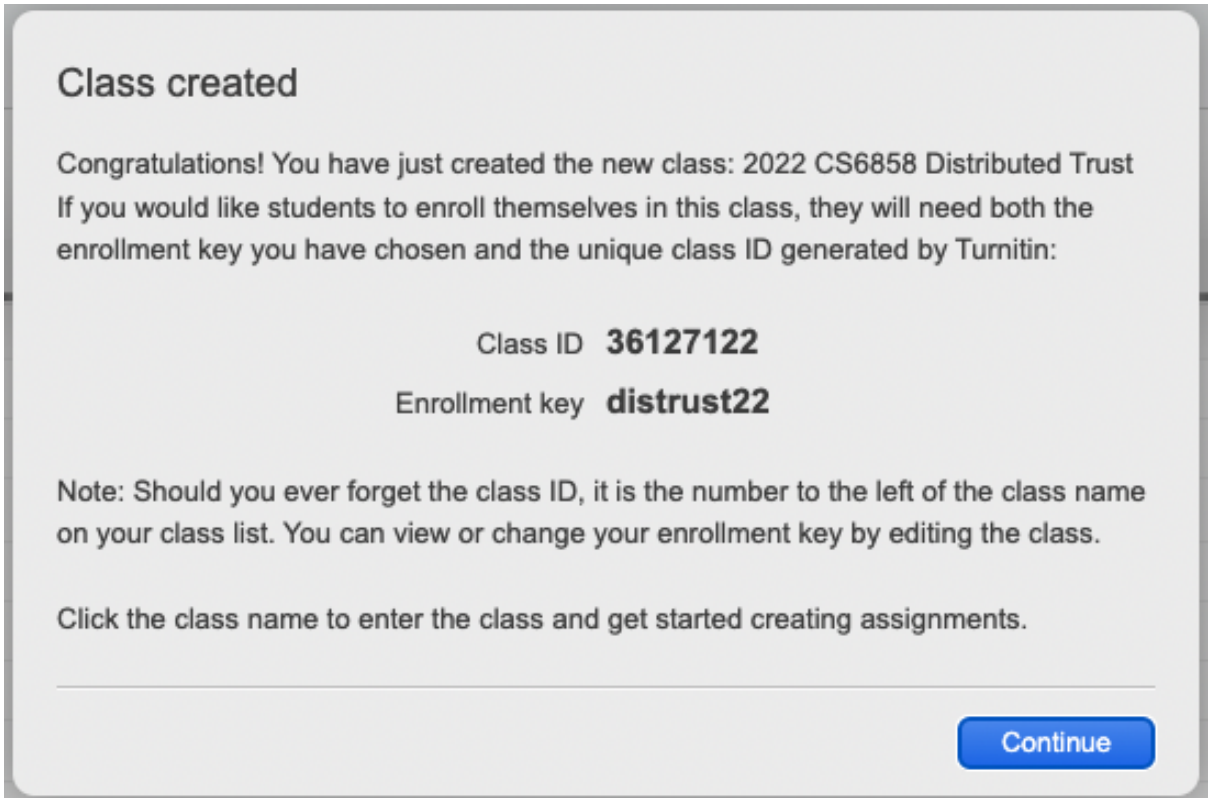


Figure 1: Turnitin Details.

Contents

Problem 1. Tossing Random Coins With PKI

In this problem, we will design attack strategies to foil protocols for tossing global coins. Consider a synchronous message passing complete network with n nodes $\{1, 2, \ldots, n\}$; you may assume PKI. Nodes can perform any required computation instantaneously at any time and this includes tossing coins to generate uniformly random bits. Up to b nodes are Byzantine for some $b < n$; b is common knowledge among the nodes. Time runs synchronously starting from round 1. Within each round, each node can send a finite-sized message to every other node during the first half of the round; nodes can also refrain from sending anything. Any good node must send the same message to all other nodes (within a round) whereas bad nodes can send different messages to different nodes. In the second half of the round, the messages will be delivered. We are exploring the challenges in designing algorithms that guarantees that

(agreement) all good nodes output the same bit r and

(valid randomness) the probability that $r = 1$ is exactly $1/2$.

You can imagine the output bit at each node to be a "write-once" bit that cannot be edited after the first (and only) write operation.
Your goal is to design attack strategies for the following algorithms. The attack strategies must be simple, precise and succeed with the ***smallest*** possible value for b.

Part 1. Using the Dolev-Strong Byzantine Broadcast protocol.

Byzantine nodes can affect the probability of the output. During the Byzantine broadcast, all the honest nodes reveal the value of their random bit to everyone after the first round. Let the xor of the random bits of all the good nodes be x. The group of byzantine nodes can work together and develop a sequence of bits such that the xor of the bits received by the honest nodes from byzantine nodes is $1 \bigoplus x$. The final output bit of the honest nodes is $1 \bigoplus x \bigoplus x = 1$. Thus, the valid randomness property is violated as the probability that $r = 1$ is not $\frac{1}{2}$.

Part 2. Relying on $b + 1$ signatures

Suppose we have 3 nodes, N_0, N_1 and N_2. N_0 and N_1 are honest, whereas N_2 is byzantine. Suppose the randomly generated input bit of N_0 and N_1 is 0, and the byzantine node N_2 sends 1 to both nodes in the first round.
Following are the arrays formed by the honest nodes at the end of the first round:
$R_0 = [(0)_0, (0)_1, (1)_2]$, $R_1 = [(0)_0, (0)_1, (1)_2]$.
In the second round, N_0 and N_1 send their arrays to each other. The byzantine node sends the array $([(0)_0, (0)_1, (1)_2])_2$ to N_0 and $([(0)_0, (0)_1, (0)_2])_2$ to N_1.

We can see that $R^*[0]$ and $R^*[1]$ decided by both N_0 and N_1 will be the same bit 0. But while deciding $R^*[2]$, N_0 will see that all the values for the 3rd bit are 1, so $R^*[2]$ for N_0 is 1. But, N_1 will see both 0(from N_0 and N_1) and 1(from N_2); thus, $R^*[2]$ for N_1 will be 0(assuming $\perp$ is 0). Thus the output bit after xor for N_0 will be 1, and for N_1 will be 0. Thus, the agreement property is not satisfied, and a single byzantine node was able to launch a successful attack.

Problem 2. Leader Election by Passing a Token

Part 1. Analysis – Consistency

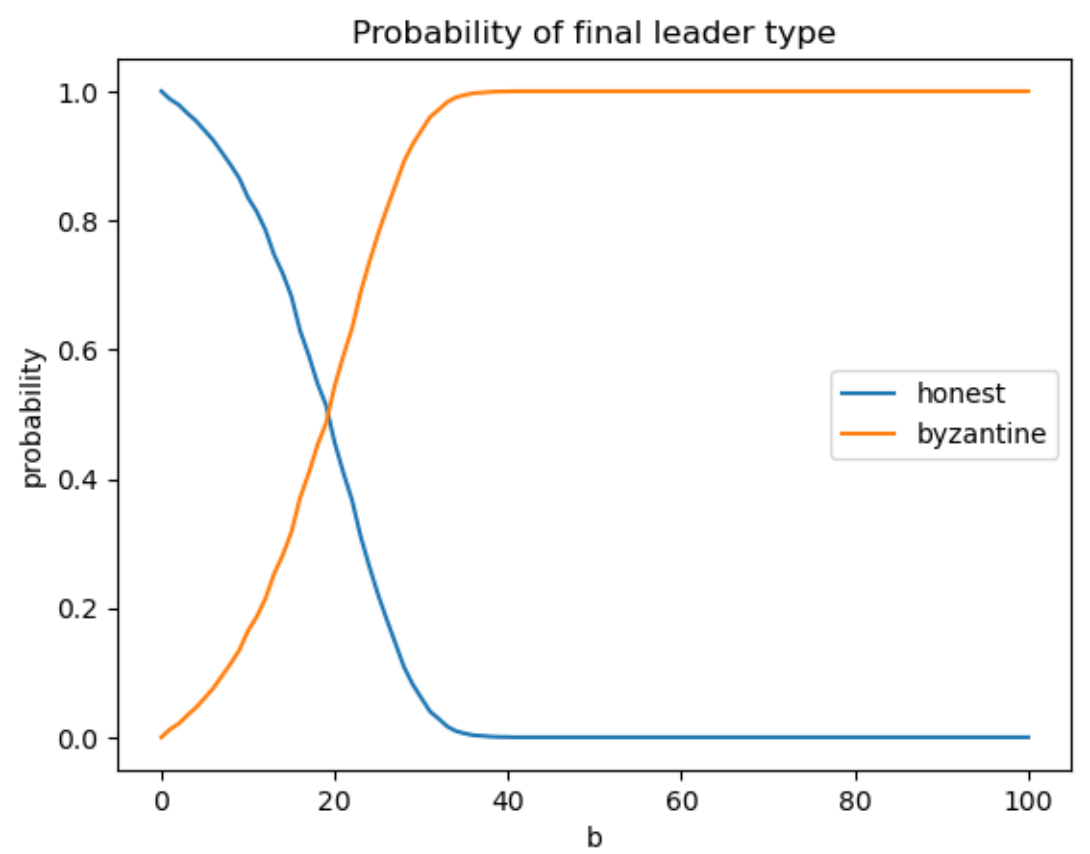
We can claim that all the honest nodes will agree on the same leader in each round. This is because all honest nodes agree on node 1 as the leader at the start of the protocol. As all the nodes can only broadcast the same message to everyone, all honest nodes get the same value of j broadcasted by the current round leader. If the current leader is honest, the broadcasted j will be a valid next-round leader, and all the honest nodes will consider that node to be the next round's leader. If the current leader is byzantine, then the j broadcasted might not be a valid next-round leader. But as all the honest nodes know which nodes are the valid next-round leaders, they can select the lowest id node if the broadcasted j is not a valid next-round leader. This will also be the same for all the honest nodes; thus, at each round, all the honest nodes will agree on the same node. Hence, at the end of the protocol, all the honest nodes will declare the same node as the leader.

Part 2. Analysis – Byzantine Strategy

From the previous discussion, we can see that it is not possible for the byzantine nodes to fool the honest nodes into selecting different leaders; all honest nodes will agree on the same node as the leader. But as the honest nodes do not know which nodes are byzantine, they can select a byzantine node as the leader. This can happen when the token ends up at a byzantine node in the last round. To maximise the probability of a byzantine node getting selected in the last round, the honest nodes should be selected as leaders in the initial rounds. So the byzantine nodes can use the following strategy to maximise the chances that a byzantine node is chosen as the final leader:
Whenever a byzantine node is selected as a leader in an intermediate round, the byzantine nodes will select an honest node instead of randomly selecting a node to pass the token to.

Part 3. Analysis – Via Experimentation

For a given value of n and b, the simulation was run 10000 times. The number of times the leader decided was an honest node or a byzantine node is noted. The following graph is obtained from the outcomes of the simulations:



The above graph was plotted assuming that the byzantine nodes always follow the optimal strategy of passing the token to an honest node.

Following is the code for the simulation:

```
import random
from collections import defaultdict

n = 100
NUM_ITER = 10000

result = {}
for b in range(n+1):
    result[b] = [0, 0]
for _ in range(NUM_ITER):
    for b in range(n + 1):
        byz = set(random.sample(range(1, n + 1), b))
        valid_next_leader = set([i for i in range(1, n + 1)])
        valid_honest_next_leader = valid_next_leader - byz

        curr_leader = 1
        valid_next_leader.discard(curr_leader)
        valid_honest_next_leader.discard(curr_leader)
```

```
        for r in range(2, n + 1):
            if curr_leader in byz and valid_honest_next_leader:
                next_leader = random.choice(list(valid_honest_next_leader))
                valid_honest_next_leader.discard(next_leader)
                valid_next_leader.discard(next_leader)
            else:
                next_leader = random.choice(list(valid_next_leader))
                valid_next_leader.discard(next_leader)
                valid_honest_next_leader.discard(next_leader)
            curr_leader = next_leader
        final_leader = curr_leader
        if final_leader in byz:
            result[b][1] += 1
        else:
            result[b][0] += 1
        assert not valid_honest_next_leader
        assert not valid_next_leader
```

Part 4. Analysis – Conjecture

Let $f(n, b, state)$ be the probability that the leader elected at the end of the protocol is honest, where n is the total number of rounds remaining, b is the number of byzantine nodes yet to be a round leader, and state is the type(0 - honest, 1 - byzantine) of the current round leader. Assuming that the byzantine nodes always follow the optimal strategy, we can define the following recursive relationship:

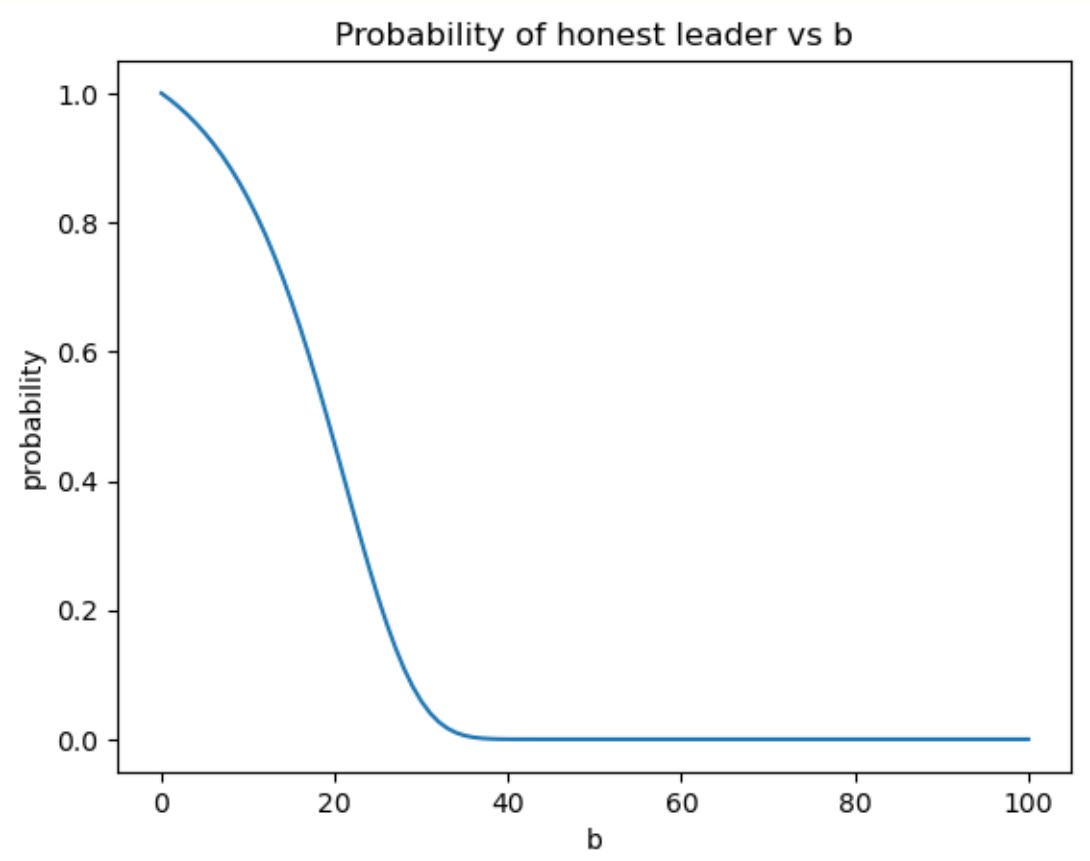
if $state == 0$ **then** ; `// i.e., current round leader is honest`

return $\frac{b}{n} * f(n-1, b-1, 1) + \frac{n-b}{n} * f(n-1, b, 0)$; `// good nodes choose either a honest node or byzantine node as the next round leader.`

else

return $f(n-1, b, 0)$; `// byzantine nodes choose only a honest node as the next round leader.`

The base cases are when n = 0. If n = 0 and state = 0, return 1 else return 0. Solving this function recursively, the following graph is obtained for n = 100.



Following is the code to solve the function recursively:

```
from functools import lru_cache
@lru_cache()
def f(n, b, state):
    if n == 0:
        if state == 0:
            return 1
        else:
            return 0

    if state == 0:
        curr = 0
        if b > 0:
            curr += (b/n)*f(n-1, b-1, 1)
        if n - b > 0:
            curr += ((n-b)/n)*f(n-1, b, 0)
        return curr
    else:
        if n - b > 0:
            return f(n-1, b, 0)
        else:
```

```
        return 0
```

Problem 3. Incorrect Implementations of the Dolev-Strong Protocol

In this problem, you are asked to help analyze two buggy implementations of the Dolev-Strong protocol. You must provide specific examples (and attack strategies) illustrating the mistake. Your example in each case must be small and the attack strategy must be as simple as possible. (This problem is taken from the book by Elaine Shi – Exercise 6 on Page 19).

Part 1. Lanie's Mistake

Suppose the sender is honest and receives the input bit 1. The sender sends $< 1 >_1$ to all nodes in the 0^{th} round. At the start of the first round, all the nodes will get the message sent by the sender, and all the honest nodes will add 1 to their extracted set. Now, the byzantine nodes know the bit the sender got as input. The byzantine nodes can send the message $< 0 >_{b1,b2}$ at the end of the first round. The message has the bit as 0 and two signatures of byzantine nodes, b1 and b2. The honest nodes will get this message at the start of the second round. They will see that there are 2 signatures associated with the message, and they will add 0 to their extracted set. At the end of all the rounds, the honest nodes will output the default value as they have both 0 and 1 in their extracted sets. This breaks validity as the sender is honest, and all the honest nodes should have 1 as their output, which was the sender's input bit. Thus, we can understand the importance of checking the sender's signature in the r-length signature chain at the start of the r^{th} round.

Part 2. Elanna's Mistake

Consider the following case; there are f byzantine nodes, out of which one node is the sender. Divide the honest nodes into two groups, H1 and H2. In round 0, the corrupt sender sends $< 1 >_1$ to all honest nodes. Thus, all honest nodes will add 1 to their extracted sets in round 1. Assume all the byzantine nodes behave like honest nodes till the last round. In the f^{th} round, the f byzantine nodes send bit 0 to group H2 with f + 1 signatures. As the attackers can forge signatures, the f byzantine nodes can forge the signature of a single honest node and add it to the message along with their f signatures. In the $(f + 1)^{th}$ round, after receiving the message with bit 0 from the byzantine node, the honest nodes in group H2 will add 0 to their extracted set, whereas the honest nodes in group H1 will have a single bit 1 in their extracted set. Thus, the honest nodes in the group will output 1, and the honest nodes in group 2 will output the default value. This breaks the consistency

property.

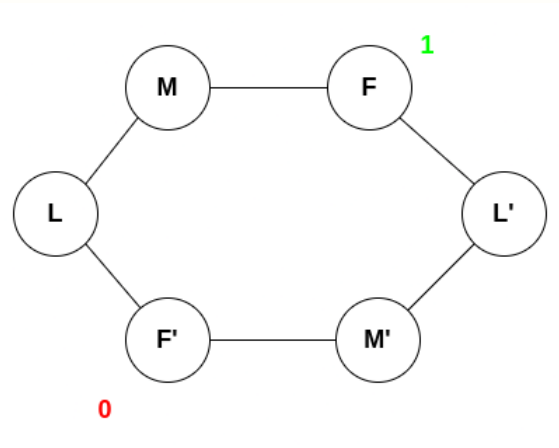
Problem 4. Randomized Lower Bound

Part 1. Reading Assignment

Provide your answer in this space.

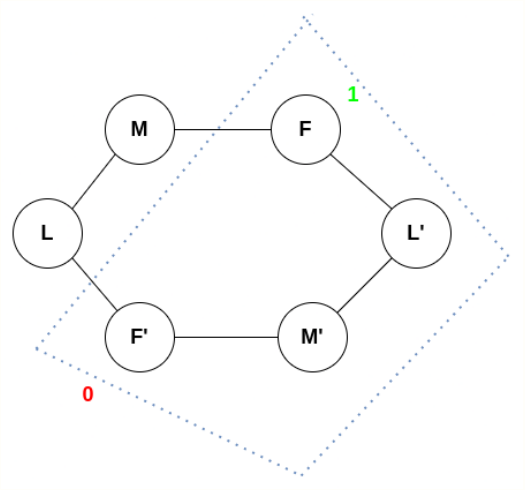
Part 2. Randomized Lower Bound

Assume the probability of error to be δ. We want to prove that it is impossible to achieve BB with $\delta \leq \frac{1}{3}$ when $f \geq \frac{n}{3}$. Validity with δ error is defined as follows: If an honest sender has an input bit b, with probability at least $1 - \delta$, all the honest nodes must output b at the end of the protocol. Agreement is defined as follows: with probability at least $1 - \delta$, all the honest nodes output the same bit at the end of the protocol.
Six nodes F, L, M, F', L' and M' are connected in a cyclic manner.



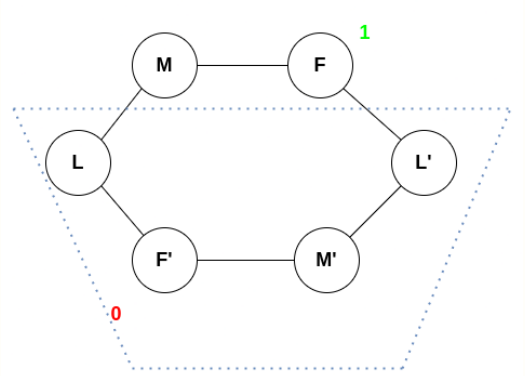
Node F has an input of 1, and node F' has an input of 0. We will assume that there exists a protocol P, that satisfies consistency and validity when $f \geq \frac{n}{3}$ with a probability of error less than $\frac{1}{3}$. We will consider 3 different interpretations of our model.

- *Interpretation 1*: Consider nodes L and M are honest, and node F is corrupt. Node F simulates the nodes F, F', L' and M'.



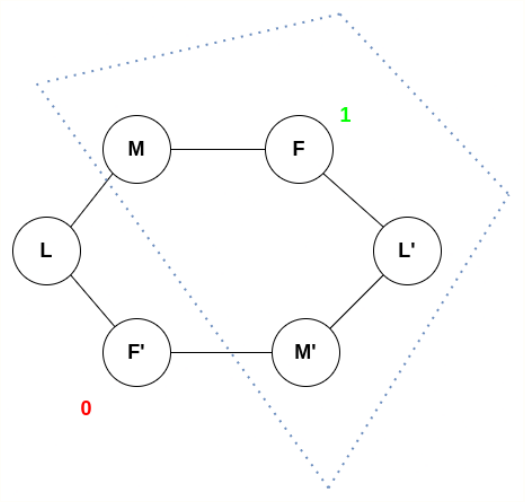
Using this interpretation, we conclude that nodes L and M must output the same bit with probability $\geq \frac{2}{3}$.

- *Interpretation 2*: Node L is corrupt, simulating all the nodes L, L', F' and M', whereas nodes F and M are honest.



Using this interpretation, by validity, we conclude that M should output F's input bit, that is 0 with probability $\geq \frac{2}{3}$.

- *Interpretation 3*: Node M is corrupt, simulating all the nodes M, M', F, and L', whereas F' and L are honest.



Using this interpretation, by validity, we conclude that L should output the input bit of F', that is, 1 with probability $\geq \frac{2}{3}$.

Thus from the results of the 3 interpretations, we reach a contradiction about the probability of what L and M will output. Hence we have successfully proved that a protocol does not exist that achieves BB with a probability of error $\delta \leq \frac{1}{3}$.

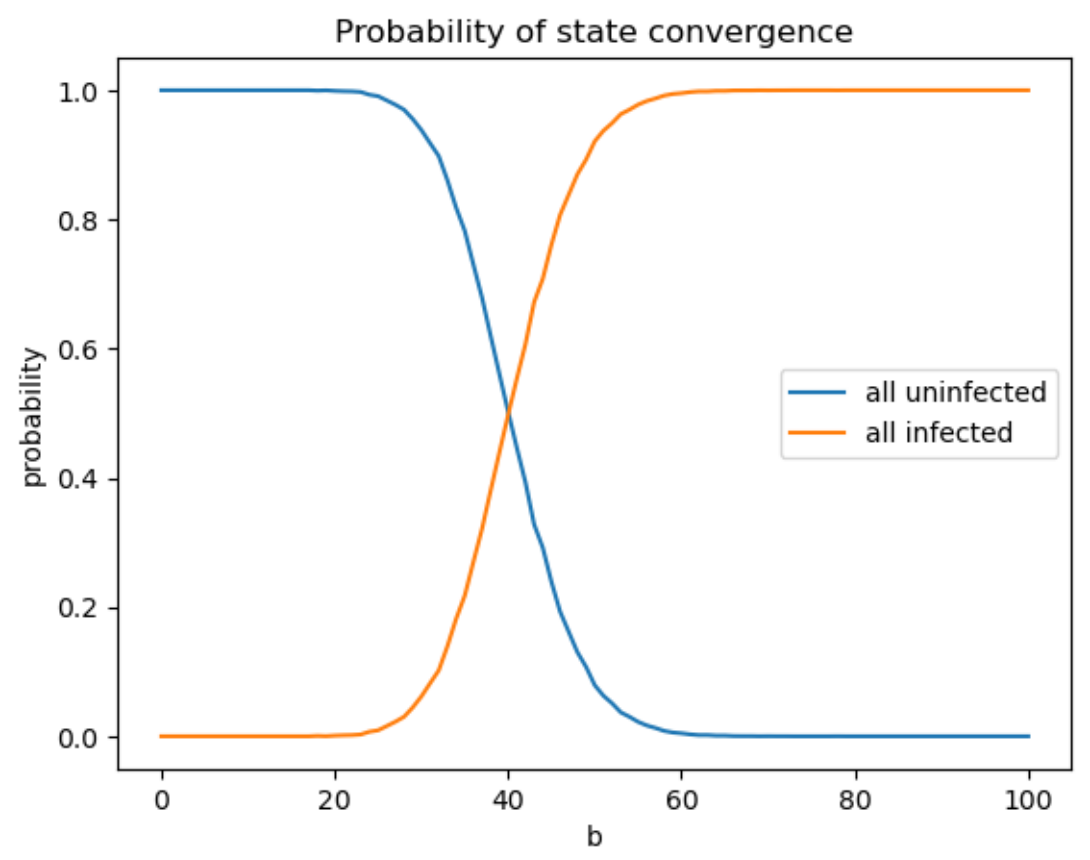
Problem 5. Going Viral

Part 1. Experimental Setup

The code for this experiment is written in Python. The total number of nodes, n = 100 and b varies from 0 to n. For each value of b, the experiment was run 10000 times. In each iteration, b random nodes are chosen from the n nodes. In each round, the infected nodes select one random node and infect it if that node is not infected already. Also, each uninfected node selects one random node and heals it if it is infected. The simulation is stopped once all nodes are uninfected(state 0) or infected(state 1). The number of rounds required to converge and the final result for each b are noted.

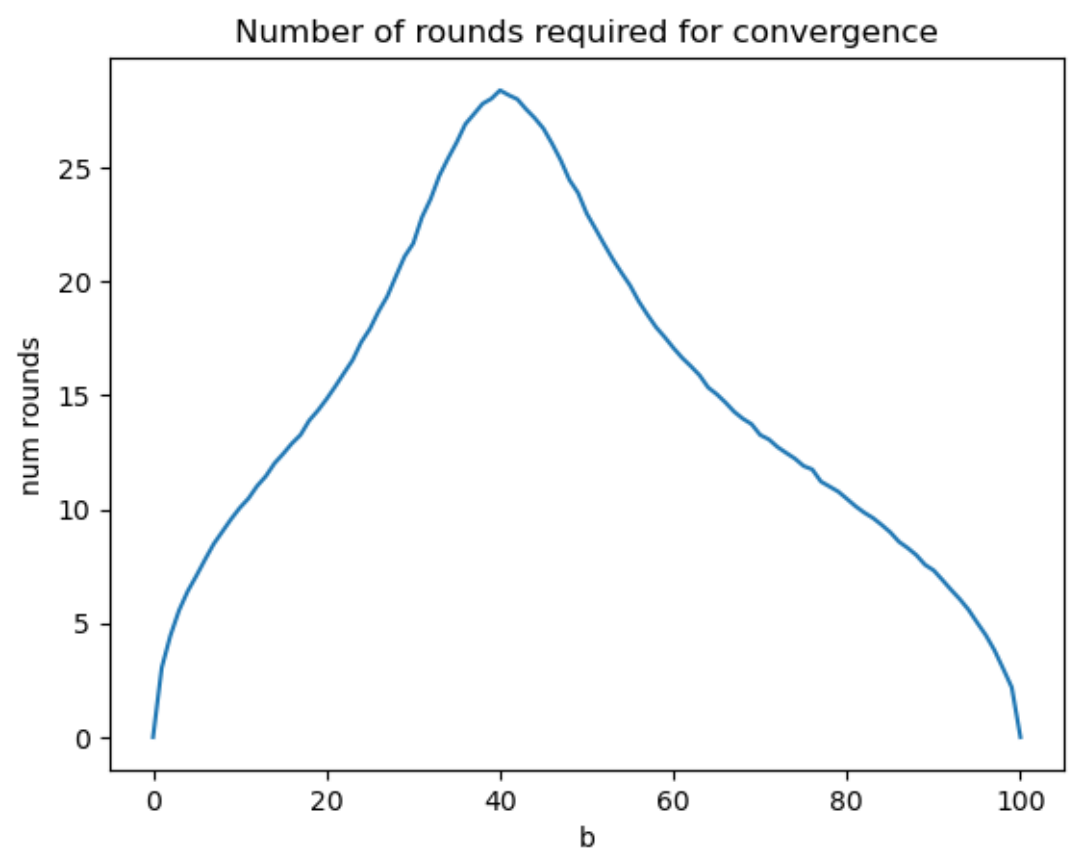
Part 2. Randomized Lower Bound

The following plot shows the fraction of times the nodes converge to state 0(all uninfected) and state 1(all infected) for all values of b.



The simulation was run for 10000 times for a given value of n, b. The fraction of times the nodes converge to a particular final state is assumed to be the probability

of convergence to that state. We can see till $b < \frac{n}{3}$ the probability of convergence to state 0 is 1. For $\frac{n}{3} < b < \frac{2n}{3}$ the outcome of convergence is not very predictable. The probability of convergence to state 0 decreases and the probability of convergence to state 1 increases. This behaviours is expected because as the initial number of infected nodes increases, the rate of new nodes getting infected will increase as compared to the rate of infected nodes getting healed. For $b > \frac{2n}{3}$ the probability of convergence to state 1 is 0.
The following plot shows the average number of rounds taken to converge for each value of b.



We can see that as the initial absolute difference between infected and uninfected nodes increases, it takes more rounds for the nodes to converge to a final state.

Following is the code for simulation:

```
import random
from collections import defaultdict

n = 100
NUM_ITER = 100000

result = {}
for b in range(n+1):
```

```
    result[b] = [0, 0]
num_rounds_dict = defaultdict(int)

for _ in range(NUM_ITER):
    for b in range(n + 1):
        infected = set(random.sample(range(0, n), b))
        uninfected = set([i for i in range(n)]) - infected
        num_rounds = 0
        while len(infected) != 0 and len(infected) != n:
            new_infected = infected.copy()
            for v in infected:
                u = random.randint(0, n-1)
                if u in uninfected:
                    uninfected.remove(u)
                new_infected.add(u)

            infected = new_infected
            new_uninfected = uninfected.copy()
            for g in uninfected:
                w = random.randint(0, n-1)
                if w in infected:
                    infected.remove(w)
                    new_uninfected.add(w)
            uninfected = new_uninfected
            num_rounds += 1


        num_rounds_dict[b] += num_rounds
        if len(infected) == 0:
                # print(f"Final result: ALL UNINFECTED :)")
            result[b][0] += 1
        else:
                # print(f"Final result: ALL INFECTED :(")
            result[b][1] += 1
```

Acknowledgments

I would like to thank the following people with whom I discussed ideas.

- Pranav Phatak - for discussing why my approach was wrong in the original Q1, which ended up being the incorrect protocol in the updated Q1.
- Rajdeep Paul