WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND MANAGEMENT

FIELD: Informatyka (INF)
SPECIALIZATION: Brak specjalnoci

ENGINEERING THESIS

An application for tracking the flow of resources for Bitcoin cryptocurrency

Program do ledzenia przepywu rodkw w sieci Bitcoin

AUTHOR:
Marcin Pieczka

SUPERVISOR:

Dr in., Radosaw Michalski, Katedre Inteligencji Obliczeniowej

GRADE:

WROCŁAW 2018

Contents

Chapter 1

Goals

Every process of analyzing data starts with accessing the data. When analyzing Bitcoin blockchain, the first step might be the hardest one. Currently Bitcoin blockchain contains over 160GB of raw binary data, and everyone who attempts to analyze it needs to possess an efficient and reliable way to work with it. Moreover, users should have obvious place for creating additional APIs that will be placed on the same server as the data. The reason for that is that operations requiring big amounts of data, but return results of significantly smaller size, can be implemented efficiently, gathered in one place and blend in with the system.

The goal of this work is to create an application that fulfills the following requirements:

- allows fast access to blockchain data
- allows access over the network
- updates its data in constant manner
- provides API for Python and R
- is easy to install

Chapter 2

Cryptocurrencies

2.1 Overview

This chapter will explain concept of cryptocurrencies, in many cases based on example of Bitcoin. The connection between cryptography and cryptocurencies will be shown, as well as means that are used to ensure correctness.

2.2 Definition

Cryptocurrency is a digital currency in which encryption techniques are used to regulate the generation of units of currency and verify the transfer of funds, operating independently of a central bank. [2]

2.3 Security

Every currency operates within its defined set of rules, and has to have means to ensure these rules are obeyed. For regular money as we know it today security is provided by central authorities in way of creating banknotes that are hard to counterfeit, creating law that penalizes creating counterfeits, creating banking systems for digital money transfer that ensures those rules and is secure from malicious actors. Cryptocurrencies have their own way of ensuring their rules, but instead of central authority, those rules are ensured by cryptography and probability.

2.4 Cryptography

Cryptocurrencies are in big part based on cryptography to ensure the rules at witch it operates. Cryptography helps not only to ensure that fraudulent and malicious operations can't have their place, but as well to encode rules of creating new units of those currencies

Cryptography is a field of knowledge that has its focus on protecting information from unauthorized access. Nowadays it is considered to be part of mathematics as well as informatics.

Cryptography can be classified as:

- Symmetrical - Information is encrypted and decrypted with the same key, this creates problem of securely transferring the key.

- Asymmetrical - Information is encrypted with different key than the key that is used for decryption. One key is usually called the public key, and is used to encrypt messages. Because public key cannot be used to decrypt message it can be shared freely without concern. Second key is called private key an is used to decrypt messages. Private key should never be shared, because everyone possessing this key can decrypt messages addressed to original owner of this key.

Public key cryptography was developed in 1970s and is widely used to ensure security of information today. Mathematical functions ware discovered that are easy to calculate, but nearly impossible to reverse. Bitcoin uses elliptic-curve functions as a base of calculating public key.[3]

Aside of encryption, bitcoin uses cryptographic hash functions in its inner workings. Cryptographic hash functions map strings of arbitrary length to constant number of bits called digest, typically from 128 to 512.

One of the main properties of hash functions is that they work one-way only. This means that for given hash digest it is impossible to find original message, but other way around it is performant operation. [4]

2.5 Cryptography in Bitcoin

Bitcoin uses cryptography in many of its mechanisms, in this section this usage will be described.

2.5.1 Usage of hash functions in bitcoin

Chapter 3

Alternative software providing similar features

blockchain.info

Web application blockchain.info provides free access to Bitcoin blockchain data by either website, JSON API or APIs dedicated to specific languages including Python, but without support for R. Number of requests is limited.

Relevant APIs provided by blockchain.info:

- getting single block, by block hash
- getting single block, by hight
- getting multiple block headers
- getting single transaction, by transaction hash
- getting all transactions of single or multiple addresses

Most common usage scenario in analytic context is getting range of blocks, for example all blocks from 10.04.2017 to 20.04.2017. Nonetheless, blockchain.info does not provide simple way of getting such data. To achieve this, we would have to make thousands of requests to API providing us with single block data, and this wouldn't be fast enough.

Another and the biggest problem is the API call limit that would make working with this application impossible for larger query's.

blockexplorer.com

This web application is very similar to blockchain.info in almost every aspect, although there are some differences. Data is accessible either by website or by JSON API, but there is no Python or R API provided. Set of APIs is almost identical to blockchain.info and does not provide easy way to get multiple blocks. The main difference is that there is no official API call limit, but using tool that is not controlled by us means that such limit can appear every moment.

libbitcoin-database

`https://github.com/libbitcoin/libbitcoin-database Last visited 25.04`

This piece of software, after installation builds in-memory database of Bitcoin blockchain, its description promises high performance. Because of almost non existing documentation it is hard to write much about it.

Although this software is actively developed, and at first sight it might seam like solution that fits needs of our users, some problems can be found. One drawback is coming from its biggest selling point - being in-memory database makes it requiring large amounts of RAM. This problem makes it not viable for hardware that is accessible for the average user. Next problem lies in its API that allows connecting to the database via C/C++ library, which would make usage in R and Python at least problematic. Another problem is lack of useful documentation, which would make the usage largely troublesome.

BitcoinDatabaseGenerator

`https://github.com/ladimolnar/BitcoinDatabaseGenerator Last visited 25.04`

BitcoinDatabaseGenerator is a data transfer tool that can feed SQL database with blockchain data. It was written in C# and as its author states, it is only meant to be run on Windows machines. From the documentation we know that this software should be ran every time we want to update our database. Database schema, that is created after update operation, consists of separate tables for blocks, transactions, transaction inputs and so forth. Such database schema would require joining tables in multitude of usage scenarios, which is known to be slow operation.

Although you can easily connect to SQL database from both R and Python, I find working with SQL databases in such case cumbersome. Abstracting away relational structure of data to objects might be necessary.

Chapter 4

Design

4.1 User requirements

4.1.1 User stories

The following sentences sum up the discussions with people that consider using this system.

As a user, I want to access blockchain data from my Python/R script so that data analysis and data accessing can be made in the same code.

As a user, I want the data be in a format idiomatic to Python/R so that I don't have to convert it myself.

As a user, I want easy access to block data from range of time or block height so that I can spend my time working with the data, and not with accessing it.

As a user, I want the installation not to require complicated operations so that I can do it without specialized knowledge.

As a user, I want to be able to run this software on my linux server so that I don't have to learn new operating system to be able to use it.

As a user, I want fast access to the data so that my analytic scripts will be pleasurable to work with.

As a user, I want to have obvious place to create my own APIs on server so that my new data hungry features can be placed on the same server as data for better performance.

4.2 System inputs and outputs

In this section I will specify inputs and outputs of the system. This will then help me to discover what transformation the input data will undergo, and what components are needed to provide outputs efficiently. Here I will consider only part of the project that is responsible for serving data, not the part that will be responsible for hosting future APIs.

4.2.1 Inputs

Bitcoind BLK files

The only source of bitcoin block data will be BLK files stored by full bitcoin node. Daemon process bitcoind gets blocks from neighboring nodes and stores them in data directory. The blk files in default configuration store up to 128MB of raw network format block data. Blocks are stored in order in which they come from network. Library's that handle parsing BLK files exist, so accessing this data should not be a problem.

One important thing to mention about bitcoin transactions stored in blocks is the fact that address of the sender is described as output of previous transaction.

Request for blocks

will be the way user communicates with the system. In this request user will specify which blocks he wants to receive. Usually those will be blocks from specified range of time, or range of height.

4.2.2 Outputs

Response on request for blocks

will contain block data in format understandable by user

4.3 Persistence

The fact that transaction sender address is not stored in bitcoin blocks directly, but by reference to other transaction, creates responsibility for the system. This responsibility is to discover addresses by finding referred transaction, and getting the address of recipient. This of course requires substantial amount of time which should not be added to the time of user waiting for his response.
Additional thing to consider is the time needed to transform raw binary bitcoin block to widely used data format such as JSON. Knowing this we will get to the conclusion that we will need some type of persistence. Although some custom way to store this data might be better suited for our needs, I will be using a database because of constraint of time I have available to build this solution.

4.3.1 Database operations

Lets consider database operations that will need to be fast, and those that are not so important in this regard.

Least important operation will be data addition. End user will not be performing these operations, and its performance will be my last priority.

Most important operation will be querying blocks, by its hash, time, and other attributes, especially querying for range of consecutive blocks described by time frame and height range.

Another important operation is operation of getting transactions by hash. This operation will be needed for discovering address of transaction sender. Although this operation will not be in usage scenarios started by end user, the number of such operations needed to discover sender address of every bitcoin transaction makes it crucial for this operation to be fast.

4.3.2 Database choice

At the beginning let's simplify the choice between RDBMS and NoSQL databases. Out of many NoSQL possibilities I have chosen MongoDB database based on some research of different NoSQL systems strengths and weaknesses.

Let's lay out some facts that will help to decide whether to use relational database, or MongoDB.

- Storing blocks in RDBMS in multiple tables, for instance table for blocks and table for transactions, would require joining those tables which is known to be slow operation.
- To achieve fast querying, the data should be strongly denormalized.
- Storing blocks in RDBMS in one table can be impossible in most systems, due to attribute count limit.
- You can achieve comparable performance from MongoDB and RDBMS, but MongoDB makes storing denormalized data idiomatic.
- MongoDB have APIs for Python and R that makes data available as objects.

Based on these facts, the database I will use will be MongoDB. Because of that choice I hope to easily achieve needed performance and native support for Python and R.

4.3.3 Database structure

Collections in MongoDB don't have structure defined upfront. To have mandatory attributes, actor responsible for data insertion needs to enforce it. Because of this, writing about database structure is not most accurate term. It is easier to think about it as structure of data, and in my system the structure of data will not change in big degree compared to form I will obtain it in. The only difference is, that by default, bitcoin transactions don't specify address of sender. Additionally I have added value that is the time that had passed between receiving resources which were input of transaction, and that transaction.

One document in MongoDB collection will be one bitcoin block, its attributes are:

- bits - current target in compact format [1]
- difficulty
- hash - hash of this block
- height - number of blocks before this block in blockchain
- merkle root - hash based on all transactions of block

- n tx - number of transactions
- nonce - value set by miner to achieve needed block hash
- prev hash - hash of previous block
- size
- timestamp - time at which block had been mined
- transactions - list of transactions with each transaction consisting of:
  - hash - transaction hash
  - inputs - list of data regarding addresses from which resources had been transfered, consisting of:
    - transaction hash - hash of transaction in which sender of this transaction got resources spent in this transaction
    - transaction index - points to certain output in transaction described by previous attribute
    - output timestamp - additional value described before this list of attributes
    - addresses - list of addresses, each containing:
      - address
      - hash
      - public key
      - type
  - outputs - list of data regarding addresses to which resources had been transfered, consisting of:
    - value
    - addresses - list of addresses, each containing just like in inputs:
      - address
      - hash
      - public key
      - type

4.3.4 Database indexes

Database indexes serve purpose of making queries faster and more efficient on cost of space and slower data insertion time. Because performance of queries is one of key user needs, and space and data insertion time are very low on priority list the indexes will be added to all of the fields that data will be queried by.

Indexed attributes:

- block hash block timestamp transaction hash - needed internally for discovering input addresses

4.4 High level architecture

This system will consist of separate entities, that will have their own, well defined responsibilities. This will make those components and overall system easy to understand and maintain.

System will consist of:

- Bitcoin node with its data directory
- Database for storing processed blocks
- Process constantly updating database with new blocks that are gathered by bitcoin node which I will refer to as database updater
- Web application ready to implement complex functionalities in

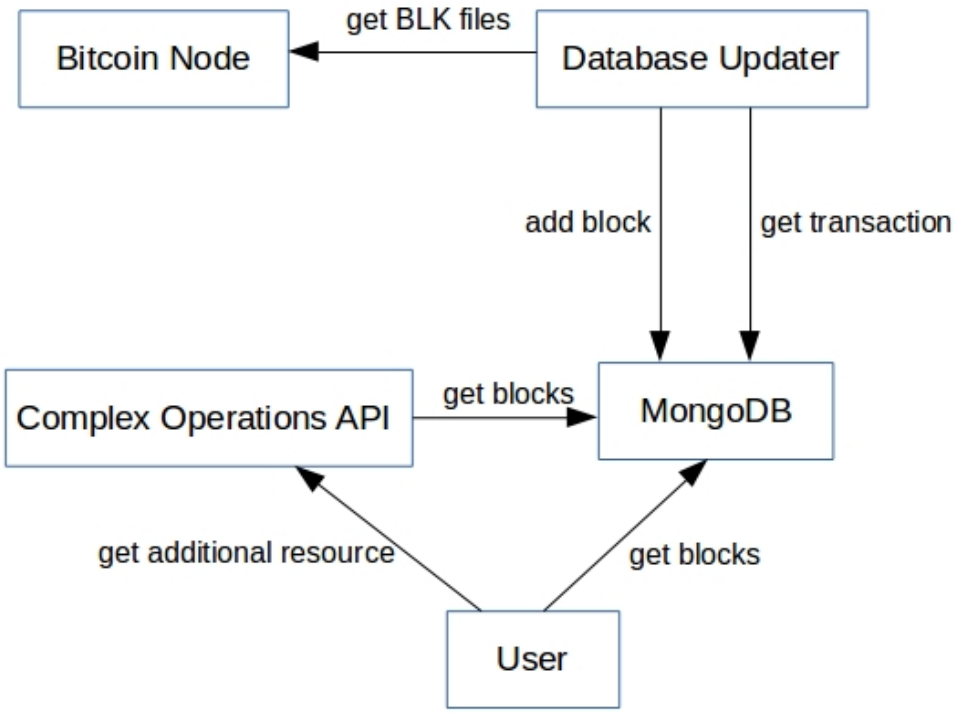

Figure 4.1 System components and their communication

4.4.1 Database Updater

This component will have responsibility for parsing raw bitcoin blocks collected from BLK files, discovering transaction sender address by transaction output referred in its input, adding blocks to database and administering the database, namely adding indexes in the best moment.

Although this components performance does not add to user wait time, it adds to time needed to get to the point at which most of the recent blocks are in the database. Because of large amount of data that needs to be processed and inserted to database, optimizations are necessary to finish initial data load in reasonable time.

The main bottleneck is discovering sender address, nowadays, one block holds about 2000 transactions, so to process one block with simplest design, database updater would have to ask the database about 2000 times for transaction for every processed block which would lead to very poor performance. Because it would require to much memory for database updater to remember every transaction it processed, it will remember some constant number of last processed transactions.

4.4.2 Complex Operations API

At the beginning I will explain why this component was requested to be implemented. Reason for this was the fact that when there is a group of people researching common field, there is big overlap in needed functionalities. When researcher gets to the point where he needs functionality that he doesn't have, he will implement it himself. The problem is that very often the same functionality was implemented by one of his peers, and this component is meant to solve this problem.

If we want users to use this API it needs to be not much harder to do than when doing it locally, and this will be crucial thing to think about during design.

Complex Operations API will be web service, which in response to its users request will return JSON data. At the main page will be located descriptions of implemented functionalities with URL's needed to access them. There will be one functionality implemented in this service which purpose will be to handle to the users a well working example of something similar to what they will be implementing.

4.5 Deployment

The main things that can be accomplished with well chosen deployment and packaging are simplicity of use and cross-platform possibilities which can positively influence adoption rate of this solution in perspective of both usage and further open source development.

For this project I will use docker containers, which will help in keeping components decoupled, and will make installation process not to require multiple complex steps.

Installation process that I want to achieve should consist only of installing full bitcoin node, docker, running one custom script and setting up simple configuration (database credentials, etc.).

All the containers will be connected to the same private network, so that communications between them will be as simple to achieve as possible.

Chapter 5

Implementation

5.1 Install.sh

I have decided to start with the installation script, because knowing its content will guide us by how containers are set up, how data flows between them, an will give us overview of whole system.

Install.sh starts with reading configuration from file "config.conf" in which following things are set up:

- Database credentials for administrator, and user with read-only privileges
- Port at which the database will be reachable
- Data directory of bitcoin node
- Directory at which we want to store our database data
- Transaction cache size

After configuration is loaded, network is being created, and building and starting of containers starts.

5.1.1 Database container setup

For database container I will show how configuration looks, so that readers without experience with docker will have chance to see it.

```
docker run -d \
    --name btc-blockchain-db \
    -e AUTH=yes \
    -e MONGODB_ADMIN_USER=$db_root_username \
    -e MONGODB_ADMIN_PASS=$db_root_password \
    -e MONGODB_APPLICATION_DATABASE=bitcoin \
    -p 0.0.0.0:$db_port:27017 \
    -v $database_dir:/data/db \
    --network=btcnet \
    aashreys/mongo-auth:latest
```

Database container is build on "aashreys/mongo-auth:latest" docker image, which lets us create MongoDB database with enabled authentication (by default in MongoDB authentication is not enabled). Admin user credentials are being set, but credentials of user with read-only privilege can't be set here, because before-mentioned image does not support such action.

There are couple other things in this configuration worth noting, namely "-v" flag creates volume for container and its purpose comes from the fact that docker containers should be possible to kill without any loss, so with volume set up, when we kill our database container and run it again, the database data will not be lost.

Another important thing is setting up common network for all the containers, here it is accomplished with flag "–network". With network configured, DNS is set up automatically and it is possible to refer other containers by their names. For instance running

```
ping btc-blockchain-db
```

from within other container connected to the same network will send messages correctly

5.1.2 Database updater container setup

For database updater docker image needs to be created first. To create docker image we need to write our configuration in file called "Dockerfile", than use command docker build to build image, and with image ready we can run it in the same way previously described image was run.

Let's go through dockerfile for database updater and see what it looks like:

```
FROM python:3

COPY requirements.txt ./
COPY ./src/ /src/
RUN pip install --no-cache-dir -r requirements.txt

# forked version from 22.03.2018
RUN pip install https://github.com/MarcinPieczka/python-bitcoin-blockchain-parser/arch

RUN mkdir /btc-blocks-data/

EXPOSE 27017

CMD [ "python", "-u", "/src/update_database.py" ]
```

Dockerfile starts with base image that is being used, here it is "python:3" image that contains as its name suggests python3. Next two lines copy needed files to the container, than libraries are installed, containers port is exposed for communication with database, and in the last line my application is being started.

Using docker makes it easy to create installation process repeatable and independent from environment, so that running software will have the same effect on every machine it is being ran (to some extend naturally).

5.1.3 Complex Operation API container setup

This container is based on "tiangolo/uwsgi-nginx-flask:python3.6" which have all the basic elements needed to run flask application already configured. The only non standard configuration that needed to be done was setting timeout in Nginx web server to larger value. By default timeout is 60 seconds, but because of type of operations that this application is made for this was not enough

5.2 Database updater

This container as stated before is responsible for database, and in more detail it is responsible for both putting data to it and for managing its configuration. In regards of managing configuration of database, this module takes responsibility for everything that could not be achieved in docker command, and that is:

- Creating user with privilege to read the database, but not to modify it. This account will be used by end user.
- Creating database indexes for block height, block timestamp and transaction hash

All this is being done in separate module responsible only for database - "mongo.py", other than caring for database configuration, it abstracts away connecting and loading data to the database. This module uses library "pymongo" in version 3.6 to connect to database.

Data insertion took the most effort to develop, and is the most complex part of the system.

All of bitcoin BLK files processing is done with "blockchain-parser" library, but not the official release, but version straight from their git repository. The reason for this is that changes in bitcoin are far more frequent than releases of this software, and some things ware not supported in official version. The version I am using does not support BIP-0173 (BIP stands for bitcoin improvement proposal) so discovering addresses from transactions using techniques described in it cannot be done.

Processing and loading data can be broken into these steps:

- Initial parsing of BLK files contained in bitcoin data directory, only headers of blocks are parsed, to get hash of block, and hash of previous block.
- Representation of blockchain is created, to get to know the order of blocks, and to know which blocks are in the main chain.
- One by one, starting from the block next after last inserted blocks are fully parsed, their transactions are cached, and input addresses are being discovered first by looking into cache, than if cache does not contain that transaction, the database is being asked.
- last step is inserting processed block into database.

Of course there is more complexity hidden inside those steps, like handling exceptions from blockchain-parser library. The reason there are exceptions while parsing bitcoin

blocks is that bitcoin transactions do not have to be fully valid. Many transactions exist with errors, and although they are of no interest for my system, they need to be recognized, and handled properly.

5.3 Complex Operation API

This component was implemented in Flask framework which was chosen for its simplicity which is well needed in application that people who very often don't have experience with web applications will extend with their code.

Main page is made purely in HTML and CSS, and the argument for simplicity that I have made holds here as well.

I have implemented two exemplary functionalities, one of them is really simple and will be a good way to show simplicity of Flask, implementation:

```
@app.route('/api/last_block')
def _last_block():
    return dumps(mongo.db.blocks.find().sort([('height', -1)])[0])
```

In this code first line specifies address at which service will be available, second one is declaration of function, and the last one is its implementation. This function connects to the database with previously created "mongo" object, finds last block in the database, converts it to JSON with "dumps" function and returns.

The second functionality is surely more complex. First of all lets notice that we can see bitcoin transactions as graph. Every transaction has its input and output, the addresses of inputs and outputs are vertices of the graph, and transactions are the edges between them. My second functionality as input takes two parameters which specify block range by hight, than creates graph out of transactions found in specified blocks using iGraph library, and at the end separates unconnected graphs and returns array of count of vertices in each graph.

The choice of this functionality was not random. Operations on graphs are very common when analyzing bitcoin transactions.

At the end I have to say that although I think that application by itself is easy to understand and in this regard it will be easy for users to add their code to it, the deployment with docker makes it not so easy to use as a developer. Biggest drawback of docker is relatively slow startup, so when we write some portion of code and we want to run it and see how things are going, we need to wait about 15 seconds (time measured on my personal computer) before process of re-installation completes.

Chapter 6

Functionality testing

6.1 Getting blocks by height

Getting bitcoin blocks by their height has been identified by me to be one of the most useful way to query blockchain, and its performance is of high priority.

6.1.1 testing on the same machine as database

consecutive blocks test

results:

blocks between height 0 - 99:	0.0035 s
blocks between height 100000 - 100099:	0.0065 s
blocks between height 200000 - 200099:	0.48 s
blocks between height 300000 - 300099:	1.7 s
blocks between height 400000 - 400099:	7.9 s

spread blocks test

results:

100 blocks spread between height 0 - 100000:	0.49 s
100 blocks spread between height 100000 - 200000:	1.2 s
100 blocks spread between height 200000 - 300000:	1.6 s
100 blocks spread between height 300000 - 400000:	3.5 s

6.1.2 testing with accessing database by network

consecutive blocks test

results:

blocks between height 0 - 99:	0.097 s
blocks between height 100000 - 100099:	0.15 s
blocks between height 200000 - 200099:	5.1 s
blocks between height 300000 - 300099:	13 s
blocks between height 400000 - 400099:	51 s

spread blocks test

results:
100 blocks spread between height 0 - 100000: 2.9 s
100 blocks spread between height 100000 - 200000: 5.7 s
100 blocks spread between height 200000 - 300000: 15 s
100 blocks spread between height 300000 - 400000: 42 s

6.1.3 Result interpretation

When we compare results for tests ran locally and over network, we can see about ten fold difference in time, which means that performance of database is not a bottleneck and there is no need for further optimizations.

Another thing we can see is that query time highly depends on height of blocks we are getting. The reason for this is the fact that because of gradual adoption of bitcoin the average number of transactions in blocks had grown over years.

6.2 Getting blocks by timestamp

Because querying by hight and by timestamp should not be different i will perform only one test.

result of run over network:
blocks between timestamp 2016-02-25 16:24 - 2016-02-26 07:23: 58 s

6.2.1 Result interpretation

Because range of time that was tested fully corresponds to getting blocks by hight with range of height being from 400000 to 400099 we can compare those results. There is no significant difference in performance between getting by height and by timestamp.

6.3 Complex operations API

In this component performance is not main goal, i will only run available services and see if results satisfying

6.3.1 Getting last block

service is located at http://base-address/api/last_block and calling it returns JSON data:


```
{
    "_id": {
        "$oid": "5ae122209043650001213f12"
    },
    "hash": "00000000000000000c0af3be64d9a335968225b35f9fb102d4962c0ff2558c2",
    "version": 536870912,
    "height": 496509,
    "prev_hash": "00000000000000000384595a7dc898a505d5c131dffa1c55431dc73cffe2355",
```

```
    "merkle_root": "9df11de2a46b352591f5980d63929c68d9af01bc2e494f4b350ae697809773ac"
    "timestamp": {
        "$date": 1511869304000
    },
    "n_tx": 2425,
    "size": 1079456,
    "bits": 402706678,
    .
    .
    .
}
```


6.3.2 Getting count of maximal connected graphs by height

service is requester with address
`http://base-address/api/count_separate_graphs?height_from=100000&height_to=100500`
and in this case it returns [1701,74,42,39,21,20,20,19,17,...] which means that biggest connected graph that can be made from transactions with height from 100000 to 100150 has 1701 vertices and so forth.

Chapter 7

Summary

In this work I have designed and implemented system providing easy access to bitcoin blockchain data. There is big problem of unnecessary work being done among bitcoin researchers, many of them would develop their own way of accessing blockchain data, others would use web services which usually have API call limits. To combat this problem there needed to be free, easy to use and maintain solution that provides access to bitcoin data. Nonetheless accessing data is not the only field on which researchers have problem which mutual code reuse, thats why platform that is ready to implement functionalities on needed to exist as well.

Success in development of this system if partial, because libraries that make parsing binary bitcoin data have problem with keeping up with changes in bitcoin, the data of newest bitcoin blocks is not fully available through my system.

Bibliography

[1] Block hashing algorithm, https://en.bitcoin.it/wiki/Block_hashing_algorithm

[2] Definition of cryptocurrency, https://en.oxforddictionaries.com/definition/cryptocurrency

[3] Bartosz Zychal, Analiza acucha tranzakcji w sieci Bitcoin https://en.oxforddictionaries.com/definition/cryptocurrency

[4] Thomsen, Sren Steffen; Knudsen, Lars Ramkilde, 2009, Cryptographic Hash Functions