UNIVERSITÀ DI PISA

Analysis of Bitcoin Transactions

Peer to peer systems and blockchains

Kostantino Prifti: 553262

Contents

1 Describe how a real Bitcoin transaction is abstracted by a transaction in the dataset (which fields are eliminated, which are abstracted and how).

1.1 Description of a bitcoin transaction

A raw transaction has the following top-level format:

1. **version:** Version number of the transaction, currently the possible versions are 1 or 2. The length of this field is 4 bytes
2. **tx in count:** Number of inputs in this transaction. The length of this field is variable
3. **tx in:** Transaction input. This field is of type txIn (see description below). Through this field the information of each transaction input is specified. The length of this field is variable. This field is repeated for each transaction input
4. **tx out count:** Number of outputs in this transaction. The length of this field is variable
5. **tx out:** Transaction output. This field is of type txOut (see description below). Through this field the information of each transaction output is specified. The length of this field is variable. This field is repeated for each transaction output
6. **lock time:** Iindicates the earliest time or earliest block when that transaction may be added to the blockchain. The length of this field is 4 bytes

TxIn: A Transaction Input (Non-Coinbase)

Each non-coinbase input spends an outpoint from a previous transaction. (Coinbase inputs are described below.) TxIn has the following top-level format:

1. **hash:** The TXID of the transaction holding the output to spend. The length of this field is 32 byte
2. **index:** The output index number of the specific output to spend from the transaction. The first output is 0x00000000. The length of this field is 4 bytes
3. **script bytes:** The number of bytes in the signature script. Maximum is 10,000 bytes.
4. **signature script:** A script-language script which satisfies the conditions placed in the outpoint's pubkey script.
5. **sequence:** Sequence number. The length of this field is 4 bytes

TxOut: A Transaction Output

TxOut has the following top-level format:

1. **value:** Number of satoshis. The length of this field is 8 bytes
2. **pk script bytes:** Number of bytes in the pubkey script. Maximum is 10,000 bytes.
3. **pk script:** Defines the conditions which must be satisfied to spend this output.

Coinbase Input: The Input Of The First Transaction In A Block

Coinbase Input has the following top-level format:

1. **hash:** A 32-byte null, as a coinbase has no previous outpoint.
2. **index:** 0xffffffff, as a coinbase has no previous outpoint.
3. **script bytes:** The number of bytes in the coinbase script, up to a maximum of 100 bytes.
4. **height**
5. **coinbase script**
6. **sequence**

Finally the **TX ID**, that is the indifier of the transaction, is generated by hashing transaction data through **SHA256** twice.

1.2 Abstractions and simplifications carried out in the dataset of bitcoin transactions

Both simplifications and abstractions have been made in the dataset, let's start by describing which simplifications have been made.

1.2.1 Simplifications

Simplification means that some fields of a real bitcoin transaction are not reported in the dataset. In particular these fields are:

1. **version**
2. **tx in count**
3. **tx out count**
4. **index:** This index refers to the index that we find in **TxIn**
5. **script bytes:** Field inside **TxIn**
6. **sequence:** Field inside **TxIn**
7. **pk script bytes:** Field inside **TxOut**
8. **lock time**

1.2.2 Abstractions

In order to better understand which abstractions are present in the dataset, I report briefly how the fields of the dataset files are structured:

1. **inputs.csv:**
   - id: unique id of this input
   - tx id: transaction this input is part of
   - sig id: scriptSig public key id, 0 if coinbase tx, - 1 if nonstandard scripts used
   - output id: id of the previous output being referenced, - 1 if coinbase tx
2. **outputs.csv:**
   - id: unique id of this output
   - tx id: transaction this output is part of
   - pk id: scriptPubKey public key id, - 1 if nonstandard scripts used
   - value
3. **transactions.csv:**
   - id: identifier of the transaction
   - block id: id of the block containing this transaction

Abstractions:

1. The *tx id* which should be a 256-bit hash produced by the double hashing, within the dataset is represented as an integer index, this is done to reduce the size of the dataset
2. To represent a coinbase transaction in the inputs.csv file the *output id* fields are set to -1 and the *sig id* field to 0
3. A big abstraction has been made regarding the *sig id* field in the inputs.csv file and regarding the *pk id* field in the outputs.csv file. In particular, the *sig id* field denotes the public key used in the scriptSig, therefore the scriptPubKey is abstracted using *pk id* denoting the public key used in the scriptPubKey. Therefore a transaction is considered valid if the *sig id* in input is equal to the *pk id* of the unspent output identified by the *output id.*

2 Check if all the data contained in the dataset is consistent, and if some data is invalid, describe what is the problem of that data and remove it from the dataset.

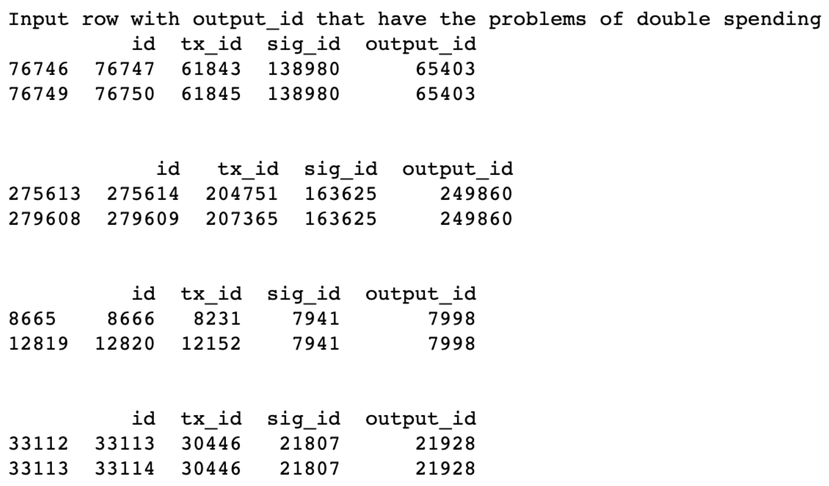
In order to verify the consistency and validity of the data within the dataset, various tests were carried out:

1. **Check if the same output id appears more than once as an input:** This represents a double spending
2. **Check if TX ID in the inputs.csv file is valid:** Test passed, all the *tx ids* present in the inputs.csv file are valid
3. **Check if TX ID in the outputs.csv file is valid:** Test passed, all the *tx ids* present in the outputs.csv file are valid

4. **Check if OUTPUT ID in the inputs.csv file is valid**
5. **Check if there are any missing BLOCK ID:** In the dataset we have 100018 blocks with index between (0, 100017). The *block id* is consecutive so there should be no missing index between (0, 100017) unfortunately this is not the case as the test showed that the block id = 91857 is missing from the dataset.
6. **Check for each transaction if the sum of the inputs is greater than or equal to the sum of the outputs**
7. **Check if PK ID of the output that is used in input is equal of SIG ID of the input**
8. **Check if the reward of a coinbase transaction is equal to 5000000000 satoshi**

2.1 Check if the same output id appears more than once as an input:

Through this analysis it is verified that the same output is not spent twice, if this happens it means that it is a double spending. The results of this analysis showed that there are some *output ids* that are used more than once, see Figure 1.



```
Input row with output_id that have the problems of double spending
            id  tx_id  sig_id  output_id
76746   76747  61843  138980      65403
76749   76750  61845  138980      65403


               id   tx_id  sig_id  output_id
275613  275614  204751  163625     249860
279608  279609  207365  163625     249860


            id  tx_id  sig_id  output_id
8665     8666   8231    7941       7998
12819   12820  12152    7941       7998


            id  tx_id  sig_id  output_id
33112   33113  30446   21807      21928
33113   33114  30446   21807      21928
```

Figure 1: Inputs row with same *output id*

Output id used twice:

- **65403**: Regarding this *output id* the invalid transaction is the transaction with *tx id* = 61845 this because it has a *tx id* greater than the other transaction that uses the same *output id*, so it means that it is using an output that has already been spent.
- **7998** Regarding this *output id* the invalid transaction is the transaction with *tx id* = 12152 this because it has a *tx id* greater than the other transaction that uses the same *output id*, so it means that it is using an output that has already been spent.
- **21928:** In this case there are not two different transactions using the same *output id* but it is the same transaction, so the transaction with *tx id* = 30446 is not valid
- **249860:** In this case the analysis showed an interesting thing, in fact in this case it is not the transaction with the highest *tx id* that is not valid, but the transaction with *tx id* = 204751. This is because the *output id* = 249860 was produced by a transaction with *tx id* = 207362, so the transaction with *tx id* = 204751 is trying to spend an output that has not yet been produced, so it is considered invalid

Therefore transactions with *tx id* = 61845, 12152, 30446, 204751 are not valid. Before eliminating these transactions, a further test was made, it was tested whether the output produced by these transactions was used for other transactions, because if so we have to eliminate all transactions that "descend" from them. Fortunately, the outputs produced by invalid transactions are not used as input for other transactions.

2.2 Check if OUTPUT ID in the inputs.csv file is valid

This test tested whether the *output id* field of each row of the inputs.csv file is valid, i.e. if it is an *ouput id* present in the output.csv file.
The analysis showed that the transaction with *tx id* = 15698 takes as input an *output id* which does not exist, in particular the *output id* = 265834. Therefore the transaction with *tx id* = 15698 is not valid. As for the previous test, also in this case, before deleting the invalid transaction, it was tested that the output produced by this transaction is not used in another transaction, also in this case this problem did not arise.

2.3 Check for each transaction if the sum of the inputs is greater than or equal to the sum of the outputs

In this test, each transaction was analyzed to check if the sum of all inputs is greater than or equal to the sum of the outputs, if this is not the case then the transaction is not valid.
The analysis showed that the transaction with *tx id* = 100929 is not valid as the total of the output is greater than the total of the inputs. In particular, input = 5000000000 satoshi and output = 5000000010 satoshi. Also in this case, before deleting the invalid transaction, it was tested that the output produced by this transaction is not used in another transaction, also in this case this problem did not arise.

2.4 Check if PK ID of the output that is used in input is equal of SIG ID of the input

In order to spend an output, the condition must be met that the output *pk id* is equal to the *sig id* in input, if not, the transaction cannot be considered valid. Analysis showed that the transaction with *tx id* = 138278 does not match this condition. In fact the *sig id* in input is: 139250 while the output that is spent has *pk id* = 16020. Also in this case, before deleting the invalid transaction, it was tested that the output produced by this transaction is not used in another transaction, also in this case this problem did not arise.

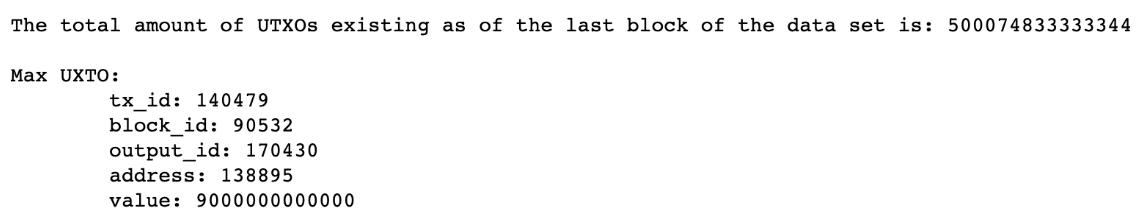
2.5 Check if the reward of a coinbase transaction is equal to 5000000000 satoshi

We know that the reward of a coinbase transaction must be 5000000000 satoshi. Through this text all the coinbase transactions have been analyzed. The test showed that there are 915 Coinbase transactions that do not have the reward of 5000000000 satoshi. However, these transactions were not eliminated from the dataset mainly for two reasons:

1. Eliminating these transactions would produce an inconsistency, in fact, since these are Coinbase transactions, if we deleted them the dataset would be in a situation in which there are more blocks than Coinbase transactions, we know instead that the number of blocks must be equal to the number of Coinbase transactions given that the Coinbase transaction represents the reward for a miner who has added a new block to the blockchain.

2. Secondly, they have not been eliminated for simplicity, as the analysis showed that 641 outputs produced by these transactions are used for other transactions and so on in cascade.

3 Compute the total amount of UTXOs (Unspent Transaction Outputs) existing as of the last block of the data set, i.e. the sum of all the Transaction outputs balances on the UTXO set of the last block. Which UTXO (TxId, blockId, output index and address) has the highest associated value?



```
The total amount of UTXOs existing as of the last block of the data set is: 500074833333344

Max UXTO:
        tx_id: 140479
        block_id: 90532
        output_id: 170430
        address: 138895
        value: 9000000000000
```

Figure 2: Total and Max UXTO

4 The distribution of the block occupancy, i.e. of the number of transactions in each block in the entire period of time. Furthermore, show the evolution in time of the block size, by considering a time step of one month.

An additional file, dates.csv, was used to carry out this type of analysis. This file reports the timestamp of the creation of each block in the considered period. In Figure 3 we can see the number of transactions carried out in each month of the considered period.

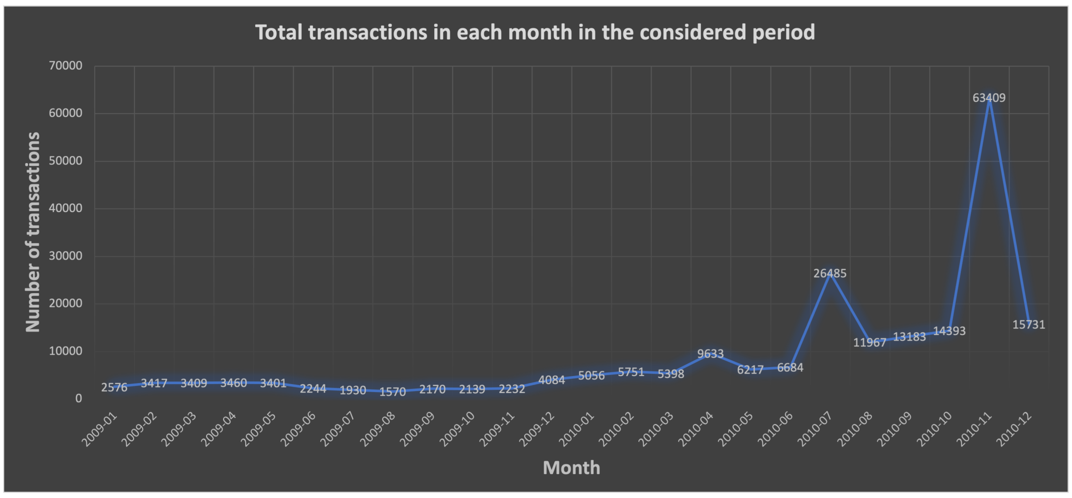


Figure 3: Month Transactions

5 The total amount of bitcoin received from each public key that has received at least one COINBASE transaction, in the considered period, and show a distribution of the values

The total number of coinbase transactions is 100018, this is a value that we should expect as there are 100018 blocks. In Figure 4 we see the percentages of coinbase and non-coinbase transactions.

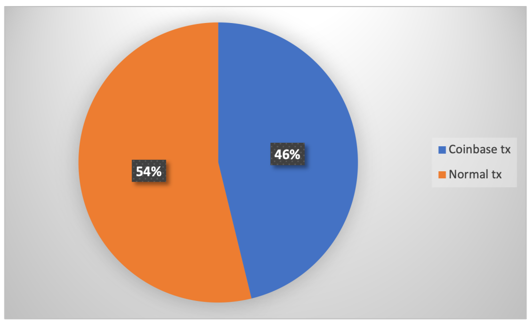


Figure 4: Coinbase and Normal Transaction

Figure 5 illustrates the reward value for each *pk id*. From the graph in Figure 6 it is evident that most of the *pk ids* have associated a reward of 5000000000 satoshi, in fact the average value is 4993257237 satoshi. Instead from the graph in Figure 7 (it is the same graph as Figure 6 but with the data shown differently) it is noted that there are some *pk id* that have a reward lower than 5000000000 satoshi.

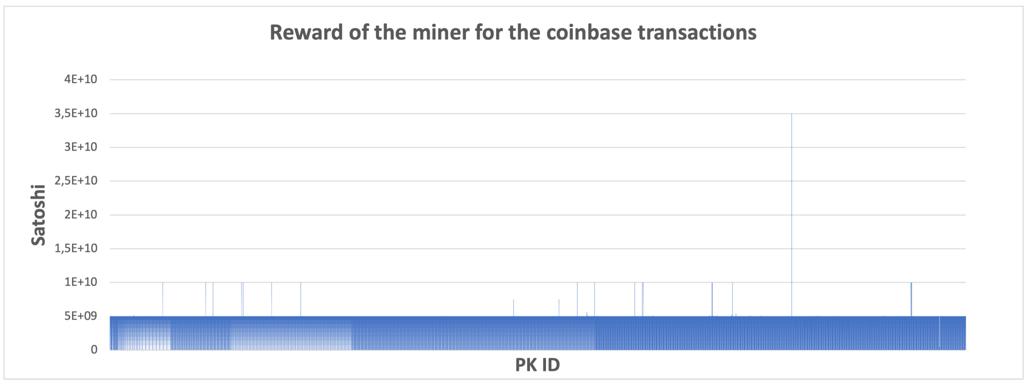


Figure 5: Reward of the miner for the coinbase transactions 1

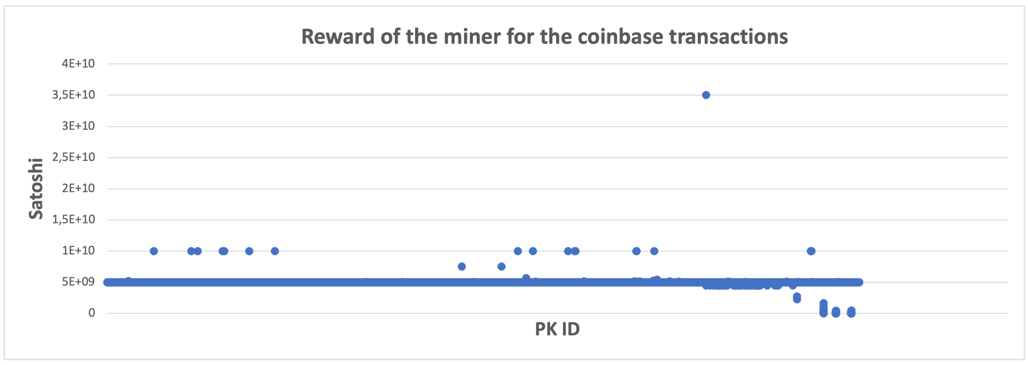


Figure 6: Reward of the miner for the coinbase transactions 2

The *pk id* that has associated the highest reward value is:109834 with a reward value = 35000000000 satoshi.

6 The distribution of the fees spent in each transaction in the considered period

Within the dataset there are 1301 transactions with fees. The transaction with *tx id* = 105281 has the highest fees with a value of MaxFees = 10000000000 sathosi. To calculate the distribution, the value of fees between 0 and MaxFees was divided into 20 ranges. See Figure 7.

```
Range 1: 0-500000000 satoshi
Range 2: 500000000-1000000000 satoshi
Range 3: 1000000000-1500000000 satoshi
Range 4: 1500000000-2000000000 satoshi
Range 5: 2000000000-2500000000 satoshi
Range 6: 2500000000-3000000000 satoshi
Range 7: 3000000000-3500000000 satoshi
Range 8: 3500000000-4000000000 satoshi
Range 9: 4000000000-4500000000 satoshi
Range 10: 4500000000-5000000000 satoshi
Range 11: 5000000000-5500000000 satoshi
Range 12: 5500000000-6000000000 satoshi
Range 13: 6000000000-6500000000 satoshi
Range 14: 6500000000-7000000000 satoshi
Range 15: 7000000000-7500000000 satoshi
Range 16: 7500000000-8000000000 satoshi
Range 17: 8000000000-8500000000 satoshi
Range 18: 8500000000-9000000000 satoshi
Range 19: 9000000000-9500000000 satoshi
Range 20: 9500000000-10000000000 satoshi
```

Figure 7: Range considered

Figure 8 shows how many transactions with fees and how many without fees there are within the dataset. It is clear that there are very few transactions with fees.

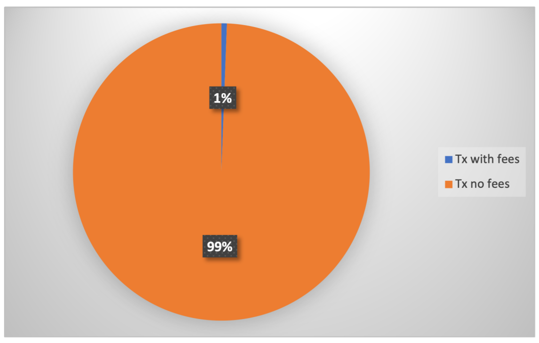


Figure 8: Transaction with fees and without fees

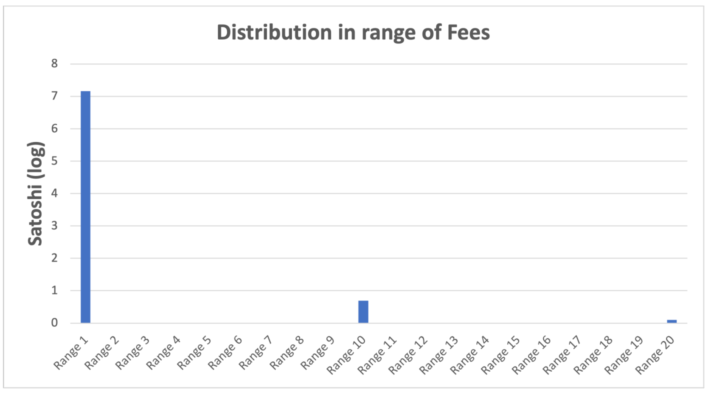


Figure 9: Distribution of transactions with fees

In Figure 10 we can see how the fees are distributed in the different ranges, in order to better visualize the data on the y axis a logarithmic scale has been applied. It is clear from the graph that most of the transactions with fees fall within the first range, so for this reason the first range has been further subdivided into another 1000 sub-ranges. From this analysis it emerged that 1199 transactions, i.e. 92.15% of transactions with fees, have fees in the range 500000-1000000 satoshi.

7 Propose one further analysis of your choice. Give a brief description of the analysis and report the results

In this analysis, the sum of the fees for each block was calculated. What I expect is that as the *block id* increases, the sum of the fees in that block also increases. Figure 9 shows the value of the fees per block, in the graph only the blocks that have the sum of the fees greater than 0 have been considered, also in this case for a better visualization of the data a logarithmic scale has been applied to the y axis.

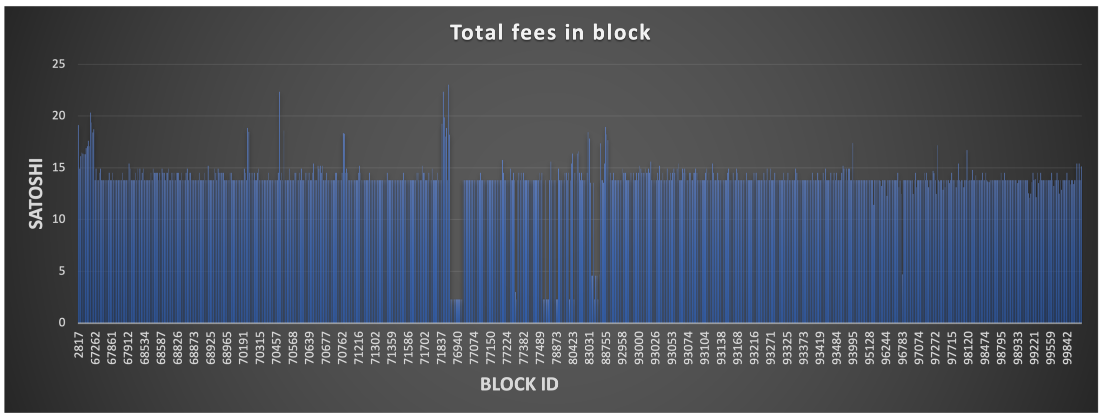
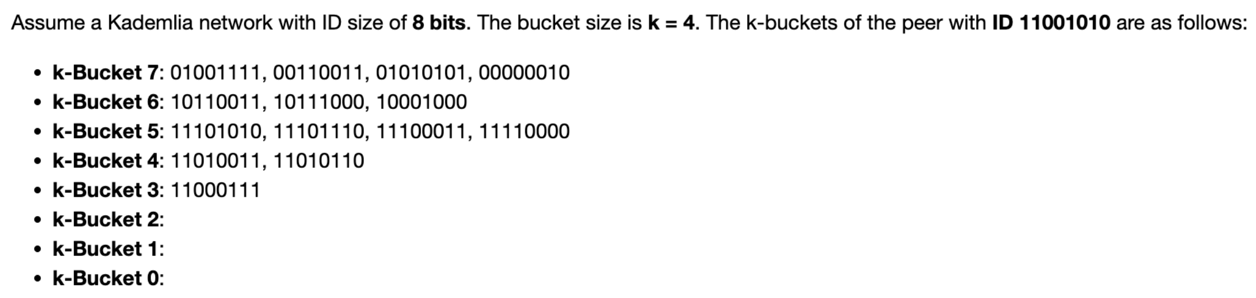


Figure 10: Fees per block

Contrary to what was expected, except for some peaks, the total fees per block remained fairly stable in the period considered, this probably because, being the first two years of life of the Bitcoin blockchain, the phenomenon of the increase in fees is not yet evident. The block with the highest value of fees is the *block id:* 75047 with a total value of fees: 10000000000 satoshi.

8 Kademlia

Assume you have the Kademlia network in Figure 11



Assume a Kademlia network with ID size of **8 bits**. The bucket size is **k = 4**. The k-buckets of the peer with **ID 11001010** are as follows:

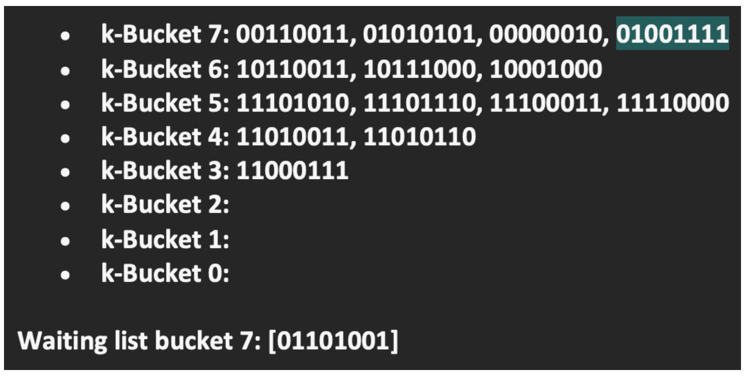
- **k-Bucket 7**: 01001111, 00110011, 01010101, 00000010
- **k-Bucket 6**: 10110011, 10111000, 10001000
- **k-Bucket 5**: 11101010, 11101110, 11100011, 11110000
- **k-Bucket 4**: 11010011, 11010110
- **k-Bucket 3**: 11000111
- **k-Bucket 2**:
- **k-Bucket 1**:
- **k-Bucket 0**:

Figure 11: Kademlia Network

Unlike what we saw in the lesson for this exercise on kademlia we assume that each bucket has a waiting list.
Question:

1. **Messages from following nodes arrive in this given order: 01101001, 10111000, 11110001, 10101010, 11100011, 11111111. How do the buckets, the orderings in the buckets and the waiting lists change?**

   - **01101001:** The node: 01101001 should be inserted in the k-buckets 7 but at this moment it is full. Therefore, since it is full, the node: 01001111 is contacted which is at the head of the bucket, assuming that it responds the node: 01001111 is queued to the bucket and node: 01101001 is inserted in the waiting list. See Figure 12.

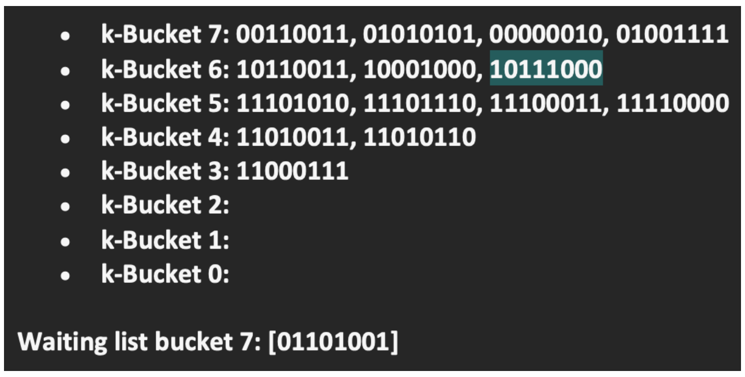


- **k-Bucket 7: 00110011, 01010101, 00000010, 01001111**
- **k-Bucket 6: 10110011, 10111000, 10001000**
- **k-Bucket 5: 11101010, 11101110, 11100011, 11110000**
- **k-Bucket 4: 11010011, 11010110**
- **k-Bucket 3: 11000111**
- **k-Bucket 2:**
- **k-Bucket 1:**
- **k-Bucket 0:**

Waiting list bucket 7: [01101001]

Figure 12: After receiving the message from the node: 01101001

   - **10111000:** The node: 10111000 is already present in k-bucket 6 therefore being the last node inside the k-bucket 6 that has been contacted, it is moved to the queue. See Figure 13.

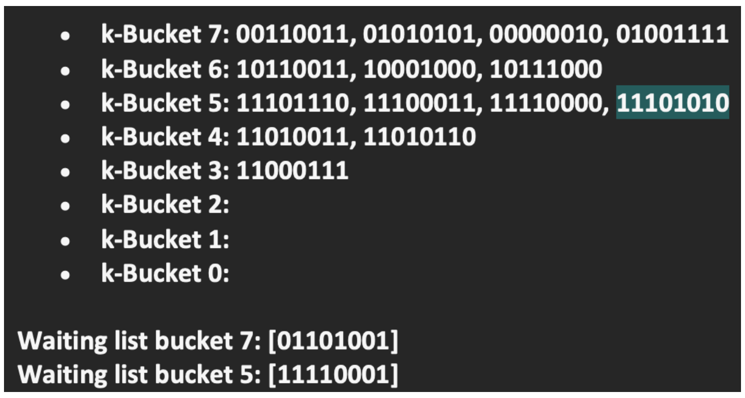


- **k-Bucket 7: 00110011, 01010101, 00000010, 01001111**
- **k-Bucket 6: 10110011, 10001000, 10111000**
- **k-Bucket 5: 11101010, 11101110, 11100011, 11110000**
- **k-Bucket 4: 11010011, 11010110**
- **k-Bucket 3: 11000111**
- **k-Bucket 2:**
- **k-Bucket 1:**
- **k-Bucket 0:**

Waiting list bucket 7: [01101001]

Figure 13: After receiving the message from the node: 10111000

   - **11110001:** The node 11110001 should be inserted in the k-buckets 5 but at this moment it is full. Therefore, since it is full, the node: 11101010 is contacted which is at the head of the bucket, assuming that it responds the node: 11101010 is queued to the bucket and node 11110001 is inserted in the waiting list. See Figure 14.

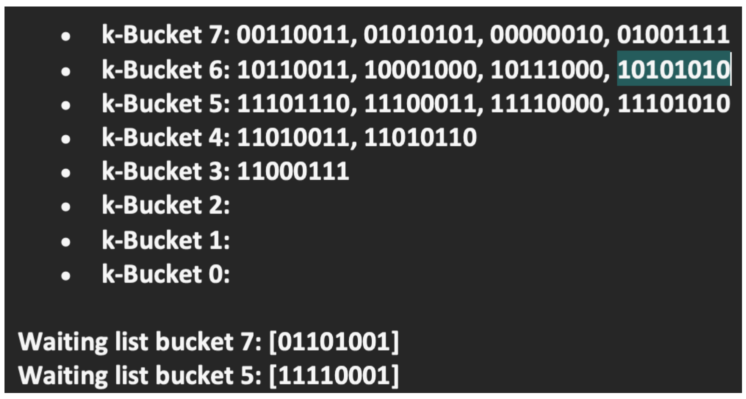


- **k-Bucket 7: 00110011, 01010101, 00000010, 01001111**
- **k-Bucket 6: 10110011, 10001000, 10111000**
- **k-Bucket 5: 11101110, 11100011, 11110000, 11101010**
- **k-Bucket 4: 11010011, 11010110**
- **k-Bucket 3: 11000111**
- **k-Bucket 2:**
- **k-Bucket 1:**
- **k-Bucket 0:**

Waiting list bucket 7: [01101001]
Waiting list bucket 5: [11110001]

Figure 14: After receiving the message from the node: 11110001

   - **10101010:** The node 10101010 should be placed in k-buckets 6 which is not full at this time, so node 10101010 is queued in the bucket 6. See Figure 15.

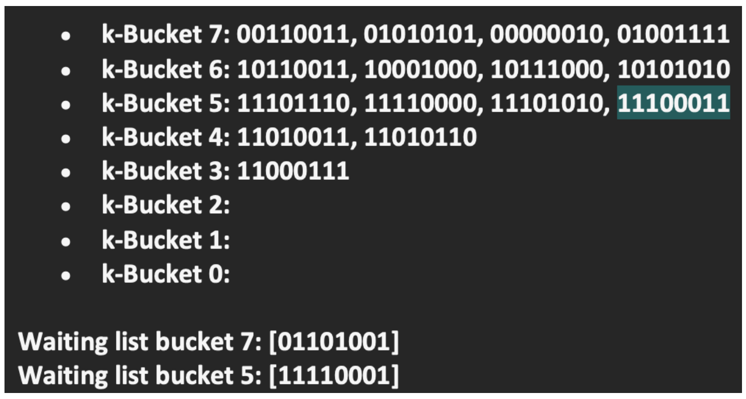


```
    • k-Bucket 7: 00110011, 01010101, 00000010, 01001111
    • k-Bucket 6: 10110011, 10001000, 10111000, 10101010
    • k-Bucket 5: 11101110, 11100011, 11110000, 11101010
    • k-Bucket 4: 11010011, 11010110
    • k-Bucket 3: 11000111
    • k-Bucket 2:
    • k-Bucket 1:
    • k-Bucket 0:

Waiting list bucket 7: [01101001]
Waiting list bucket 5: [11110001]
```

Figure 15: After receiving the message from the node: 10101010

- **11100011:** The node: 11100011 is already present in k-bucket 5 therefore being the last node inside the k-bucket 5 that has been contacted, it is moved to the queue.

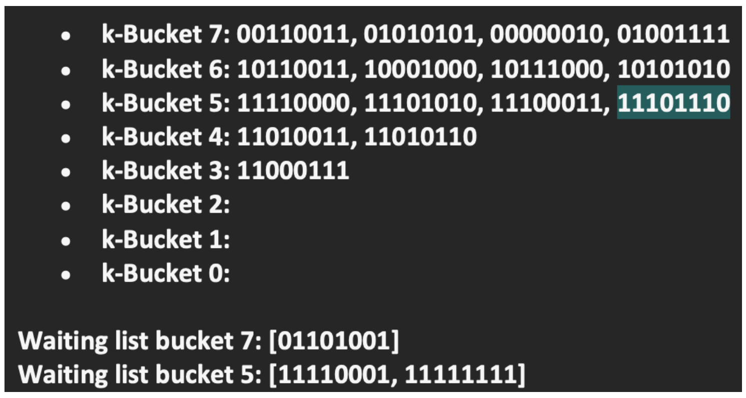


```
    • k-Bucket 7: 00110011, 01010101, 00000010, 01001111
    • k-Bucket 6: 10110011, 10001000, 10111000, 10101010
    • k-Bucket 5: 11101110, 11110000, 11101010, 11100011
    • k-Bucket 4: 11010011, 11010110
    • k-Bucket 3: 11000111
    • k-Bucket 2:
    • k-Bucket 1:
    • k-Bucket 0:

Waiting list bucket 7: [01101001]
Waiting list bucket 5: [11110001]
```

Figure 16: After receiving the message from the node: 11100011

- **11111111:** The node 11111111 should be inserted in the k-buckets 5 but at this moment it is full. Therefore, since it is full, the node: 11101110 is contacted which is at the head of the bucket, assuming that it responds the node: 11101110 is queued to the bucket and node 11111111 is inserted in the waiting list. See Figure 17.

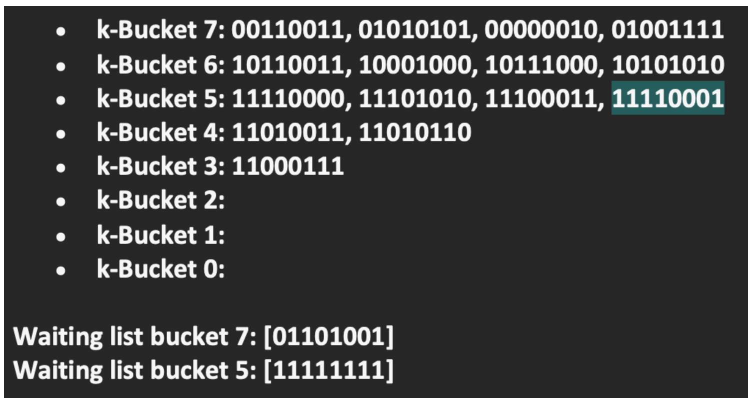


```
    • k-Bucket 7: 00110011, 01010101, 00000010, 01001111
    • k-Bucket 6: 10110011, 10001000, 10111000, 10101010
    • k-Bucket 5: 11110000, 11101010, 11100011, 11101110
    • k-Bucket 4: 11010011, 11010110
    • k-Bucket 3: 11000111
    • k-Bucket 2:
    • k-Bucket 1:
    • k-Bucket 0:

Waiting list bucket 7: [01101001]
Waiting list bucket 5: [11110001, 11111111]
```

Figure 17: After receiving the message from the node: 11111111

2. **Now the node detects that peer 11101110 cannot be reached anymore, what is the reaction?** As node:11101110 is not responding it is deleted from k-bucket 5, the element at the top of the waiting list of bucket 5 is taken and queued at k-bucket 5. See Figure 18.



- k-Bucket 7: 00110011, 01010101, 00000010, 01001111
- k-Bucket 6: 10110011, 10001000, 10111000, 10101010
- k-Bucket 5: 11110000, 11101010, 11100011, 11110001
- k-Bucket 4: 11010011, 11010110
- k-Bucket 3: 11000111
- k-Bucket 2:
- k-Bucket 1:
- k-Bucket 0:

Waiting list bucket 7: [01101001]
Waiting list bucket 5: [11111111]

Figure 18: After peer 11101110 cannot be reached anymore

3. **Which addresses would the peer reply to a lookup looking for ID 11010010?** We assume a value of alpha = 1. The value of alpha represents how many parallel requests are made. Making this assumption it is evident that the first node to contact is the node: 11010011 since it is the node that is less distant in terms of XOR, in fact 11010011 XOR 11001010 = 1.