Assignment 1: CS765

190050113 Shivam Raj
190050017 Aniket Agrawal
190020010 Aman Singh

September 7, 2020

Selfish Mining

Depicted below is an instance of selfish mining. The green dots represent attacker blocks in the blockchain, the red blocks show the blocks which are a part of the longest chain and the blue dots represent all other blocks that were generated. Note that the blocks in the longest chain which are mined by the attacker are colored green.
A long fork at the top of the figure below depicts a situation where the attacker was able to generate a lot of private blocks. When the leads comes down to 2 it release blocks from its private chain and kills the 'one block smaller' public chain (the long chain of blue blocks depicts the killed pulic chain).

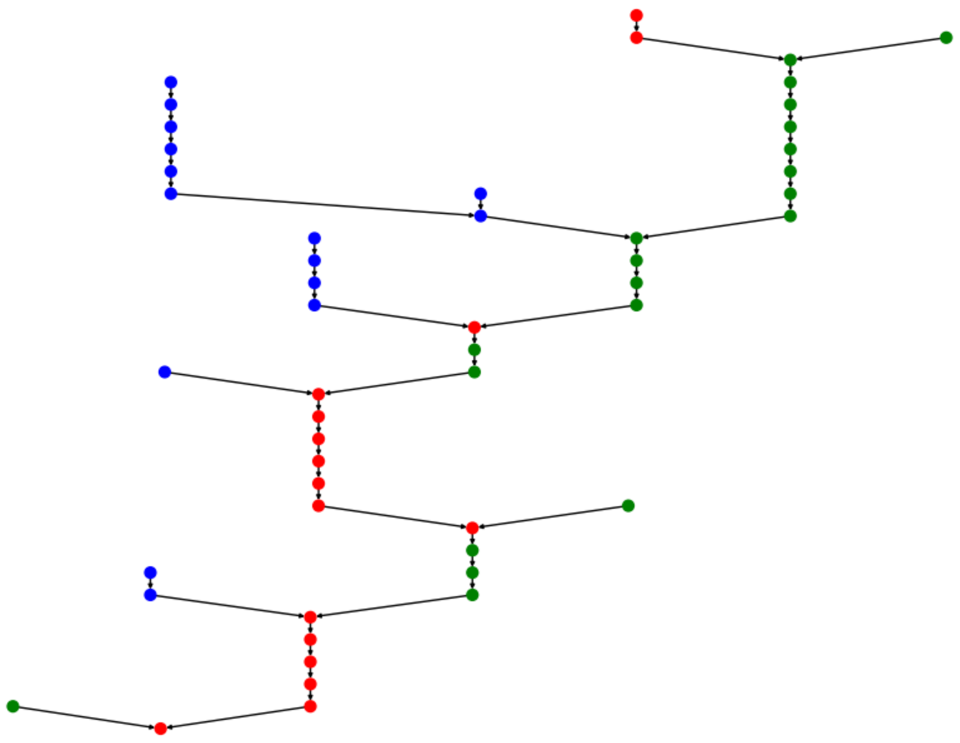


Figure 1: Blockchain of a random node in network after simulation

Depicted below is a failed version of stubborn mining attack where the attacker isn't able to get his blocks into the main chain even when he gets a lead of 3 as it does not deplog the policy of releasing 2 front blocks when the lead is 2.

Stubborn Mining

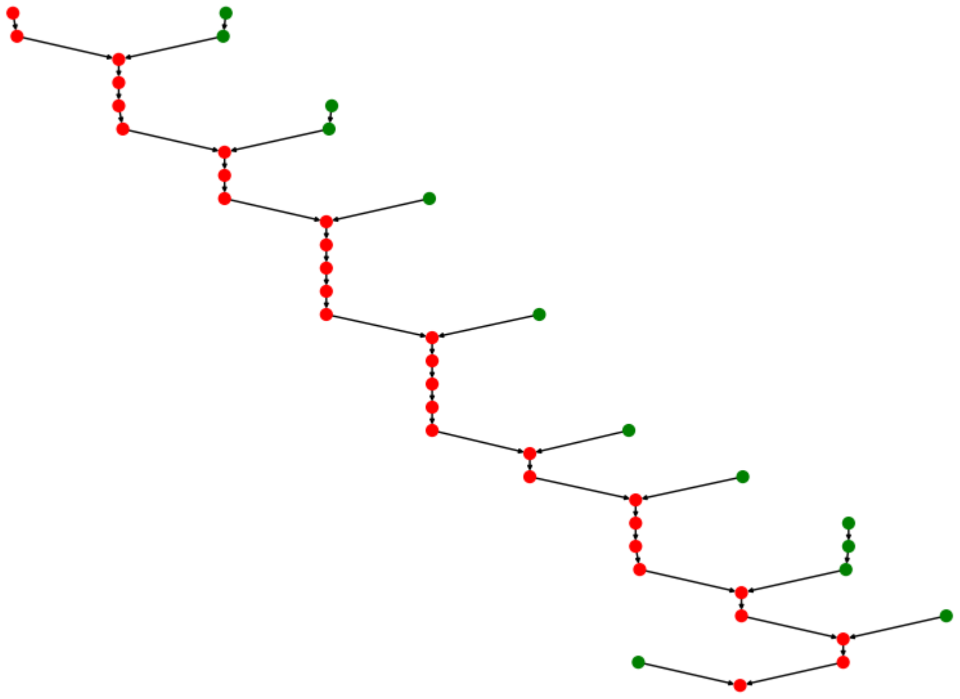


Figure 2: Failed stubborn mining

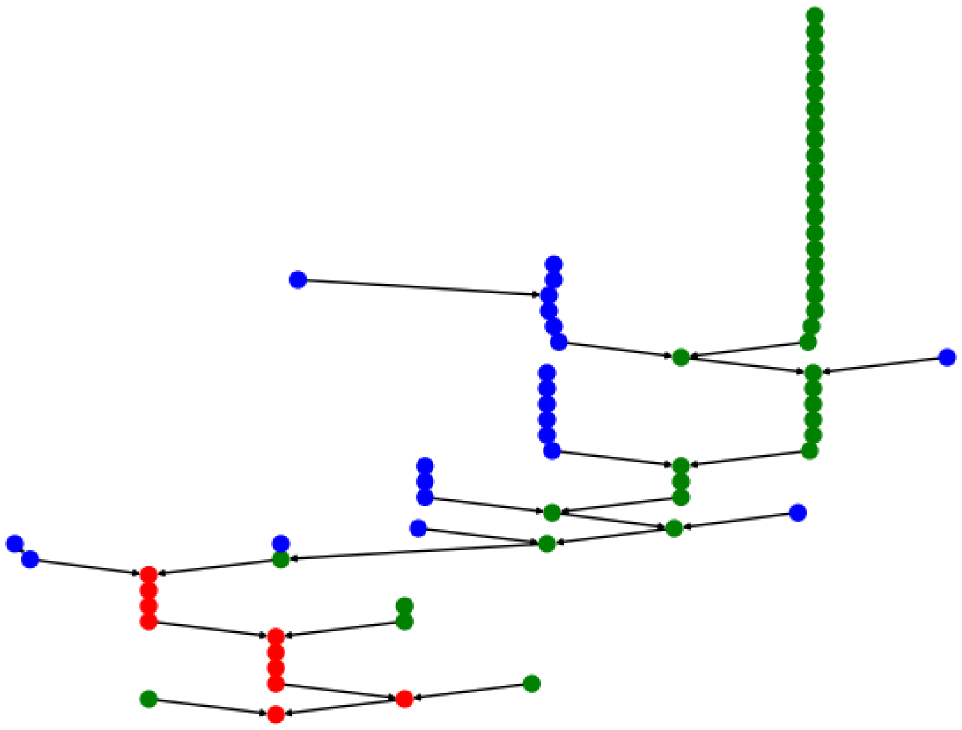


Figure 3: Succeded stubborn mining

Stats for different parameters of Selfish Mining

1. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 25%
   $MPU_{node_{adv}}$ = 0.5263157894736842
   $MPU_{node_{overall}}$ = 0.7073170731707317
   Fraction of attaker blocks in main chain = 0.3448275862068966

2. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 50%
   $MPU_{node_{adv}}$ = 0.3125
   $MPU_{node_{overall}}$ = 0.5945945945945946
   Fraction of attaker blocks in main chain = 0.22727272727272727

3. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 75%
   $MPU_{node_{adv}}$ = 0.45
   $MPU_{node_{overall}}$ = 0.6046511627906976
   Fraction of attaker blocks in main chain = 0.34615384615384615

Stats for different parameters of Stubborn Mining

1. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 25%
   $MPU_{node_{adv}}$ = 0.0
   $MPU_{node_{overall}}$ = 0.6904761904761905
   Fraction of attaker blocks in main chain = 0.6956521739130435

2. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 50%
   $MPU_{node_{adv}}$ = 0.3125
   $MPU_{node_{overall}}$ = 0.56
   Fraction of attaker blocks in main chain = 0.5714285714285714

3. Number of nodes =100, Mining power of adversary node=0.34, Connectivity of adversary node = 75%
   $MPU_{node_{adv}}$ = 0.8780487804878049
   $MPU_{node_{overall}}$ = 0.6338028169014085
   Fraction of attaker blocks in main chain = 0.8

As we see above the selfish miner is always able to get blocks into the public chain when it gets a lead greater than or equal to 2, whereas, it isnt the case with stubborn miner. But when the stubborn miner is able to win the crypto-puzzle at state 0', it gets more blocks into the chain than the selfish miner.
Although in both the cases, we see that with suitable mining power and fraction of directly connected peers, we are able to get more blocks into the main public chain with the attack than what we could is we were honest.

Design Doc

There are 5 important classes in our solution : `Transaction`, `Block`, `Event`, `Node` and `Simulation`. The `Transaction` and `Block` classes model their standard meanings. `Transaction` has a sender, a receiver, a value and a unique id (tid). `Block` has a block id, a parent block id, transactions included in the block and a miner.

```
class Transaction:
    def __init__(self, tid, sender, receiver, value):
        self.sender=sender
        self.receiver=receiver
        self.value=value
        self.tid=tid
class Block:
    def __init__(self, bid, pbid, txnIncluded, miner):
        self.bid = bid # block id
        self.pbid = pbid # parent block id
        self.txnIncluded = txnIncluded.copy()
        self.miner = miner
class Blockchain:
    def __init__(self, gblock):
        self.blocks = {gblock.bid : (gblock, 0)}
        self.head = gblock
        self.g = nx.DiGraph()
```

The `Event` class is inherited by 4 other classes `TxnGen`, `TxnRecv`, `BlockRecv` and `BlockMined`. The event class has attributes such as a sender, receiver, time and eventid. The 4 subclasses of `Event` class form all the events that we need to add to our Simulation object's event queue.

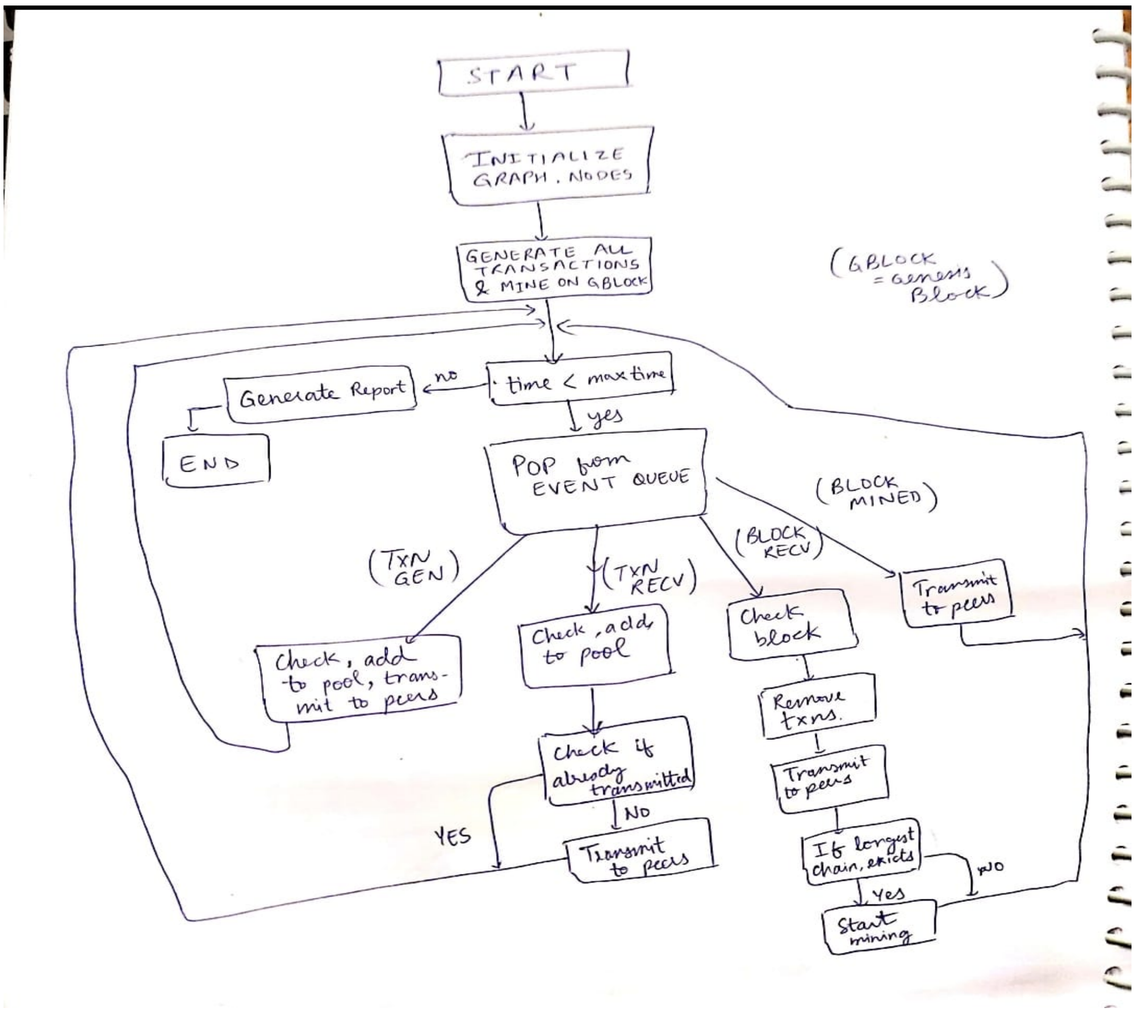
```
class Event:
    def __init__(self, time,eventId, sender=-1, receiver=-1):
        self.time=time
        self.sender=sender
        self.receiver=receiver
        self.eventId=eventId
```

The `Node` class has all the relevant fields like the mean of generating time required to mine a block, it's id, whether or not it is "fast" etc. This class also includes all the event handler functions. Whenever an event (object of the `Event` class) is popped from the event queue, we inspect the details of of the event and call the relevant node's event handler. The event handlers then add other events to the event queue at appropriate times. The time attribute of the events which are added are calculated using appropriate latencies (block size, link speed, speed of light delay, time required to mine etc.).

```
class Node:
    def __init__(self, nid, speed, genesis, miningTime):
        self.nid = nid # unique id of all thr nodes
        self.speed = speed # 1=fast, 0=slow
        self.blockchain = Blockchain(genesis)
        self.miningTime = miningTime # represent the mining power of node, mean mining time of no
        self.peer = set() # neighbours of the node
        self.txnReceived = set() # txn received till now
        self.blockReceived = set() # blocks received till now
```

The `Simulation` class maintains the simulation parameters, initialzes nodes and connects the nodes appropriately. It also maintains an event priority queue from which it pops events and calls relevant event handlers passing in the appropriate parameters.

Changes :
There are 2 adversary classes `AdversaryNode` (represents selfish miner) and `StubbornNode` (represents stubborn miner) defined in similarly named files. These have custom block receive and self-block receive function which have a corresponding implementation of the attacker behaviour. The type of adversary and other constants such as fraction of peers it it connected to is defined in `params.py`


START
INITIALIZE GRAPH, NODES
GENERATE ALL TRANSACTIONS & MINE ON GBLOCK
(GBLOCK = Genesis Block)
time < max time
no
Generate Report
END
yes
Pop from EVENT QUEUE
(TXN GEN)
(TXN RECV)
(BLOCK RECV)
(BLOCK MINED)
Check, add to pool, transmit to peers
Check, add to pool
Check if already transmitted
YES
No
Transmit to peers
Check block
Remove txns.
Transmit to peers
If longest chain, exists
Yes
NO
Start mining
Transmit to peers