Practical Exercise 2

TA in charge -Oded Salton

All the data and assumptions in this exercise are for educational purposes only.

Due date:08/06/20

In this exercise you will practice some of the network concepts learned in class.

Background story :

this exercise is involving the blockchain technology, in this technology every time X the system is producing a block.

The miner is interested in selling the room in the block for the highest fee.

Each user has one or more transactions, and wants to buy room in the block for these transactions (paying low fees if possible).

In this exercise, all mined blocks enter the chain and stay in it forever (no block is wasted or disposed).

The fee is paid in units called **Satoshi** (Satoshi is the smallest currency unit in Bitcoin , and there are 100,000,000 Satoshi in 1 BTC). The range of used fees is very large and it can be between 1 and 2000 **Satoshi per byte**.

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 **mempool** is the set of pending transactions.

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

`output`	number (int)	The amount of Satoshi (1e-8 Bitcoin) spent to this **output. May be 0**
`removed`	number1 (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 following:
"155d22d2553eed00170233939238b0ee5098d5b1ef91d19420ef96899b677bd3": {"output": 11743613, "removed": 1504473541, "fee": 16046.0, "time": 1504472139.99, "size": 225}

Meaning: this TX entered the mempool on time 1504472139. It entered a block 1402 seconds later, on time 1504472139.99. The size of the TX is 225 bytes, and the total fee offered to the miner was 16046 Satoshi (about 75 Satoshi/byte). The TX transferred 11743613 Satoshi (about 0.11 bitcoin) from some wallet to another (we do not know the source and target).

Python packages that you are allowed to use (all in their python 3.6 version) : networkx, numpy , pandas , matplotlib , scipy

*if when you run the program in the VM and it doesn't find the packages use the command /usr/bin/python3.6 <filename>.py

Part A (45 points) :

You are a miner and you are working on mining a new block ("solving the riddle").

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

1. Implement the function filter_mempool_data(mempool_data,current_time)

This function returns all pending transactions from the mempool. A transaction TX is pending if: time(TX) < current_time < removed(TX)

The function returns all transactions in a dataframe. The column titles must be identical to the column titles in the csv file.

2. Implement the function greedy_knapsack(block_size, all_pending_transactions)

This function returns tx_list, which is a set of TXID of transactions that fit 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 TXs has the same fee-per-byte, choose first the one with lower id (compare strings).

3. Calculate revenue in pay-your-bid mechanism:
complete the function : evaluate_block(tx_list, all_pending_transactions)

the function returns the miner's overall revenue from including these transactions (in Satoshi).

4. Implement VCG_prices(block_size, all_pending_transactions, tx_list)

The function returns a dictionary of prices where the key is TXID

for any tx_i in tx_list, using the fee as bid :

- compute $p_i = V^S_{B-tx_i} - V^{S-size(tx_i)}_{B-tx_i}$
- $V^S_{B-tx_i}$ is the total fees that the greedy algorithm can collect if transaction tx_i does not exist.
- $V^{S-size(tx_i)}_{B-tx_i}$ is the total fees that the greedy algorithm currently collects from all transaction that are not tx_i

compare the revenue (sum of fees) under pay-your-bid versus VCG revenue. what is the difference in the revenue? think: what is the benefit of implementing a VCG mechanism ?

Comments:

* 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

5. (**bonus** up to 5 points): implement the function blocks _after_time_1510266000()

The function should return a list of 10 integers, where each entry is the Unix epoch time of a real block that is about to create after time 1510266000 .
you can assist the function:
blocks_by _time_1510266000():
which returns the integers of the real block created between time 1510261377 and time 1510266000

- 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.

Part B (45 points):

In this part you implement an agent that is bidding on behalf of the user. Your goal is to get your transactions into the blockchain as fast as possible, while paying the minimal possible fee to the miners.

You will have access to two files. One is the current state of the mempool as in part A.

use the function
filter_mempool_data(mempool_data,current_time)
This function returns all pending transactions from the mempool. A transaction TX is pending if: time(TX) < current_time < removed(TX)

The function returns all transactions in a dataframe. The column titles must be identical to the column titles in the csv file.

The other file contains a list of transactions that you want to insert to the blockchain. It has the following format:

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
time	number (int)	The time the transaction entered the memory pool, Unix epoch time format
min_value	number (int)	The minimal value of this transaction to the user (in satoshi).
max_value	number (int)	The maximal value of this transaction to the user (in satoshi).

The value of a transaction depends on the time that passes until the transaction gets into the block, and it goes down from max_value to min_value as time passes. This value reflects how important it is for the user (you) to get this transaction into the blockchain. (think: how is this value related, if at all, to the amount of bitcoin transferred?)

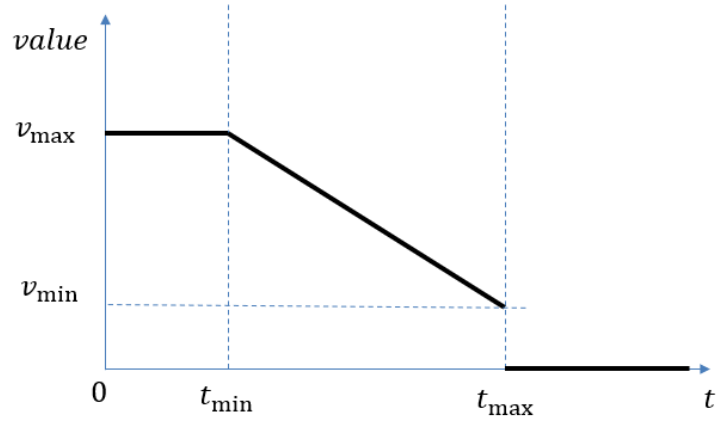
- size, min_value ($v_{\min}$) and max_value ($v_{\max}$) are explained in the table.
- t_min and t_max are fixed parameters that determine when the value of a transaction starts to decrease.

The next two parameters depend on the actions of the bidding agent, and are therefore not part of the input:

- z is the fee (in **satoshi-per-byte**, limited to nonnegative multiples of 5).
- t is the time until transaction is added to some block in the chain, which may depend on the fee, as well as on the other transactions in the mempool.

The utility to the user from a transaction with fee z that is inserted to a block after t seconds is:

$Value(v_{max}, v_{min}, size, t_{max}, t_{min}, t)$ decreases linearly as in the following figure:



The utility to the user is value minus the fee, where $fee(z) = z * size$ if the transactions enters any block, and 0 otherwise (think: why is there no fee if the transaction does not enter a block?)

Note that we assume the agent pays her own bid and not a VCG price.
Note that as the fee is higher, the transaction is likely to be inserted sooner (t will be lower).

You can see an example of a simple bidding agent, who always bids the middle between the minimal and maximal value, with a slight increase for large transactions. This simple agent completely ignores the current state of the mempool.

1. The "forward" bidding agent looks only at the current mempool. It tries to "guess" the time T(z) it will take the transaction to be added, for any fee z. Then it calculates the utility gained (or lost) from every bid. Then it selects the fee z that maximizes the "guessed utility":
$\boldsymbol{GU(z) = Value(v_{max}, v_{min}, size, t_{max}, t_{min}, T(z)) - fee(z)}$

- Every 1 minute exactly, a new block is created, and clears all pending transactions (note this **does not include**[1] the user's transactions) using the greedy pay-your-bid algorithm from part A1.
- The agent calculates for every z, the time T(z) (in seconds) until the first transaction with fee at most z (per byte) is cleared.
- The different z values are multiples of 5 between 0 and 1000 (0,5,10,15,20,25,30,…995,1000).
- Here is an example for transaction ($v_{max} = 25000, v_{min} = 2500, t_{max} = 10000, t_{min} = 1000, size = 10$)

Z	T(z)	GU(z)
0	Never	0
5	Never	0
10	Never	0
15	10000 sec	2500 - 150 = 2250
20	9400 sec	25000-8400/9000*22500 - 200 = 3800
…		
520	1800 sec	25000-800/9000*22500 - 5200 = 17800
525	1200 sec	25000-200/9000*22500 - 5250 = 19250
530	1200 sec	25000-200/9000*22500 – 5300 = 19200

[1] This is since your transactions are not competing with one another. You are not practicing inserting 500 transactions to the blockchain, but practice 500 scenarios of inserting a single transaction.

You should implement the function

Implement ForwardBiddingAgent.bid()

The function returns a tuple (fee, time, utility) where:

- Fee is fee(z) for z that maximizes the guessed utility **GU(z)** (marked in yellow). Note that you should return the total fee, not fee-per-byte.
- Time is T(z)
- Utility is GU(z)

You will be tested on the 500 transactions in the file my_TXs.csv, and on 500 additional similar transactions.

A notes on efficient implementation: sorting long lists takes time. Make sure you do not run redundant sortings on the same data.

2. Competitive part (15 points)

Implement CompetitiveBiddingAgent.bid()

The function has the same signature as the bid function in the other agents. Your goal is to maximize your utility from all transactions.

We will evaluate the performance as follows:

We run a simulation from time t0=1510262000 until time t*=1510268001.

Note that the last transaction is at time 1510267294 but you only see transactions released until time 1510264253 in your data. The input to your bid() function for a user transaction at time current_time will contain all transactions until that time.

Every 1 minute exactly a new block of size 75,000 Bytes will be created, and will be filled using the greedy knapsack algorithms from the list of pending transactions. The data will not contain the field "removed". A transaction is considered "removed" either if it entered a previous block, or if 30 minutes passed from its creation.

As explained in Footnote 1, the transactions in TX_data are not considered pending transactions.

We then check for each of your transactions, to which block it would have entered. This is similar to what you do in question B1, except that more transactions continue to arrive after your transaction is released, so if the fee is too low, your transaction may be postponed indefinitely.

Then we compute your utility from each transaction using the same formula as above. Any transaction that did not enter a block until time t* is considered lost (t=never).

Your grade will be based on the total utility from all of your transactions.

<u>Submission instructions</u>: you should create a zip file names – hw2_ID1_D2.zip that carries :
hw2_part1.py (your solution) ,hw2_ForwardAgent.csv,hw2_CompetitiveAgent.csv
hw2_ForwardAgent.csv contains the columns of: Index, Time, Bid, Utility
hw2_CompetitiveAgent.csv contains the columns of: Index, Bid

The csv functions already implemented!

Make sure there are no sub folders in the zip file. Make sure your file name does not have a double suffix, e.g. hw2_studentID1_studentID2.zip.zip (funny but happens a lot).

Only one student in a group should submit

We may test your code with a different input. The format will be the same (same column and data types), but do not assume anything about the number of rows or their content.