HW2 – Blockchain

TA in charge – Sagitt Efrat

We have seen in class that the fee required to get a transaction into a block in the chain is highly uncertain, especially when there are many pending transactions. The range of used fees is very large, between 1 and 2000 Satoshi per byte (Satoshi is the smallest currency unit in Bitcoin, and there are 100,000,000 Satoshi in 1 BTC).

In this exercise you will experiment with the Bitcoin mechanism from 2 sides of the users: first as a miner that will implement an **alternative** pricing mechanism. Then, as a user that is facing **the current** pay-your-bid mechanism used by the miners, trying to get your transactions into the blockchain as fast as possible at a minimum cost.

The data you are using was collected by an active node in the Bitcoin network during the months August-November 2017. The node "listened" to the transactions flowing through the network and collected for each transaction the following:

Name	Type	Description
`TXID`	string	The TXID of a transaction in the memory pool, encoded as hex in RPC byte order. Unique for each TX.
`size`	number (int)	The size of the serialized transaction in bytes
`fee`	number (float)	The transaction fee paid by the transaction in Satoshi (1e-8 Bitcoin)
`time`	number (int)	The time the transaction entered the memory pool, Unix epoch time format
`depends`	array	An array holding TXIDs of unconfirmed transactions this transaction depends upon (parent transactions). Those transactions must be part of a block before this transaction can be added to a block, although all transactions may be included in the same block. The array may be empty
`output`	number (int)	The amount of Satoshis (1e-8 Bitcoin) spent to this **output. May be 0**
`removed`	number (int)	The block header time (Unix epoch time) of the block which includes this transaction. Only returned for confirmed transactions.

You will receive a json file with the information about the transactions waiting in the mempool, each object looks like the following:

```
"155d22d2553eed00170233939238b0ee5098d5b1ef91d19420ef96899b677bd3": {"output": 11743613, "removed": 1504472141, "fee": 16046.0, "time": 1504472139, "size": 225}
```

Where the transaction id is(unique):
"155d22d2553eed00170233939238b0ee5098d5b1ef91d19420ef96899b677bd3"

*Note that when output= -1 the tx (transaction) was not confirmed (still it waits in the mempool and should be considered)

Mandatory part:

Question 1

a. You are a miner and you have just mined a new block ("solved the riddle"). Implement a greedy knapsack algorithm that fills the block with the most profitable pending transactions.

   Denote B' = greedy_knapsack(block_size, mempool_data):

   In the file 'bitcoin_mempool.json' you can see the current state of the mempool. You will be tested on similar file with similar number of pending transactions.

   First: complete the function $load\_mempool\_data((block\_size, mempool\_data\_name, current\_time)$- the function gets the path to mempool_data_name file and loads the the pending transactions into an object called "mempool_data". A transaction T is **pending** if: time(T) < current_time < removed(T)

   Then, complete the function $greedy\_kanpsack(block\_size, mempool\_data)$, that receives the mempool_data object and the block_size and returns a list of the tx id's you want to include in the block:

   - The greedy knapsack algorithm sorts all pending transactions according to satoshi-per-byte. Then it adds transactions to the block in decreasing order, until no more transactions can be added (the total size of added transactions may not exceed the size of the block). If two tx has the same fee-per-byte choose first the one with lower id (compare strings).
   - Each of the functions above should finish running in 2 minutes (for current_time= 1510264253.0). Fill the functions in the file hw2_part1.py attached in moodle.

b. Calculate revenue in pay-your-bid mechanism: complete the function $evaluate\_block(tx\_to\_insert, mempool\_data)$ that receives the object of mempool_data and the list of tx id's you decided to insert into the block in section a, and returns the miner overall revenue from including the transactions (in Satoshi). Fill the functions in the file hw2_part1.py attached in moodle.

c. Implement VCG pricing[1] for any tx_i in B', using the fee as bid:

   - compute $p_i = V^S_{B-tx_i} - V^{S-j}_{B-tx_i}$
   - $V^S_{B-tx_i}$ is the total fees that the greedy algorithm can collect if transaction tx_i does not exist.

[1] Since we use the greedy algorithm, which only provides an approximation of the largest sum of bids, we only get *approximate VCG prices*. The theorems we saw in class may not apply to these prices, but they will be good enough for our needs in this exercise.

- $V_{B-tx_i}^{S-j}$ is the total fees that the greedy algorithm currently collects from all transactions that are not tx_i (j is the space in the block used by tx_i)

Where B contains all pending transactions, S is the block size, and $B' \subseteq B$ is the set of transactions that the greedy algorithm selected to put in the block.

Complete the function $VCG(block\_size, tx\_to\_insert\_list, mempool\_data)$, that receives the block_size, tx_to_insert_list - the list of id's that you have found in part a and the object of the mempool_data. The function should return the overall revenue of the miner for his chosen transactions from part a in the case that each of them pays its VCG price. Fill the functions in the file hw2_part1.py attached in moodle. For 1,000 pending tx in the mempool this function should finish running in 5 minutes.

Compare the revenue under pay-your-bid versus the VSG revenue. What is the difference in the revenue? Think what is the benefit of implementing a VCG mechanism?

Question 2

a. Implement a simple bidding agent that gets the utility function, which depends on 5 parameters: $(v, r, size, z, t)$. The first 3 parameters will be given as input:
   - v is the maximal value (in Satoshi). This value is not related to the field output(T). It reflects how important the transaction is.
   - $r \in [0,1]$ is an urgency parameter.
   - $size$ is in bytes.

   The next two parameters depend on the actions of the bidding agent:

   - z is the fee (in satoshi-per-byte, limited to nonnegative multiples of 10)
   - t is the time until the transaction is added to some block in the chain, which may depend on the fee and the other transactions in the mempool.

   The utility from a transaction is that is inserted to some block is:

$$\boldsymbol{Utility(v, r, size, z, t) = v * 2^{-t * \frac{r}{1000}} - z * size}$$

   If the transaction is not inserted to any block (t=never) then

$$\boldsymbol{Utility(v, r, size, z, t) = 0}.$$

   Note that as the fee is higher, the transaction will typically be inserted sooner (t will be lower). A **bidding agent** accepts as input the parameters $(v, r, size)$ and data about past and present transactions in the mempool, and needs to output a fee z. In this exercise you will be given one bidding agent as an example, and will be required to implement two other bidding agents.

- The “truthful” bidding agent calculates the value of the transaction in 1 hour and uses this as the bid. Thus $z = v * 2^{-3.6*r)}$. This code is implemented at the file hw2_part1.py attached in moodle.

- The “forward” bidding agent looks only at the current mempool. It tries to “guess” the time $GT(z)$ it will take the transaction to be added for any fee z, and the guess utility earned or lost. Then it selects the fee z that maximizes the “guessed utility”

$\boldsymbol{GU(z) = Utility(v, r, size, z, GT(z))}$**.** It assumes no new transactions are added to the mempool after the current time. The insertion of the tx to a block goes as follows:

  - Every 10 minutes exactly, a new block is created, and clears transactions using the greedy pay-your-bid algorithm from part 1a ($greedy\_knapsack()$).
  - The agent calculates for every z, the time $GT(z)$ (in seconds) until the first transaction with fee at most z (per byte) is cleared.
  - The different z values are multiples of 10 between 0 and 5000 (0,10,20,30,…4990,5000).

  Here is an example for transaction (v=25000, r=0.5, size = 10) (The times $GT(z)$ are made up and not based on the real data).

z	$GT(z)$	$GU(z)$
0	Never	0
10	Never	0
20	Never	0
30	66000 sec	25000*2^(-33) - 300 ~= -299
40	48000 sec	25000*2^(-24) - 400 ~= -399
…		
1000	1800 sec	25000*2^(-0.9) – 10000 ~= 3397
1010	1200 sec	25000*2^(-0.6) – 10100 ~= 5583
1020	1200 sec	25000*2^(-0.6) – 10200 ~= 4673
…		
4990	600 sec	25000*2^(-0.3) – 49900 ~= -79593
5000	600 sec	25000*2^(-0.3) – 50000 ~= -79693

  - The agent bids the z that maximizes the guessed utility $\boldsymbol{GU(z)}$ (marked in yellow)

  You should implement the $forward\_bidding\_agent(tx\_size, value, urgency, mempool\_data, block\_size, time\_added)$ that returns the bid z (in Satoshi-per-byte, nonnegative multiple of 10).

For questions 1-2 you will load the data once, so for both questions you can use the same object mempool_data you have loaded in part 1a.

Competitive part:

- Implement any bidding agent. We will run the simulation with the greedy pay-your-bid algorithm, with transactions added from the data:

- In every simulation, the agent receives data until time t0 and a transaction $X = (v, r, size)$, and sets a fee z.
- To evaluate the expected utility, we check how much time it takes until X is added to the chain, evaluated by data:
    - Consider the last $k = 10$ transactions $(Y_1, \dots, Y_k)$ with fee-per-byte similar to z - by l_2 norm.
    - For each Y_i, consider the times $t0_i$ when Y_i was added to the mempool, and $t1_i$ when Yi was added to a block. $Let\ t_i = t1_i - t0_i$
    - For each Yi, let $w_i = v * 2^{-ti*r/1000}$
    - The agent gains $\frac{1}{k}\sum_{i=1}^{k}(w_i - z * size)$

For this part you will receive a csv file named hw2_part2.csv with the tx to complete their fees. Each row represents a transaction. The data you will be tested on a similar distribution as the data you were given – similar fee values, distribution over the time of day, number of total transactions to compare, etc...

*time_of_day is a float between (0,24), for example 3.18333 stands for 11 minutes after 3am.

<u>Submission instructions:</u>

# Make no use in the local path in your computer for reading files. For example if you need to upload the data call it via 'os.path.abspath('bitcoin_mempool_data.json')', not via: '/home/sagit/Desktop/ci/data/Bitcoin_mempool_data.json'

# Do not use main inside hw2_part1.py, you can change the main you were given but please note that we will not use the same one so your changes will not be submitted. Do not change/override the constants we call in the functions, as they will be redefined for different dataset.

# Attach one zip file (not rar or tar or gz or you got it...) that carries the name hw1_studentID1_studentID2.zip for only one submitter than the name should be hw1_studentID1.zip (fill in your own ID's).

The zip file should include:

1. hw2_part1.py

2. hw2_part2.csv

3. Bitcoin_mempool_data.json

# make sure there are no sub folders in the zip file. Make sure your file does not have the name hw1_studentID1_studentID2.zip.zip (funny but happens a lot).

# Only one student in the pair should submit