Universiteit Leiden

Opleiding Informatica

Mining Bitcoins with

Natural Computing Algorithms

Name: Niels Samwel
Studentnr: 1020919

Date: 20/08/2014

1st supervisor: Dr. M.T.M Emmerich
2nd supervisor: Prof. Dr. T.H.W Bäck

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Lately bitcoins have become more popular as a monetary unit. The ease and the anonymity for online payments do not go unnoticed by many people and Bitcoin payments getting more and more accepted every day. Only the way bitcoins are minted differs greatly from the way ordinary coins are minted. To create bitcoins you have to find a hashed value lower then a certain target. This thesis looks at this as a black box optimization problem. For these problems evolutionary algorithms have been proven to work good. This study uses metaheuristics, like simulated annealing and two types of genetic algorithms to try and find that value quicker than random search.

Contents

1 Introduction **3**

2 Bitcoins **4**
- 2.1 What are bitcoins? 4
- 2.2 Mining Bitcoins 5
- 2.3 SHA-256 6
- 2.4 Related Work 6

3 Methods **7**
- 3.1 Simulated Annealing 7
- 3.2 Standard Genetic Algorithm 9
- 3.3 Niching Genetic Algorithm 10

4 Results **12**
- 4.1 Simulated Annealing 12
- 4.2 Genetic Algorithm 14
- 4.3 Niching Genetic Algorithm 15

5 Conclusion **16**
- 5.1 Discussion 16
- 5.2 Future work 17

1 Introduction

Bitcoins [7] are gaining in popularity lately. There are many reasons for that, and one of them is that you can mine bitcoins yourself. But more on that later. First we need to explain what bitcoins are. A bitcoin is a digital monetairy unit. Which means that they can be used to pay with. Normally we do that with coins like the Euro where we exchange them or let a bank do that for us. The bank does that also digitally but in a different way. When money is deposited in the bank, the bank is trusted so that it does not get stolen and the payments made are done correctly.

With bitcoins that is different. There is no central bank or government institution that we have to trust to keep our money safe. So a bitcoin is no more then some bits on a computer and they can be split up so people don't have to make transactions with them as a multiple of one. With regular money when a digital transaction is done we trust the bank that they do it correctly. But because there is no central institute looking over bitcoins there must be another way. This is done using hashes. Computers have to try and get a value out of the hash lower than a certain target, but there is only al small chance to find it. That is why a lot of computers have to try it at the same time. To get people to spend their computing time on it there is a reward if that value is found. The reward is a block of bitcoins. Because of that the first research question is: *How can mining bitcoins be viewed as a black box optimization problem?*

Mining bitcoins is conventionally done by taking the smallest number a starting value and incrementing each try a lower value is not found, or by taking random values. There is a study about a theorem called the no free lunch theorem [11] that suggests that it is not possible to solve a blackbox optimization problem for which is nothing known about the function in a faster way. But there is also a study [3] that suggests that it is possible to speed up the process of solving the optimization problem if you have specific information about the problem like if the time to run the function is polynomial instead of exponenial. Therefore my second research question is: *Is it possible to solve a block of bitcoins faster than random search using evolutionary algorithms?*

2 Bitcoins

2.1 What are bitcoins?

The old fashioned transactions rely on trusting financial institutions. The weakness of that system is that transactions might be reversed. To ensure the transactions are done correctly financial institutions have to work together. This will increase the cost of transactions, making small transactions less viable. These costs can be avoided if there is no need of those trusted third parties to oversee the transactions. Before bitcoins were invented there was no system like that. Bitcoin is a payment system based on cryptography, where cryptographic proof is used instead of trust to verify the transactions. Reversing transactions is still possible but impossible if the majority of the computational power is not working together to attack the network.

A bitcoin is a chain of digital signatures. If someone transfers a coin he or she digitally signs it with a hash of the previous transaction and the public key of the next owner. The recipient can verify the signatures of the chain to make sure it is legitimate. To make sure that a coin is not spent twice all transactions are announced to the whole network. Timestamps are used to make sure that the first announced transaction is the one that is carried out. The majority of the network has to agree on which transactions is recieved first.

To use those timestamps a proof of work system is needed. For bitcoins that is a value when hashed with the SHA-256 algorithm starting with a number of zeros. A block is solved by incrementing a nonce until such a value is found. A block can not be changed unless all the work is redone for all the blocks that come after it.

The network works in a way that a new transaction is broadcasted to the whole network. The nodes try to find a proof of work for the block with the new transactions in it. If a node finds a proof of work, it broadcasts it to all nodes. The other nodes only accept the block if all the transactions in it can be verified. If the block is accepted the nodes will continue the chain with the latest hash. The longest chain is always correct and other nodes will always work on the longest chain.

As stated before with bitcoins there is no central authority that governs the distribution of the bitcoins. Because of that, bitcoins are rewarded to the node that solves a block. It is rewarded as the first transaction of the new block. This way bitcoins are steadily distributed. It also works as an encouragement to process transactions. Everytime a certain amount of bitcoins is rewarded the amount that is rewarded is lowered. On top of that there are also transaction fees that are rewarded to the node that solves the block. So when all bitcoins are handed out, there should still be enough motivation for nodes to keep solving blocks and verifying transactions.

2.2 Mining Bitcoins

The term bitcoin mining refers to scanning for a value that begins with zeroes when it is hashed with the SHA-256 algorithm. The difficulty and the target number of zeroes are set so that a new block is solved every ten minutes. After a block with the new transactions in it is solved and a proof of work is obtained, the block cannot be changed unless the work is done again. When someone attempts to change a block in the middle of the block chain all the work of the blocks after the changed block has to be done again as well. Because the network always considers the longest chain the correct one it makes it nearly impossible to do that. The more nodes that are in the network increases the speed blocks are solved so a node has smaller chances to solve a block which makes it even harder to attempt fraud. So the only way to spend your own transactions again is by having the majority of the computational power of the network so you have the longest chain. But if someone has that much computational power it is probably more profitable to solve blocks the legal way and earn bitcoins by doing so instead of trying to fraud the network and undoing its previously done transactions.

When bitcoins started, not many people knew about it and not many people were mining bitcoins. The difficulty was set to a low value and the target amount of zeroes was not high. So nodes mining had a good chance of solving a block once in a while. Even though the value wasn't that high at that time people still did it. But when bitcoins gained in popularity, the amount of people or nodes mining also increased. Because more bitcoins were distributed the number of target zeroes also increased. And to keep the solve rate at a steady one block per ten minutes the difficulty keeps increasing. This makes it these days nearly impossible for a single node to solve a block.

Because the main reason people are mining bitcoins is the reward they get after solving a block. So they started working together in a so called mining pool. Mining pools are in different sizes and the network stays safe as long as a mining pool does not have the majority of the computational power. If someone in a mining pool solves a block the reward is shared. There are different ways mining pools share the rewards but the most common way is that people in the pool get rewarded proportional to the amount of computational power they have spent on that block. This way people still have a fair chance to make money while mining bitcoins.

2.3 SHA-256

SHA stands for secure hash algorithm. SHA-256 is part of the SHA-2 set of cryptographic hash functions which are designed by the NSA. Such hash functions can be used to verify the authenticity of data because when some data is hashed more then once the outcome will be the same every time. Another reason why these hash functions are used is because it is nearly impossible to reverse the algorithm and calculate the original input from an output value. This is the reason why this algorithm is used with bitcoins, even though it has not been specifically designed for this application.

2.4 Related Work

Work related to finding bitcoins quicker or solve a block in another way than enumeration is not easy to find. Or whether or not solving the problem is best done with enumeration does not exist. A reason for that may be that if someone found a way to find a proof of work and by that to make money the person is not likely to publish his work and thereby losing income. Work that can be found is about whether or not bitcoin is a good currency or not[1]. Other parts of the bitcoin subject that are researched is about the security[4] of the currency or whether it is really as anonymous[8] as it is led to believe. There is no point in going further into details about these aspects, as the topic of this study is focusing on another aspect of bitcoins.

3 Methods

Because mining bitcoins is a lucrative business this study tries to find out if it is possible to not mine bitcoins the 'hard way' but the 'smart way'. Where instead of using enumeration or random search this study looks at simulated annealing and genetic algorithms to see if they are viable options. But to do that some code was needed that is able to do the hash with the required inputs [6]. This code is able to do a hash with the inputs that were used when the first block was solved. This information is used throughout all the experiments except for the nonce which is changed every hash. The code has been changed so that a different nonce can be used as an input and the hashed value is returned as an output. But instead of returning a 256-bit character array the twenty eight most significant bits are stored in an unsigned integer and returned because it is easier to work with.

$$SHA\text{-}256(SHA\text{-}256(x)) \rightarrow min|x \in S, S = \{0,1\}^{32} \tag{1}$$

3.1 Simulated Annealing

The first algorithm that this this study looks at is simulated annealing[10], that is why here the research question is: *Is it possible to solve a block faster than random search by using simulated annealing?*

First simulated annealing needs to be explained. Simulated annealing is a probabilistic heuristic optimization algorithm that is used to find an optimum of a function, as can be seen in equation 1.

Simulated annealing is based on the annealing process from metallurgy. Annealing is used to create a metal object where to molecules from the metal are aligned perfectly so the metal does not have any flaws after the process has finished. To do this the metal object is warmed up to its meltingpoint and cooled down slowly so the energy levels in the molecules stay equal during the process. This ensures the molecules get aligned and a minimal energy state is found, this also makes the metal object stronger.

In computer science this process is mimicked to solve optimization problems. Usually a random value is used to start with that needs to be optimized. An evaluation function is also needed to determine how good or bad the value

is. Next a neighboring value is generated and evaluated. If it has a better evaluation that the old value it is used as the new value, if not there is a probabilistic function depending on the temperature that decides whether or not the value is still accepted as the new value, this is the annealing part. This takes place in a while loop which stops if a value is found which has an evaluation better than a certain target or a maximum number of iterations are done. At the end of the while loop the temperature is lowered so the neighborhood in which worse evaluations are also accepted gets smaller over time. This is not done every iteration so more evaluations can be done with the same temperature.

For the bitcoin problem at the start of the algorithm a random nonce is generated and evaluated by the SHA-256 as seen in the pseudo code below. The nonce and the evaluation are both recorded as the current best values. Next a while loop starts which stops when a maximum number of computations have been done or when the evaluation value has fallen below a certain target value. As stated before at the end of the loop the temperature is lowered. In the while loop is a for loop, which does a hundred thousand iterations. Each iteration a new nonce is generated based on a mutation from the best one found. The mutation randomly flips bits on a set mutation rate. The newly generated nonce is then evaluated and compared to the best evaluation currently found. If it is better it stored with the nonce as the best found nonce and evaluation so far. When the evaluation is not better than the best one a probability is calculated. The value between zero and one is compared to the exponential function of the current evaluation minus the best evaluation divided by the temperature. If that value is higher than the randomly generated probability it is accepted as the new best nonce and evaluation.

Pseudo code

```
1  nonce := rand(); e := E(nonce)
2  bestnonce := nonce; ebest := e;
3  k := 0
4  while k < maxcomp and e > target do
5    for i := 0 to 100000 do
6      nonce := mutation(bestnonce)
7      e := E(nonce)
8      k := k + 1
9      if e < ebest
10       ebest := e; bestnonce := nonce
```

```
11       else if e != ebest and rand() / RAND_MAX <
12           exp((e - ebest) / T)
13         ebest := e; bestnonce := nonce
14     end for
15     T := 0.95*T
16   end while
17   return bestnonce
```

3.2 Standard Genetic Algorithm

The second algorithm that this study looks at are genetic algorithms[2], because of that the research question here is: *Is it possible to solve a block faster than random search by using a genetic algorithm?*

In a genetic algorithm a population is used to solve optimization problems. The algorithm starts from a randomly generated population. Each individual from the has to be evaluated by an evaluation function to decide the fitness of a possible solution. When the starting population is initialized a loop is started, each iteration in that loop is also called a generation of the population. In that loop is another loop that ranges from zero to the population size to generated new offspring. To generate offspring first a parent is selected from the population then it is detemined whether or not a crossover is done. If so a second parent is selected and a crossover is done. If not the first parent is used. Next a mutation is done on the newly generated offspring or the first parent based on a set mutation rate. When the offpspring population is created it is evaluated and merged with the old population, where the individuals with the best fitness from old population and the new population are kept. Now a generation is finished and a new generation is started until a maximum number of evaluations are done.

When this is applied to the blackbox optimization problem of bitcoins, this is how it is implemented. First a population is randomly initialized which are then evaluated. After each individual in the population has a fitness based on the outcome of the SHA-256 algorithm, they are ranked from best to worst. Then a while loop is started which loops until a maximum number of evaluations are done. Next a for loop starts which ranges from zero to the population size. In that loop the first parent is selected by randomly selecting two individuals from the population and taking the one with the highest rank, then a probability is calculated and compared to a crossover rate to determine whether or not a crossover takes place. If a crossover

takes place a second parent is selected the same way as the first parent. The crossover is done by calculating a probability for each of the 32 bits in the nonces in both parents to decide if a bit comes from the first or the second parent in the new nonce. If there is no crossover the first parent is used as the new nonce. Next a mutation function is used on the new nonce. The mutation function calculates a probability for each bit and compares in with the mutation rate. If the calculated probability is high enough the bit is flipped, if not it is left as it is. This is done for each individual in the offspring. Then the population is evaluated, ranked and merged where the best individuals from the offspring and old population survive into the next generation.

Pseudo code

```
1  initialize(P)
2  evaluate(P)
3  rank(P)
4  k := 0
5  while k < maxcomp do
6    for i := 0 to pop_size do
7      p1 := select_p(P)
8      if rand() / RAND_MAX < pc
9        p2 := select_p(P)
10       p_new[i] = crossover(p1, p2)
11     else
12       p_new[i] = p1
13     p_new[i] := mutation(p_new[i])
14   end for
15   evaluate(P)
16   rank(P)
17   merge(P)
18   k := k + pop_size
19 end while
```

3.3 Niching Genetic Algorithm

When using a genetic algorithm sometimes the population converts and the diversity in the population decreases, that is why a genetic algorithm with niching[5] is implemented. The research question here is: *Is it possible to solve a block faster than random search by using a genetic algorithm with*

niching?

When the diversity of the population decreases the landscape that is being searched on is getting smaller. That makes it possible to get stuck on a local optimum instead of finding better local optima or even the global optimum. Niching tries to prevent that from happening where the idea is to share the fitness of the population. This is done by decreasing the fitness of individuals when they are similar. To do this a sharing function is needed, it looks as follows:

$$sh(d_{i,j}) = \begin{cases} 1 - (\frac{d_{i,j}}{\rho}) & \text{if } d_{i,j} < \rho \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Where $d_{i,j}$ is the distance between individuals i and j. And where ρ is the maximum distance between two individuals. With this sharing function the niching count can be calculated with the following function:

$$m_i = \sum_{j=1}^{N} sh(d_{i,j}) \tag{3}$$

Next is the only function that differs slightly from [9] because it is for a maximization problem and this study looks at a minimization problem. The function to calculate the shared fitness is:

$$f_i^{sh} = f_i * m_i \tag{4}$$

Where f_i is the old fitness. When this is done for each individual the population is ranked again so it can be used in the next generation with parent selection.

This is implemented in the standard genetic algorithm as the last step in a generation. The functions are all implemented as can they are listed above and can be viewed in the pseudo code below, except for the distance between two individuals. For that the hamming distance is used, which counts the bits that are different between the two nonces. With this implemented when two nonces have similar bitstrings they have a small distance between them and their fitness is increased so it is less likely to have a many similar individuals in the population.

Pseudo code

The following code goes between line 17 and 18 of the pseudo code of the standard genetic algorithm.

```
1 for int i := 0 to pop_size do
2   m := 0
3    for int j := 0 to pop_size do
4      m += sharing_function(distance(p[i], p[j]))
5    end for
6    f_sh[i] = (i + 1) * m
7 end for
8 rank(P, f_sh)
```

4 Results

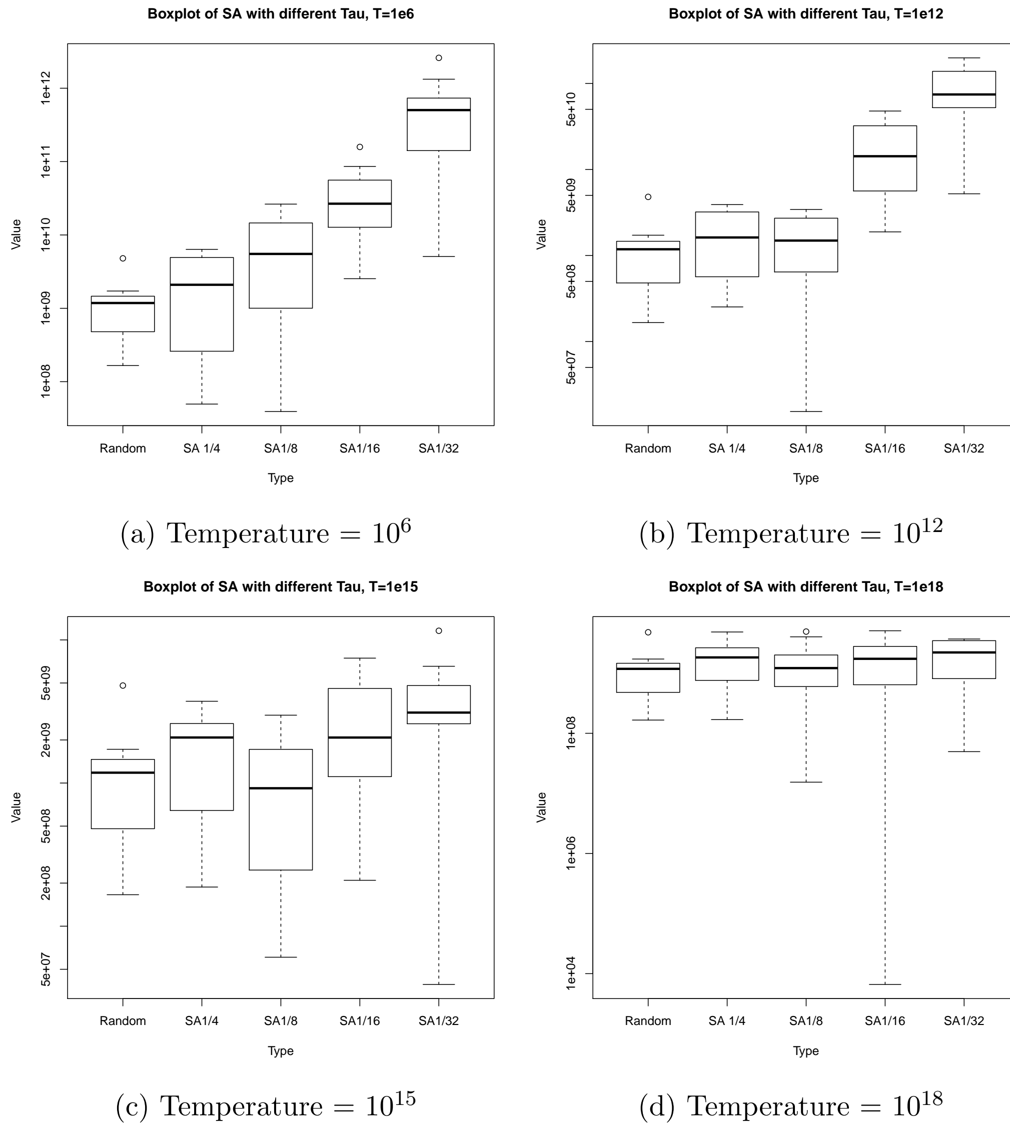
For this study it was important that the SHA-256 function in each algorithm got the same amount of CPU time. That is why a maximum amount of hashes was set to fifty million, which takes about ten minutes to run on the CPU that was used. This way the same amount of hashes are done so the different algorithms can be compared.

4.1 Simulated Annealing

For simulated annealing this study looks at four different temperatures as can be seen in figure 1. Usually with simulated annealing the temperature is lowered every set amount of iterations but for this study the temperature was kept the same to see what kind of landscape was searched on. In figure 1 there are four boxplots, on the vertical axis of the boxplots are the hashed values and on the horizontal axis is displayed if it are random nonces that were tried or different values for the parameter tau which is the mutation rate in the algorithm. In each boxplot ten runs of fifty million hashes are done.

The mutation rate in these figures was set to four different values. A higher mutation rate means that the mutated nonce has a higher chance to change more from the old nonce where a mutation rate of 0.5 would basically generate a new random nonce. In figure 1.a and 1.b which are the figures with the lowest temperatures of the four figures it is noticeable that the higher the mutation rate the better the results are. It also shows that a higher mutation rate results in hashed values closer to random which is not strange because the higher the mutation rate the closer the mutated nonce is to a randomly generated nonce.

When looking at figure 1.c and 1.d, the boxplots with the highest temperatures, one can observe that different mutation rates don't really make a difference. It is likely that this has something to do with the high temperatures. Because when using simulated annealing high temperatures mean that candidate solutions a lot worse than the best one found can be accepted as new best solution. This makes it closer to random search and that is why the solutions are similar to a random search. There is one solution found on figure 1.d with a mutation rate of $\frac{1}{16}$ which is the nonce of the first bitcoin but that was more likely obtained by luck than by the algorithm.



(a) Temperature = 10^6

(b) Temperature = 10^{12}

(c) Temperature = 10^{15}

(d) Temperature = 10^{18}

Figure 1: Simulated Annealing with different temperatures and tau's

4.2 Genetic Algorithm

When using a genetic algorithm this study looks at four different population sizes as can be seen in figure 2. Located on the vertical axis are the hashed values and on the horizontal axis is shown how many generations are done with the size of the population or if it was a random search. The population sizes and generations are different but each run did fifty million hashes and

each boxplot is ten runs. The crossover rate is set to one and the mutation rate is set to $\frac{1}{32}$. From the figure can be observed that the different population sizes do not really differ much from eachother. The results from the genetic algorithm are also more or less equal to random search other than the two low results from a population of ten and one thousand, but that might as well have been plain luck. So a genetic algorithm may give similar results than random search, but random search is definately a lot faster.

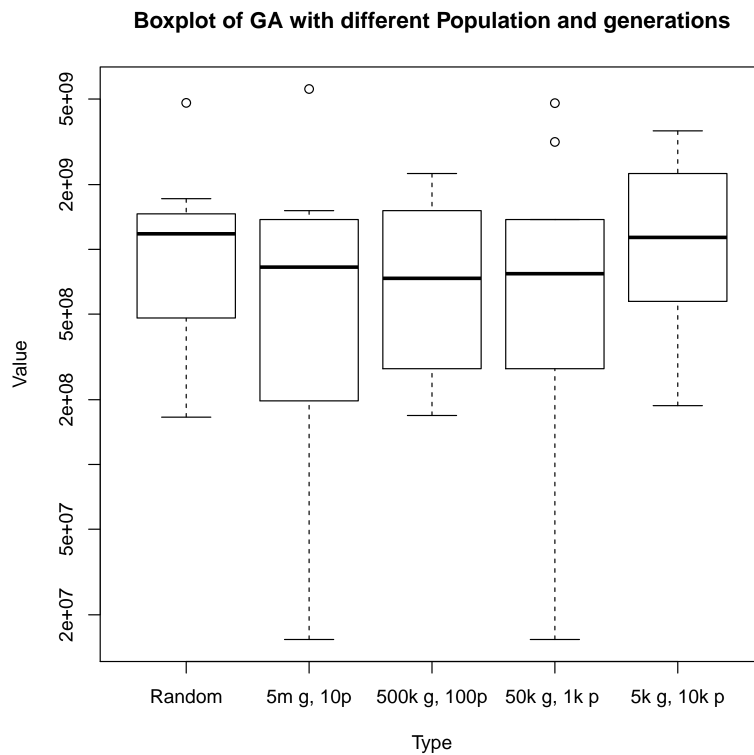


Figure 2: Genetic algorithm with different population sizes

4.3 Niching Genetic Algorithm

When looking at figure 3 which shows results from the genetic algorithm with niching the axis depict the same as in figure 2. Each boxplot is again made up of ten runs of fifty million hashes. But with niching only two population sizes are looked at. These are the two smallest populations and it is because calculating the niching count takes such a long time with larger populations

that it would definately not be faster than random search. The crossover and mutation rate are also the same as before. Here the results are again similar to random search and because niching also takes time it is again a lot slower than random search.

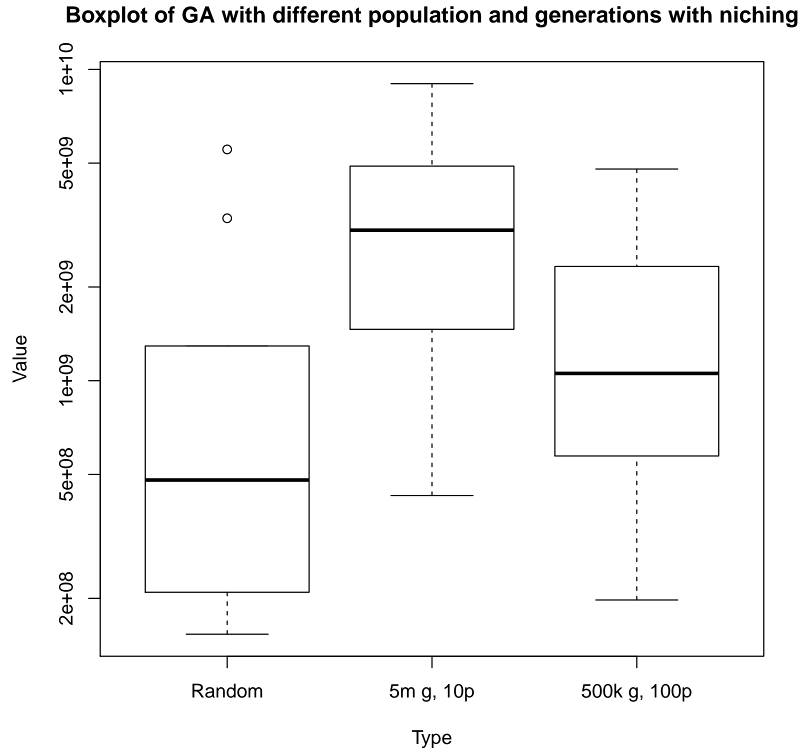


Figure 3: Niching genetic algorithm with different population sizes

5 Conclusion

5.1 Discussion

Looking back at at the results, it is probably safe to say that what is tried in this study is not a better way to find a solution than random search. Even though there are some results that are better than random there is a good chance it has more to do with luck than with the algorithm doing its job. This answers the last three questions from this study, there is no way using simulated annealing or a genetic algorithm to solve the black box

optimization problem that was looked into. That is a good sign for the SHA-256 algorithm, if it would have worked the hashing would not be as safe as is expected and therefore making bitcoins unsafe in general. The answer to the first research question, is it possible to view this problem as a black box optimization problem is yes, it is possible to do his, but as comes with the answers of the other two research questions it didn't do any good. To make a long story short, if you want to mine bitcoins, do it the conventional way by using random search or enumeration. In this study random search is chosen above enumeration, this makes duplicates possble. But since the search space is very large this has a small chance of happening.

5.2 Future work

The fact that this study shows that there will be probably no standard meta-heuristics for mining bitcoins more efficiently than random search. For the genetic algorithm someting that can be researched is trying to optimize the parameters of the algorithm, a lot of parameters are possible and testing them takes a lot of time. Another way to look at this problem is to not look at it as a black box optimization problem, but look more into the structure SHA-256. Doing this requires strong knowledge about cryptography. Or a gray box optimization algorithm an be created where only a small amount of knowledge about SHA-256 is needed. But even with that it is still unlikely due to the hasing algorithm. If it is a safe and secure hashing algorithm, finding a better algorithm than random search may be not possible.

References

[1] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better—how to make bitcoin a better currency. In *Financial Cryptography and Data Security*, pages 399–414. Springer, 2012.

[2] Lawrence Davis et al. *Handbook of genetic algorithms.* Van Nostrand Reinhold New York, 1991.

[3] Stefan Droste, Thomas Jansen, and Ingo Wegener. Perhaps not a free lunch but at least a free appetizer. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1(1):833–839, 1998.

[4] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *arXiv preprint arXiv:1311.0243*, 2013.

[5] David E Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[6] Joseph Matheney. Code of hash function. http://pastebin.com/EXDsRbYH/, March 2011.

[7] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *http://bitcoin.org/bitcoin.pdf*, 2008.

[8] Fergal Reid and Martin Harrigan. *An analysis of anonymity in the bitcoin system*. Springer, 2013.

[9] Ofer Michael Shir. *Niching in derandomized evolution strategies and its applications in quantum control*. Natural Computing Group, LIACS, Faculty of Science, Leiden University, 2008.

[10] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated annealing*. Springer, 1987.

[11] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

Appendix

SHA-256 as an optimization function

The following code can be found here [6] and was edited so the code can be used as an optimization function. It is a class called Hashfunction with a function calles hash that computes the hash. To use it you have to call the hash function with the nonce as the parameter. The nonce is an unsigned int. The function will return the 28 most significant bits of the output of the function in a long variable. The header file can be found below.

hashblock.cc

```
/*
```

```
#Copyright (c) 2011, Joseph Matheney
#All rights reserved.

#Redistribution and use in source and binary forms, with
#or without modification, are permitted provided that the
#following conditions are met:

#     Redistributions of source code must retain the above
#     copyright notice, this list of conditions and the following
#     disclaimer.
#     Redistributions in binary form must reproduce the above
#     copyright notice, this list of conditions and the following
#     disclaimer in the documentation and/or other materials
#     provided with the distribution.

#THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
#CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
#WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
#WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
#PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
#THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY
$DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
#CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
#PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
#USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
#CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
#STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
#ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
#ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

*/
#include <openssl/sha.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <limits>
#include <sstream>
#include "hashblock.h"

 using namespace std;
```

```c
// this is the block header, it is 80 bytes long (steal this code)
typedef struct block_header {
        unsigned int    version;
        // dont let the "char" fool you, this is binary data
        // not the human readable version
        unsigned char   prev_block[32];
        unsigned char   merkle_root[32];
        unsigned int    timestamp;
        unsigned int    bits;
        unsigned int    nonce;
} block_header;


// we need a helper function to convert hex to binary, this
// function is unsafe and slow, but very readable
// (write something better)
void hex2bin(unsigned char* dest, unsigned char* src)
{
        unsigned char bin;
        int c, pos;
        char buf[3];

        pos=0;
        c=0;
        buf[2] = 0;
        int length = strlen((char*)src);
        while(c < length)
        {
                // read in 2 characaters at a time
                buf[0] = src[c++];
                buf[1] = src[c++];
                // convert them to a interger and recast to
                // a char (uint8)
                dest[pos++] = (unsigned char)strtol(buf, NULL, 16);
        }

}


// this function swaps the byte ordering of binary data, this
```

```
// code is slow and bloated (write your own)
void byte_swap(unsigned char* data, int len) {
        int c;
        unsigned char tmp[len];

        c=0;
        while(c<len)
        {
                tmp[c] = data[len-(c+1)];
                c++;
        }

        c=0;
        while(c<len)
        {
                data[c] = tmp[c];
                c++;
        }
}

// this function takes the 28 most significant bits from the
// char array and converts them into an unsigned long
unsigned long chartoint(unsigned char* hash) {
    char buffer[20];
    int size = 0;
    for(int i = 0; i < 7; i++){
        size += snprintf(buffer+size, 19, "%.2x", hash[i]);
    }
    unsigned long value;
    stringstream ss;
    ss << hex << buffer;
    ss >> value;
    return value;
}

unsigned long Hashfunction::hash(unsigned int nonce) {
    // start with a block header struct
    block_header header;

    // we need a place to store the checksums
    unsigned char hash1[SHA256_DIGEST_LENGTH];
```

```
    unsigned char hash2[SHA256_DIGEST_LENGTH];

    // you should be able to reuse these, but openssl sha256 is
    // slow, so your probbally not going to implement this anyway
    SHA256_CTX sha256_pass1, sha256_pass2;


    // we are going to supply the block header with the values
    // from the generation block 0
    header.version =            1;
    hex2bin(header.prev_block,
      (unsigned char*)"0000000000000000000000000000
      000000000000000000000000000000000000");
    hex2bin(header.merkle_root,
      (unsigned char*)"4a5e1e4baab89f3a32518a88c31bc
      87f618f76673e2cc77ab2127b7afdeda33b");
    header.timestamp =          1231006505;
    header.bits =               486604799;
    header.nonce =              2083236893;
    header.nonce = nonce;
    // the endianess of the checksums needs to be little,
    // this swaps them form the big endian format you normally
    // see in block explorer
    byte_swap(header.prev_block, 32);
    byte_swap(header.merkle_root, 32);


    // Use SSL's sha256 functions, it needs to be initialized
    SHA256_Init(&sha256_pass1);
    // then you 'can' feed data to it in chuncks, but here were
    // just making one pass cause the data is so small
    SHA256_Update(&sha256_pass1, (unsigned char*)&header,
      sizeof(block_header));
    // this ends the sha256 session and writes the checksum to hash1
    SHA256_Final(hash1, &sha256_pass1);

    // same as above
    SHA256_Init(&sha256_pass2);
    SHA256_Update(&sha256_pass2, hash1, SHA256_DIGEST_LENGTH);
    SHA256_Final(hash2, &sha256_pass2);
```

```
    byte_swap(hash2, SHA256_DIGEST_LENGTH);
    return chartoint(hash2);
}
```

hashblock.h

```
#ifndef hashblock_H
#define hashblock_H
class Hashfunction {
  public:
    unsigned long hash(unsigned int nonce);
};

#endif
```