Cryptocurrency mining strategies comparison

David Klemenc

Abstract

This document describes possible mining strategies for miners with slower hardware. It also provides a brief overview of the Bitcoin cryptocurency, mining and the technologies involved.

Introduction

Figure 1: `https://gitlab.klemenc.io/FRI/kripto`

A cryptocurrency is a digital asset designed to work as a medium of exchange that uses cryptography to secure its transactions, to control the creation of additional units, and to verify the transfer of assets. Cryptocurrencies use decentralized control as opposed to centralized electronic money and central banking systems. The world's first decentralized digital currency was introduced in 2008 and is called Bitcoin. In this paper I will first explain how a prototypical cryptocurrency works [1] and how to build one. Then I will analyze a miner probability of successfully minig a block (a miners hit rate) [2] and show a few strategies of improving the aforementioned probability. Lastly I will discuss possible implementation of the mentioned mining strategies [3].

[1] section: Cryptocurencies

[2] section: Can we ensure that a slower miner will have a hit rate > 0 ?

[3] section: Discussion

Cryptocurencies

Cryptocurencies are a form of decentralized trustless verification system based on digital signatures and hash functions.

A prototypical cryptocurency uses a communal ledger [4] to store all transactions made by users. A normal ledger stores transactions made by users, so they can calculate and redistribute founds on a fixed period rather then having to pay after each transactions. (For example a group of friends could write on a piece of paper how much each has spend for drinks and food every day and after each month they would calculate how much each is due. A possible problem to this approach is if one of the group does not show up at the end of a month - this can be solved by first collecting money and writing on the paper Alice receives 100 €, Bob receives 100€ ... and then using this to calculate how much each individual has at the end of the month). Cryptocurencies use a ledger similar to this with the following rules:

[4] ledger: a book or other collection of financial accounts

- broadcast transactions

- only signed transactions are valid
- no overspending is allowed
- trust the ledger with the most computational work put into it (Proof of work)

This means that verifying a transactions requires knowing the full history of transactions up to that point (this is computationally improved by validation chaning)

So we can look at Bitcoin as just a history of transactions.

Communal ledger

			LEDGER		
ID	FROM	WHAT	TO	AMOUNT	SIGNATURE
0	Alice	pays	Bob	100BC	$\mathcal{ALICE}0$
1	Alice	pays	Bob	100BC	$\mathcal{ALICE}1$
2	Bob	pays	Oliver	112BC	$\mathcal{BOB}2$

[5]

[5] Bitcoin internally uses ECDSA (Elliptic Curve Digital Signature Algorithm) for signing transactions

Proof of work [6]

[7] The proof-of-work involves scanning for a value (in bitcoin terminology this is referred to as a "nonce" [8]) that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, November 2008. URL `https://bitcoin.org/bitcoin.pdf`

[7] proof of work is the output of a miner

[8] The "nonce" in a bitcoin block is a 32-bit (4-byte) field whose value is set so that the hash of the block will contain a run of leading zeros. The rest of the fields may not be changed, as they have a defined meaning.

Ledger organization (Block-Chain)

In the Bitcoin protocol the ledger is split into blocks. Each block consist of a list of transactions together with a proof of work (it is only valid if it contains a proof of work) and it must also contain the proof of work of the previous block - thus creating a block-chain. If we want to change block $x, \in [0, n]$, where n is the current block - we need to compute the proof of work of all the blocks from x to n.

Block creation (mining)

Creating a proof of work is also called mining. To incentivize miners we give them a block reward - which is a special transaction for creating a proof of work. Block rewards do not contain a sender or a signature and add to the total amount of money available. From a miners perspective each block is similar to a miniature lottery. A miner tries to guess the correct input for the hash function so he gets

the desired output - whichever miner guesses first gets the reward. Statistically the fastest miner has the highest chance of guessing correctly thus getting the block reward.

For example, in Bitcoin the hashing algorithm is *double-SHA256* [9] ($SHA256^2$) and the predefined structure is a hash less or equal to a target value T. The success probability of finding a nonce n for a given message msg, such that $H = SHA256^2(msg||n)$ is less or equal to the target T is

$$Pr[H \leq T] = \frac{T}{2^{256}}$$

This will require a party attempting to find a proof of work to perform, on average, the following amount of computations

$$\frac{1}{Pr[H \leq T]} = \frac{2^{256}}{T}$$

[9] Krzysztof Okupski. Bitcoin developer reference. URL `https://github.com/minium/Bitcoin-Spec`

Block trust

Since an evil operator (Oliver) could potentially try to fool Alice by creating a fraudulent block. Oliver sends the fraudulent block to Alice (this block includes a payment from Alice to Oliver), but he does not broadcast this block to the rest of the network. To accomplish this, Oliver would have to find a valid proof of work before anybody else. Alice wold now know about Olivers block, but she would also hear conflicting block broadcasted from other miners. To keep fooling Alice Oliver would have to keep guessing the proof of work before other miners - which means that he would have to have more then 50% of the combined computational power - which is unlikely! If Oliver does not posses such computational power - sooner or later the competing block-chain will be longer and Alice will no longer trust Olivers block-chain. This means that a new block can not be trusted immediately - one should wait for a few more blocks before trusting!

Miner incentives

As discussed earlier upon providing a valid proof of work for a given block a miner gets a block reward. In the case of Bitcoin this reward decreases geometrically over time, this is why there is another way of incentivizing miners. You can add a transaction fee that is going to the miner - this means that a given miner is more likely to include your transaction into the block he is mining. [10]

[10] each Bitcoin block is limited to approximately 2400 transactions

Can we ensure that a slower miner will have a hit rate > 0 ?

The reference miner implementations for Bitconin [11] and some other

[11] Jeff Garzik. Pyminer. URL `https://github.com/jgarzik/pyminer`

cryptocurrencies, pick random numbers to try as the nonce , I will compare this method to picking sequential numbers and show that using the correct strategy sequentialy picking numbers yields better results.

To compare miner hit-rates I simulated a fast and a slow miner as follows:

```
const crypto = require('crypto');

module.exports = class Miner {

    constructor (baseText, strategy, iteration, name, hashZeroes) {
        this.baseText = baseText;
        this.strategy = strategy;
        this.iteration = iteration;
        this.name = name;
        this.hashZeroes = hashZeroes;
    }

    mine() {
        const hash = crypto.createHash('sha256');

        let textToTry = "";

        if (this.strategy === 'RANDOM') {
                textToTry = this.baseText + this.getRandomInt(Number.MAX_SAFE_INTEGER);
        } else {
                textToTry = this.baseText + this.iteration;
        }

        hash.update(textToTry);
        let ctext = hash.digest('hex');

        if (this.checkHash(ctext)) {
            return {
                hash: this.name,
                value: ctext,
                iteration: ++this.iteration,
                string: textToTry
            }
        } else {
            ++this.iteration;
            return false;
        }
    }

    checkHash(hash) {
        if (hash.substr(0, this.hashZeroes) === this.getComparisonString()) {
```

```
            return true
        }
        return false;
    }

    getComparisonString() {
        let str = "";
        for (let i = 0; i < this.hashZeroes; i++) {
            str += "0";
        }
        return str;
    }

    getRandomInt(max) {
        return Math.floor(Math.random() * Math.floor(max));
    }

}
```

By calling the Miner class with different arguments I create two miners. Both miners are called in a loop, the speed difference is simulated by calling the fast miner more times then the slow miner (in our simulation the fast miner is ten times faster than the slow miner).

```
let fastHasher = new Miner(randomMessage, program.fastminer, 0, 'fastHasher', hashZeroes);
let slowHasher = new Miner(randomMessage, program.slowminer, 0, 'slowHasher', hashZeroes);
```

Testing hits by both miners gives us the tables below. Each row represent a search for 100 hits (in this example we searched which number added to a random string will produce a hash beginning with "000").

```
npm test -- -z 3 -i 100 -t 8 -f RANDOM -s RANDOM -o result.txt
```

Meaning of various options:

- -z (number of leading zeroes)
- -i (number of iterations)
- -t (number of tests)
- -f (fast miner strategy, defaults to RANDOM)
- -s (slow miner strategy, defaults to SEQUENTIAL)
- -o (filename to write results to)

Results:

fast miner hits	fast miner searches	slow miner hits	slow miner searches
89	286064	11	3440
84	348266	16	5373
92	352617	8	2381
93	415154	7	2312
92	372840	8	1433
91	338755	9	3880
93	358425	7	1831
95	301759	5	865

Changing both strategies to sequential:

```
npm test -- -z 3 -i 100 -t 8 -f SEQUENTIAL -s SEQUENTIAL -o result.txt
```

Results:

fast miner hits	fast miner searches	slow miner hits	slow miner searches
100	429421	0	0
100	448040	0	0
100	416723	0	0
100	406854	0	0
100	395534	0	0
100	400507	0	0
100	416656	0	0
100	390311	0	0

As we can observe this is a loosing scenario for the slower miner, because both miners use a sequential search, the faster miner will try the same combinations as the slower one!

We can overcame this by prefixing all the slow-miner searches with a random string:

```
npm test -- -z 3 -i 100 -t 8 -f SEQUENTIAL -s SEQUENTIAL -p 11 -o result.txt
```

Here we used the option $-p$ 11 to prefix the slower miner with the number 11, so the slower miner will try the following numbers: $110, 111, 112, 113, 114, ..., 119, 1110, 1111, 1112, ...$

Results:

fast miner hits	fast miner searches	slow miner hits	slow miner searches
90	314658	10	2490
90	324937	10	3709
89	351935	11	2770
90	346650	10	2716
91	313221	9	2348
92	387160	8	5483
84	326716	16	4427
90	359529	10	3721

As we can see this enables the slower miner to use a sequential search strategy!

Why use a sequential search strategy?

To search 10.000.000 hashes using random number strategy our miner needs 22s on average, if we search using a sequential numbers strategy (simply by incrementing a counter each iteration) we need less then 17s on average.

methond	time (ms)	average time (ms)
sequential numbers	16601	
sequential numbers	16608	
sequential numbers	16625	
sequential numbers	16628	
		16615.5
random numbers	21731	
random numbers	21841	
random numbers	21835	
random numbers	22404	
		21952.8

Will searching a subspace decrease our chances of finding the correct hash input?

Prefixing our search numbers with some random number does not change our likelihood of finding the correct input to produce the desired number of zeroes. The possibility of finding the correct input depends only on the number of leading zeroes!

The SHA-256 algorithm produces a string, each character of that string represents a hexadecimal number - which means that each character has 16 possible values.

If x is our hash input than the chance (P) of it producing the correct number of leading zeroes (n) is:

$$P = \frac{1}{16^n}$$

(This assumes that the hashing function used gives us an uniform distribution over integers)

If we solve for $n = 3$ we get: $P = \frac{1}{4096}$

Since

$$E(x) = \frac{1}{P}$$

we should try 4096 combinations on average, before getting a hash beginning with 3 zeroes!

Discussion

Using sequential numbers with a random prefix is a possible strategy when mining a cryptocurrency. Since generating sequential numbers is faster then generating pseudo-random numbers - we get a speed increase - which means we can search for more hashes - improving our chances of finding the correct hash.

In the case of Bitcoin there are some additional properties that our random prefix could have to further improve our hashing speed. Since the hashing algorithm in Bitcoin hashes 512 bit blocks and the average block size is $1MB$ we process:

$$\frac{10^6 * 8}{512} \approx 15000 \text{ blocks}$$

We could use our prefix to fill the last block to full (this is done automatically by the SHA 256 algorithm), now we can compute all the static blocks and save the intermediate result (we compute ≈ 15000 blocks only once), and then use the stored intermediate result to compute our final hash (we are only changing the last block).

Appendices

Miner simulator:

The git repository for the simulator can be found here:
`https://gitlab.klemenc.io/FRI/kripto`

test.js:

```
const program = require('commander');
const ss = require('simple-statistics');
const pjson = require('./package.json');
const Miner = require('./Miner');
const Writer = require('./Writer');

program
  .version(pjson.version)
  .option('-z, --zeroes [number]', 'Number of leading zeroes', 'parseInt')
  .option('-i, --iterations [number]', 'Number of iterations', 'parseInt')
  .option('-t, --tests [number]', 'Number of tests', 'parseInt')
  .option('-o, --output [file]', 'Filename to write to')
  .option('-p, --padding [value]', 'String to prepend')
  .option('-s, --slowminer <strategy>',
    'Slow miner strategy, defaults to SEQUENTIAL',
    /^(SEQUENTIAL|RANDOM)$/i, 'SEQUENTIAL')
  .option('-f, --fastminer <strategy>',
    'Fast miner strategy, defaults to RANDOM',
    /^(SEQUENTIAL|RANDOM)$/i, 'RANDOM')
  .parse(process.argv);

console.log('Miner simulation running with:');

console.log(' - slowminer strategy: %j', program.slowminer);
console.log(' - fastminer strategy: %j', program.fastminer);

const hashZeroes = (program.zeroes) ? parseInt(program.zeroes) : 3;
const numberOfIttertations = (program.iterations) ? parseInt(program.iterations) : 100;
const numberOfTests = (program.tests) ? parseInt(program.tests) : 6;
const slowMinerStrategy = (program.slowminer) ? parseInt(program.slowminer) : 'SEQUENTIAL';
const fastMinerStrategy = (program.fastminer) ? parseInt(program.itterations) : 'RANDOM';
const prependString = (program.padding) ? program.padding + '': '';

console.log(' - leading zeroes (difficulty): %d', hashZeroes);
console.log(' - itterations: %d', numberOfIttertations);

// simulate some document - block that we have to mine
function getRandomString() {
    var text = "";
    var possible = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    for (var i = 0; i < 15; i++) {
        text += possible.charAt(Math.floor(Math.random() * possible.length));
```

```
    }

    return text;
}

function getRandomInt(max) {
    return Math.floor(Math.random() * Math.floor(max));
}

function searchHash() {
    let foundHash = false;
    let randomMessage = getRandomString();
    let fastHasher = new Miner(randomMessage, program.fastminer, 0, 'fastHasher', hashZeroes);
    let slowHasher = new Miner(randomMessage + prependString, program.slowminer, 0, 'slowHasher', hashZero

    while (!foundHash) {

        if (getRandomInt(2)) {
            for (let i = 0; i < 10; i++) {
                foundHash = fastHasher.mine();
                if (foundHash) {
                    break;
                }
            }
            if (!foundHash) {
                foundHash = slowHasher.mine();
            }
        } else {
            foundHash = slowHasher.mine();
            if (!foundHash) {
                for (let i = 0; i < 10; i++) {
                    foundHash = fastHasher.mine();
                    if (foundHash) {
                        break;
                    }
                }
            }
        }
    }

    return foundHash;
}

let results = [];
```

```
let distribution = {};
let slowHits = 0;
let slowSearched = 0;
let fastHits = 0;
let fastSearched = 0;

let latexText = '';

function resetTest() {
    slowHits = 0;
    slowSearched = 0;
    fastHits = 0;
    fastSearched = 0;
}

for (var j = 0; j < numberOfTests; j++) {
    for (var i = 0; i < numberOfIttertations; i++) {
        let result = searchHash();
        results.push(result.iteration);
        let lastHashChar = result.value.substr(result.value.length - 1);

        if (distribution[lastHashChar] === undefined) {
            distribution[lastHashChar] = 1;
        } else {
            distribution[lastHashChar] += 1;
        }
        if (result.hash == "fastHasher") {
            fastHits++;
            fastSearched += result.iteration;
        } else {
            slowHits++;
            slowSearched += result.iteration;
        }
    }

    latexText += fastHits + " & " + fastSearched +  " & " + slowHits + " & " + slowSearched + '\\\\\n';
    resetTest()
}

console.log('\n Results: \n');

let sum = results.reduce(function(a, b) { return a + b; });
let avg = sum / results.length;
console.log(' * average: %j', avg);
```

```
console.log(' * distribution: %j', distribution);
// calculating tTest to check for possible mistakes
let distVals = Object.values(distribution);
let distSum = distVals.reduce(function(a, b) { return a + b; });
let distAvg = distSum / distVals.length;
let tTest = ss.tTest(distVals, (numberOfTests * numberOfIttertations) / 16);
console.log(' * tTest of distribution: %j', tTest);

if (program.output) {
    const writer = new Writer(program.output, latexText);
    writer.write();
}
```

Miner.js:

```
const crypto = require('crypto');

module.exports = class Miner {

    constructor (baseText, strategy, iteration, name, hashZeroes) {
        this.baseText = baseText;
        this.strategy = strategy;
        this.iteration = iteration;
        this.name = name;
        this.hashZeroes = hashZeroes;
    }

    mine() {
        const hash = crypto.createHash('sha256');

        let textToTry = "";

        if (this.strategy === 'RANDOM') {
                textToTry = this.baseText + this.getRandomInt(Number.MAX_SAFE_INTEGER);
        } else {
                textToTry = this.baseText + this.iteration;
        }

        hash.update(textToTry);
        let ctext = hash.digest('hex');

        if (this.checkHash(ctext)) {
            return {
                hash: this.name,
                value: ctext,
                iteration: ++this.iteration,
                string: textToTry
            }
        } else {
            ++this.iteration;
            return false;
        }
    }

    checkHash(hash) {
        if (hash.substr(0, this.hashZeroes) === this.getComparisonString()) {
            return true
        }
```

```
        return false;
    }

    getComparisonString() {
        let str = "";
        for (let i = 0; i < this.hashZeroes; i++) {
            str += "0";
        }
        return str;
    }

    getRandomInt(max) {
        return Math.floor(Math.random() * Math.floor(max));
    }

}
```

Nodejs is required, to install dependencies run:

```
npm install
```

References

Jeff Garzik. Pyminer. URL `https://github.com/jgarzik/pyminer`.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, November 2008. URL `https://bitcoin.org/bitcoin.pdf`.

Krzysztof Okupski. Bitcoin developer reference. URL `https://github.com/minium/Bitcoin-Spec`.