Proof of Evolution: leveraging blockchain mining for a cooperative execution of Genetic Algorithms

Francesco Bizzaro
University of Padua, Italy

Mauro Conti
Dep. of Mathematics, University of Padua, Italy

Maria Silvia Pini
Dep. of Information Engineering, University of Padua, Italy

***Abstract*—Proof of Work (PoW) is the consensus protocol introduced with Bitcoin, and is still one of the most used protocols, thanks to its security properties. However, it is very expensive in terms of energy consumption. For this reason, many other protocols have been designed in order to earmark part of the computations for useful tasks, or to reduce them, but few of these have the same properties of PoW.**

With this paper we propose a new consensus protocol for blockchains, called Proof of Evolution (PoE), that keeps the security features of PoW, and uses part of the mining computations for the execution of genetic algorithms (GAs) that some *clients* can submit. Moreover, PoE enables a form of cooperation among miners. During the mining process, in fact, miners have to maintain and evolve a population of solution candidates; PoE offers them the possibility of sharing their current best found solutions, that they can add to their *population*. This exchange seems to enhance the quality of the solutions they can achieve with the GAs in use. PoE is close to *Proof of Search* (PoS), which in turn extends PoW in order to solve optimization problems while mining. While PoS stimulates miners in submitting solutions for a problem of interest, the contribution of PoE is to encourage them to share their current best found solutions, allowing cooperation.

I. Introduction

Blockchains have been widely used since their introduction, that followed the publication of the Bitcoin's paper by Satoshi Nakamoto at the end of 2008 [4]. Currently, they are finding applications in many fields, including Finance, Artificial Intelligence, Internet of Things and Healthcare, and their use is still increasing in both industry and academia [3].

PoW, which was introduced with Bitcoin, is still one of the most used consensus protocols in blockchains, even if a large variety of different protocols have been developed in time. The strength of PoW is that the system can be considered secure as long as honest nodes collectively control more computational power than any cooperating group of attacker nodes [4]. This claim stands on the properties of the hashing puzzle miners have to solve to instantiate a block. These are: the *Intrinsic Hardness* of the problem, an easy *Solution Public Verifiability*, the *Homogeneous Hardness* of all the problems, their *Difficulty Adjustability*, *Block Sensitivity*, and *Non-Reusability*, and the *Independent Distributability* of the whole computations, that should be distributed independently to any number of nodes [7], [9].

Despite its security, PoW has some drawbacks, and one of the most relevant ones is the huge energy consumption this protocol induces in order to keep the blockchain operational. For example, the total annual power spent only by Bitcoin was already comparable to the total energy consumption of Ireland in year 2014 [5] and this consumption has been increasing each year. To handle this issue many consensus protocols have been proposed, but few of these can both be more sustainable than PoW and maintain all its properties. For example protocols like Proof of Stake [8] or Proof of Learning [9] do not have all the properties listed before, and others like Proof of Work on Random Multivariate Quadratic Functions [7] do not use less energy. With this work, we present a new consensus protocol, PoE, that leverages mining to execute GAs in a cooperative way. PoE is close to PoS [1], which is another protocol that extends PoW to enable the execution of optimization problems while mining.

Our contribution is a consensus protocol that keeps all the properties of PoW, uses part of the computational power spent in mining to execute GAs, and enables cooperation among miners, that can improve the quality of GA's solutions. Moreover, we provide an estimation of the percentage of useful work in the total amount of mining computations, and we show experimentally that the quality of the solutions reached with PoE outperforms the results reached with PoS.

In Section II, we provide a background on PoW, PoS and GAs. Then in Section III, we describe our protocol, and in Section IV we compare PoE to PoW and support the introduction of cooperation. Finally in Section V we make conclusions.

II. Background

We fix a common knowledge on PoW in Section II-A, on PoS in Section II-B, and on GAs in Section II-C.

A. Proof of Work

In PoW each block of the blockchain contains a numeric nonce. Once a miner fixes the current block's content, it has to find a nonce whose insertion makes the block's hash starting with a fixed number of zero bits. The process of searching a valid nonce is called mining, and when a miner succeeds in it, it propagates its block to all the other miners in the blockchain, receiving a prize.

The process performed by miners that use PoW can be broken down into the components represented in Figure 1. When a miner receives a block from others, it has to verify its

978-0-7381-0495-9/20/$31.00 ©2020 IEEE
DOI 10.1109/Blockchain50366.2020.00065

validity and update its local copy of the blockchain accordingly. This task is performed by the *Verifier*, represented by path *a* of Figure 1. If the miner receives more than one valid block, it has always to keep the block that points to the longest valid blockchain. Path *b* of Figure 1, instead, represents the transactions' checks to be performed before their acceptance.

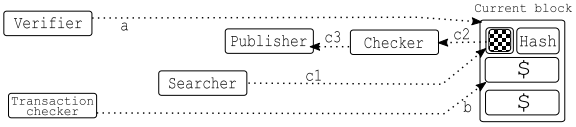


Fig. 1. Representation of Proof of Work's main steps.

Finally, the mining process is represented in Figure 1 by the path $c1 \rightarrow c2 \rightarrow c3$, and is made of multiple steps, summarized with 3 elements: a *Searcher*, a *Checker* and a *Publisher*. The *Checker* keeps verifying if the current block becomes valid whenever the nonce changes. If a proper value is found, the miner publishes the block through the *Publisher*. The *Searcher*, instead, has to constantly change the nonce until a valid value is found.

B. Proof of Search

PoS is a consensus protocol designed to use part of the computational effort spent in mining for the resolution of optimization problems [1]. It is an extension of PoW, in which the nonce number is replaced by a nonce that consists in a couple ("possible solution to the problem","evaluation of that solution"). This change ensures that a large number of solutions has to be evaluated before finding a nonce that can mine the block.

The optimization problems are submitted by some *Clients*, that have to offer a money prize for the miner that will find the best solution. Thus, while mining, each node can collect the solutions' evaluations it has to calculate to generate new nonces, and submit the best ones it finds. At the end of a fixed time, the best solution is delivered to the client, and the corresponding node is paid for its work. A client can be any node that would like to have a good solution for an optimization problem. To submit a problem, it has to define a *Job*, that is the definition of the problem with all the data needed to find the best solution. Each job must contain a program that gives a deterministic evaluation for the possible solutions of the problem. This program is called *Evaluator* and has to be written by the client, in a common programming language. Moreover, jobs must also contain a *Searcher*: another program, that implements a randomized search algorithm, that miners are advised to use for searching the best solution. At the end of a specified time, the node that finds the best scored solution for a job wins the proposed prize. However, the mining process and the process of determining the winner for a job are distinct, and the node that instantiates the block is usually different from the one who wins the prize for the job used.

We give an high level representation of this protocol in Figure 2. The mining task still requires to find a nonce

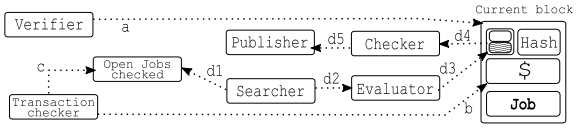


Fig. 2. Representation of Proof of Search's main steps.

whose insertion in the current block gives an hash value with a fixed number of zero bits. However, the nonce in PoS is a concatenation *(sol,eval)*, such that *sol* is a possible solution for the job selected, and *eval* is the evaluation of that solution, obtained with the evaluator specified inside the job. The mining process is represented by path $d1 \rightarrow ... \rightarrow d5$ of Figure 2: first of all the Searcher chooses the Job to use among the available ones, then it keeps trying solutions with their evaluations, until the Checker states that the block is valid and sends it to the Publisher. The verification process, instead, is represented by path *a* of Figure 2, and consists in checking the hash like in PoW, and the validity of the evaluation of the solution in the nonce. This verification must be efficient, and so the evaluator. The transactions are checked like in PoW (path *b* of Figure 2), and also Jobs are verified before they can be used for mining (path *c* of Figure 2).

In order to provide to the client a solution in less time, the protocol uses *mini-blocks* to execute multiple jobs for each block. Each mini-block contains a job, a *(sol,eval)* nonce, the ID of a miner and the hash of the last block. A fixed number of mini-blocks is required per block. Each time a valid nonce is found for a mini-block, the miner broadcasts it to the other nodes and a new block is instantiated each time all its mini-blocks are mined. A block becomes valid whenever all its mini-blocks are valid, its transactions are valid, and the hash of the previous block has been checked. A mini-block is valid if its hash has the required number of zero bits, its nonce is a solution to the corresponding job with the related evaluation, and the hash of the previous block is correct.

C. Genetic Algorithms

A GA is a search heuristic that takes inspiration from the biological evolution [2]. GAs are used as problem solving strategy to approximate optimal solutions for a problem. They are general algorithms that can be applied to a wide typology of problems. For example in pattern recognition applications, scheduling, robotics, biology and medical applications, and in general in all optimization problems [10].

To fix some terminology, the main components of a GA are: the *Individuals* (any possible solution to the problem), the *Chromosomes* (the encoded representations of individuals), the *Population* (a set of individuals), the *Genes* (the atomic components of any chromosome), and the *Fitness* (a function that assigns a score to each individual).

The execution of a GA is a cycle that begins with an initial population, and keeps updating that population until a stop condition is met. Each update is called *generation* and consists in the evaluation of all the individuals of the

population, followed by the generation of a new population through a series of *mutation* and *crossover* operations applied on a *selected* set of (fittest) individuals.

III. Our proposal: Proof of Evolution

PoE has the aim of executing GAs while mining, allowing cooperation among miners during the search of a good solution. The cooperation is limited to the execution of the GAs: the search of a valid nonce inside the mining process remains non-cooperative. Figure 3 gives an high level view on how PoE can be broken down.

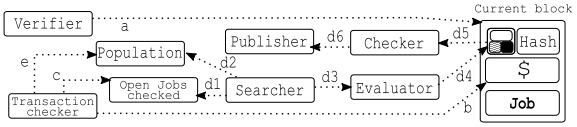


Fig. 3. Representation of Proof of Evolution's main steps.

The blocks' structure is similar to the one used by PoS, but has some differences:

- the nonce is a triple ("sol","eval","cmp"), where "sol" is a possible solution for the GA in use, "eval" its fitness, and "cmp" a measure of the complexity of the fitness function used (see Section III-A);
- all *Jobs* must be GAs implemented with a fixed API;
- miners can submit solutions in clear, before the ending of the execution of the Job, to allow cooperation.

Furthermore, a miner that uses PoE has a component that maintains a population of solution candidates for each Job. This population is constantly updated by the Searcher, but can also be updated through the cooperation with other miners. The *Evaluator* is a program defined by the *Client* like in PoS, and has to give a score to each possible solution of the GA. The *Verifier*, the *Checker* and the *Publisher* are defined like in PoS. The *Transaction Checker* has to validate transactions and *Jobs* like in PoS, but also the solutions miners decides to share. The *Searcher*, instead, is the GA a client submitted with a Job, and is supposed to be executed during mining. In order to execute more GAs in parallel, PoE makes use of mini-blocks like PoS does. In this case the mini-blocks follows the rules of the blocks in PoE.

This protocol maintains the properties of Proof of Work as well as PoS. The hashing puzzle, in fact, is still present and is used to give a probabilistic guaranty that a great number of individuals have been evaluated. The paths *a* and *b* of Figure 3 represent respectively the steps of block's validation and transaction validation, and are close to the ones in PoS. The miner also validates the submitted jobs (path *c* of Figure 3) and the shared solutions it decides to add to its population (path *e* of Figure 3). Finally, the mining process is summarized by path $d1 \rightarrow ... \rightarrow d6$ of Figure 3.

We provide more details on the nonce used in our protocol in Section III-A. In Section III-B, instead, we describe the steps a miner has to follow to share its solutions, and in Section III-C when and how miners are payed. Finally we points out some security considerations in Section III-D.

A. The structure of our nonce

In PoE's nonce a third value is requested alongside the solution for a job and its evaluation. This third value is called "Complexity" and has to be computed by the fitness function in parallel with the evaluation. This complexity value has to approximate the number of elementary operations that are performed by the fitness function in order to evaluate the individual in consideration.

Each elementary operation has its own weight, and these weights are not fixed: they are altered by a pseudo-random offset that is determined by both the hash of the previous block and the public key of the miner. Thus, the complexity of the same solution can vary among miners and among the blocks in which the nonce is inserted, but knowing the solution, the miner and the block, the complexity can be checked. Thus, miners cannot reuse the same solution for different blocks without recalculating its fitness. In addition, miners cannot steal the evaluations of others, because the complexity depends in their public key, too. This approach differs from the one adopted by PoS and avoid introducing errors in the evaluations of the solutions, nor altering the code miners have to execute.

Another use of the complexity value is in choosing the hardness to require for a job. As we have said, the value can vary among miners, but its order of magnitude must be the same, and must approximate the number of elementary operations that are performed by the fitness function. The hardness of mining is given by the number of 0 bits that are required at the beginning of the block's hash. Since our nonces contain the evaluation of a solution for a job, whose calculation depends on the definition of the job, the hardness should be adjusted considering the complexity of evaluating a solution for that job, and this value is the complexity we have introduced in the nonce. In this way, tuning the hardness of each job, we can manage to perform statistically the same amount of operations that is required mining with PoW with a fixed hardness k. Given a nonce in which the complexity of the job is x, we set the hardness y of mining that block to: $y = k - log_{16}(\frac{w+x}{w})$, where w is the number of operations spent in performing the hash of the block. We have built this formula to perform a global number of mining operations equal to the one required in PoW as order of magnitude. We justify this statement and the formula with Theorem IV.1, in Section IV.

B. How cooperation is introduced

Miners should be able to share their solutions, making impossible for others to steal them. In PoS this problem is avoided, because only the miner that declares the best solution's evaluation, publishes its solution at the end of the competition, and at that point the other miners cannot submit solutions for that specific problem anymore.

In PoE, instead, when miners want to share their current best solution, they have to follow these steps:

1) Check that the Job their solution belongs is open.
2) Check that their solution is better than the current best known solution.

3) Submit in a transaction the hash of the concatenation of their solution, their ID, and the score of that solution, obtained through the Evaluator.
4) Wait until the current block is mined, so that their previous transaction is confirmed. At that point they disclose their solution to other miners making another transaction with the complete solution in clear.
5) Now other miners can add to their local population that solution, and use it to perform crossovers. If they find a better solution they can share it following the same procedure, until the job is closed.

Instead of closing a Job when it is successfully used to mine a block (or a mini-block), in PoE the Job is closed after a fixed number of block's epochs. In this way miners can cooperate in making grow a big population of solution candidates.

C. Rewarding

In PoE there are three distinct and independent rewards: the reward for the first miner who finds a valid nonce, the reward for the miner who finds the best solution for a GA a client submitted, and a reward to stimulate miners in cooperation. The reward for mining is assigned by the system to the first miner that is able to instantiate a block like in PoW.

The reward for the best solution, instead, is paid by the client that submitted the corresponding GA, as it is in PoS. In this way the client has no interest in submitting a Job he already knows the best solution and cannot cheat.

Finally, miners should be stimulated in sharing individuals in order to build a truly cooperative system; thus they should receive a small reward for their contribution, whenever their solution is useful to others. This reward cannot be generated by the system, otherwise clients might earn money cheating; so this reward has to be payed by clients, too. Therefore, clients should allocate a fixed amount of money to pay miners that share "useful" solutions. This prize is then split among them at the end of the execution of the corresponding Job.

The system should fix some criteria to state if a solution is "useful", in order to determine when a miner has to be paid.

D. Security considerations

A client is not encouraged to submit solutions for his own job, and in particular to submit jobs he already knows a good solution, because he has to pay to use this system. However, a malicious client could build a fake evaluator, in a way such that it returns the maximum score for a particular input he knows, and the regular score for every other values in input. If miners share their solutions in the blockchain, the client can read them and, at the end of the competition, he will be the winner with the fake best solution, receiving the prize he paid. Thus, he will read some good solutions from others, without paying them. To prevent this bad behaviour, PoE requires a payment also for the set of useful shared solutions, and this payment has to be performed by the client. Moreover, if a client submits a fake best solution, no other solution will be better, and consequently no better solution will be published from that moment on.

There are also several ways to perform a Denial of Service attack that prevents execution or payment for jobs. One way is to submit a false high evaluation for a solution candidate. This attack, however, can be detected checking the solution with the evaluator. Thus, the nodes that perform the attack can be banned from the network, and the selection of the best solution can restart.

Another possible issue of PoW is caused by the competitive nature of mining, that pushes miners in purchasing custom hardware in the form of an Application Specific Integrated Circuit (ASIC) [6]. This hardware can be more efficient than generalized hardware and consequently more profitable. Unfortunately it is accessible only by a limited number of miners because of its costs. Therefore, mining has tended towards centralization in the recent past, with the total hashing power of the blockchain being controlled by a few large mining pools. With PoE, each client can specify any type of GA, so the protocol should be ASIC-resistant.

IV. Testing and Comparisons

In this section we compare PoE to PoW and PoS, and we provide some experimental results in support to the use of cooperation in our protocol.

PoS is designed to use the same order of magnitude of energy of PoW: the hardness of the hashing puzzle is tuned considering the complexity of the Evaluator for a given Job. PoE does the same, so these protocols should require the same amount of energy, but unlike PoW, PoS also solve optimization problems while mining, and PoE, as we claim, can solve these same problems, allowing an improvement in the quality of the solutions through the cooperation among miners. We support this claim in Section IV-A.

In the following paragraphs, instead, we try to estimate the percentage of useful work that is performed within mining to evaluate GAs. In both PoW and PoE, the amount of computation that is needed to find a valid nonce depends on the hardness of the mining task and on the number of operations that are performed at each mining step. The hardness of the mining task is given by the number of zeros required at the beginning of the hash string. We assume that the hash is stored in the blocks as an hexadecimal string, thus, given an hardness k, the probability of a random nonce to be valid is $p = 16^{-k}$. The mining process requires many steps in which miners generate a new nonce and check its hash, until a valid one is found. The final amount of operations is given by the number of attempts performed by all miners, multiplied by the number of operations required at each one. Assuming that an hardness k is required by a PoW mining task, we claim that:

1) the hardness of the mining task required by PoE for a job can be tuned in order to require a total amount of operations equal to the one of PoW (for any job whose complexity does not exceed that value only in a single execution);
2) if the hardness chosen by PoE for a specific job is $k-1$, then the 93% of the operations spent at each mining attempt is useful work;

3) if the hardness chosen by PoE for a specific job is $k-2$ or smaller, then the 99% of the operations spent at each mining attempt is useful work.

These claims are all consequences of our Theorem IV.1, that links the hardness of a job in PoE to its complexity.

The Point 1, in fact, recalls the second approximation of the theorem: the hardness y of a mining task in PoE can be tuned for the computational requirements x of a specific job by calculating $y = k - log_{16}((w+x)/w)$, where w are the computational requirements of making the hash of a block, and k is the target hardness in PoW.

Point 2 and Point 3, instead, can be proven as corollaries. If we have that $y = k-1$, then by the approximation 1 of Theorem IV.1, we have that $x \sim w(16^1 - 1) = 15w$. Since at each mining attempt the performed computations are $w+x = w+15w = 16w$, the percentage of useful work at each attempt is $x/(w+x) = 15/16 \sim 93\%$. Point 3 can be shown similarly.

Theorem IV.1. *Let k be the hardness of the mining task in PoW, and y the hardness that PoE should require to make the mining process with a specific job equal to PoW in terms of total amount of operations. Let w be the amount of operations spent by PoW at each attempt to perform an hash and check it, and let x be the additional amount of operations spent by PoE at each attempt to generate a new nonce by evaluating a solution candidate for the job.*

Then these two approximations hold:

1) $x \sim w(16^{k-y} - 1)$
2) $y \sim k - log_{16}((w+x)/w)$

Proof. (Theorem IV.1) We prove the first approximation; then you can prove the second one isolating the y variable from it.

In order to prove the theorem, we need the following lemma, that is easy to prove:

Lemma IV.2. *Let p_k be the probability that an hash is valid during a mining process with hardness k, and a be a desired threshold. The number of attempts N, that miners should perform in order to have a probability $K \geq a$ of having found a valid nonce is $N = \frac{log(1-a)}{log(1-p_k)} - 1$.*

We assume that the probability p_k that an hash is valid in a mining attempt with hardness k is $p_k = 16^{-k}$. Using Lemma IV.2, we get that the expected number of attempts N_{PoW} to get a probability $K_{PoW} > a$ of identifying a valid nonce in PoW is $N_{PoW} = \frac{log(1-a)}{log(1-16^{-k})} - 1$, and thus the total amount of operations required by PoW is $w(\frac{log(1-a)}{log(1-16^{-k})} - 1)$.

Similarly, we can calculate the total amount of operations required by the mining process of PoE, but this time the result depends also on the additional operations that the job in use introduces. Let x be the amount of operations necessary to evaluate a solution candidate for the job in use, and let y be the hardness required this time. The total amount of operations in this case is $(w+x)(\frac{log(1-a)}{log(1-16^{-y})} - 1)$.

If $x > 0$ and $y = k$ then the total amount of operations required to mine a block with PoE would be greater than the one with PoW. However, the two quantities can be equalized by solving in x or in y the following equation:

$$w(\tfrac{log(1-a)}{log(1-16^{-k})} - 1) = (w+x)(\tfrac{log(1-a)}{log(1-16^{-y})} - 1).$$

In other words, given a job of complexity x, it is possible to find an hardness y that makes equal the quantities, and vice versa, given an hardness y, it is possible to get the complexity x the job must have in order to equalize the quantities.
Assuming $0 \ll a \ll 1$, $k > 2$ and $y > 2$, then the following approximation holds:

$$log(1-a) - log(1-16^{-k}) \sim log(1-a) - log(1-16^{-y}),$$

and the previous equation can be simplified in:

$$\tfrac{w}{log(a-16^{-k})} = \tfrac{w+x}{log(1-16^{-y})} \Leftrightarrow x + w = \tfrac{w \cdot log(1-16^{-y})}{log(1-16^{-k})}.$$

Since $log(1-z) \sim z$ for $z \sim 0$, then

$$\tfrac{log(1-16^{-y})}{log(1-16^{-k})} \sim \tfrac{16^{-y}}{16^{-k}} = 16^{k-y},$$

thus $x + w \sim w \cdot 16^{k-y} \Rightarrow x \sim w(16^{k-y} - 1)$. □

A. The improvement of cooperation

Both PoS and our protocol can solve optimization problems through GAs, but we claim that PoE can improve the quality of the final solutions through the cooperation among miners it enables. To support this statement we have executed a set of GAs in both standard and cooperative mode, for a fixed number of generations, on a group of miners initialized with different seeds. The cooperative execution of a GA is a standard execution, in which at a fixed interval of generations, miners exchange each one his best found solution with the others.

In these tests we have considered many instances of 4 GAs implementations for: some TSP problems (minimization), some Knapsack problems (maximization), a so called *Locomotion* problem (a task with a complex fitness that involves calls to an external program), and a Symbolic Regression problem (with Genetic Programming (GP) approach). TSP is the problem of finding in a fully connected graph the minimal Hamiltonian cycle. Given a set of items, each with a value and a weight, the Knapsack problem is the problem of determining the subset of items with the maximal sum of values, and the sum of the weights below a fixed threshold. Moreover, given a 3D model with 20 joints, that represents a human being, our Locomotion problem consists in determining the movements of each joint of the model in a fixed interval of time, such that on a physics simulated environment, the model reaches the maximal distance from the starting point, without falling down. Finally, the symbolic regression problem is the problem of determining a function that best approximate a set of points, combining a set of fixed primitives using GP.

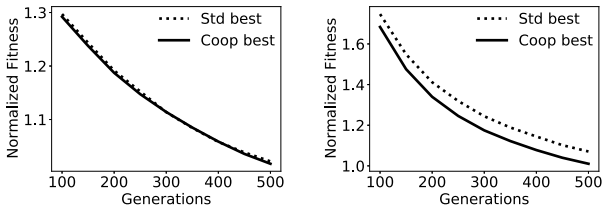
We have implemented these GAs in two different versions: one in Python and one in Go-lang. We published the complete code of both implementations on GitHub[1], so that our results, that are summarized in Table I can be reproduced. All the

[1]https://github.com/D33pBlue/Study-on-Genetic-Algorithms

Problem	Hardness	Gen	Fit Std GA	Fit Coop GA	Gain
TSP	0	500	1.056	1.0	5%
TSP	1	500	1.107	1.0	10%
TSP	2	500	1.061	1.0	6%
TSP	3	500	1.063	1.0	6%
TSP	all	500	1.078	1.0	7%
Knapsack	0	500	0.99	1.0	1%
Knapsack	1	500	0.95	1.0	5%
Knapsack	2	500	0.93	1.0	7%
Knapsack	all	500	0.96	1.0	4%
Locomotion	-	500	0.79	1.0	21%
Locomotion	-	800	0.88	1.0	12%
Regression	-	500	1.0	1.0	0

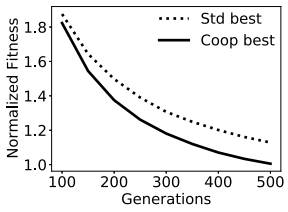
TABLE I

Comparison of the fitness of GAs executed in standard (Fit Std GA) and cooperative (Fit Coop GA) modes, on *Gen* generations, in different problems. The level of *Hardness* is proportional to the number of vertices in the TSP instances, and to the number of items in the Knapsack ones. The fitness is normalized in each row with respect to the best found solution. The *Gain* measures the percentage of improvement of the cooperative executions with respect to the standard ones.

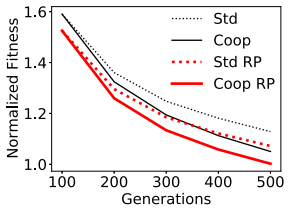


(a) TSP results with 10 miners

(b) TSP results with 100 miners.



(c) TSP results with 500 miners.



(d) TSP results with static (black) and random (red) parameters.

Fig. 4. Results of the GA implementation for TSP instances varying the number of miners, and comparison of random and static parameters for 100 miners.

tests we considered suggest that cooperation can lead to an improvement in the fitness of GAs' best found solutions.

All the GAs we have described depend on the probabilities of applying a crossover or a mutation during generations, on the size of the initial population, on the number of individuals that are selected at each generation, and on the number of those that are generated. In the previous executions we fixed these values by hand, and we initialized all the miners with them. We have then executed again the same GAs, initializing the miners with random generated parameters. We show the TSP's results in Figure 4(d), in which we compare them with the mean fitness achieved before. As it is visible, some miners reached better results, and so the curves Std RP (Std best with random parameters) and Coop RP (Coop best with random parameters), in red, are under the corresponding ones with fixed parameters, in black. The thing to notice is that also varying the parameters, the final fitness obtained through the cooperative execution are better than the ones in the standard execution. We noticed this behaviour in all the problems considered.

In the previous executions we have executed each GA in 100 parallel instances that represent miners. But the number of miners is also a parameter to take in consideration, and the number 100 is not a realistic value. We expect hundreds of thousands miners working in parallel in a good scenario. However, to conduct these tests we suffer from hardware limitations. We have tried, in any case, to identify a trend from the behaviours of our executions with 10, 100 and 500 parallel instances. In Figures 4(a), 4(b) and 4(c) we show the normalized fitness achieved for TSP instances, with and without cooperation by respectively 10, 100 and 500 miners. It is visible that the final gain given by cooperation increases with the number of miners. This was expected because with more parallel instances there is more probability of finding good individuals, that then can be shared. This trend suggests that in a real scenario the final improvements given by cooperation could be more significant than the ones reported in Table I.

V. Summary

We have designed a consensus protocol that maintains the properties of PoW, while using most of the computational effort spent for mining to execute genetic algorithms. This protocol enables a form of cooperation among miners, that increases the possibilities of approaching the real optimal solution. Moreover, we have estimated the percentage of useful work with Theorem IV.1. Finally, we have implemented a simplified but full functional blockchain with PoE, in order to validate the protocol. We published our code at https://github.com/D33pBlue/poe.

References

[1] N. Shibata, *Blockchain consensus formation while solving optimization problems*, 2019.

[2] A. Thengade, R. Dondal, *Genetic Algorithms - Survey paper*, 2012.

[3] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, *An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends*, 2017.

[4] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008.

[5] K. J. O'Dwyer, D. Malone, *Bitcoin Mining and its Energy Footprint*, 2014.

[6] Y. Georghiades, S. Flolid, S. Vishwanath, *HashCore: Proof-of-Work Functions for General Purpose Processors*, 2019.

[7] J. Ding, *A new Proof of Work for blockchain based on Random Multivariate Quadratic Equations*, 2019.

[8] Ethereum, *Proof of Stake FAQ*, https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ, accessed in 2020.

[9] F. Bravo-Marquez, S. Reeves, M.Ugarte, *Proof of Learning: a consensus mechanism based on machine learning competitions*, 2019.

[10] N. Chaiyaratana, M.S. Zalzala, *Recent developments in evolutionary and genetic algorithms: theory and applications*, 1997.