Fetch Smart Ledger

A Synergetic Computing Framework

(Version 0.16)

This is preliminary work that is subject to change.

Khan R. Baykaner, Troels F. Rønnow

Fetch.AI, https://fetch.ai

March 21, 2019

Abstract

In this paper we develop a framework that enables new types of smart contracts within distributed ledger technologies (DLTs). These contracts enable trading of solutions to computational problems that can be used within other contracts to provide a general-purpose platform for smart market mechanisms. We have designed a ledger that is built upon a notarised Directed Acyclic Graph (DAG) where the purpose of the DAG is to manage the submission of specialised contract types that may or may not alter the state of the ledger, and where the DAG is used to establish an absolute order of events. We illustrate the use of this extension to smart contract technology with an example that involves a fleet of autonomous vehicles being used to deliver a collection of parcels, and where processing nodes or miners are tasked with providing schedules for the vehicles. The contract that governs this process takes fees from the vehicles and parcel delivery customers, and uses these as payments to the miners for providing the solution to the optimisation problem. We demonstrate that, in the case of randomised heuristic optimisation, the competition between the miners improves the performance of the overall system and that there exists an optimal number of miners that allow the problem to be solved with no additional computational overhead. This is an important feature as it enables Decentralised Autonomous Organisations (DAOs) to compete directly with centralised organisations in terms of efficiency. In the case

of transportation, this means DAOs can compete directly with companies such as Uber and Lyft in performing optimal fleet operation but without the need for a central authority to manage travel schedules. We conclude by discussing applications of the framework to optimise parallel transaction execution for a sharded blockchain and a decentralised combinatorial auction.

Contents

1 Introduction

One of the key features of blockchain technology are smart contracts, which are programs that allow states to be maintained and updated on a trustless network of processing nodes [1]. Smart contracts have the potential to allow properties that apply in the real world such as the rightful owner of some property or a person's identity to be represented on a blockchain and to encode programmatically the rules for changing these states. This would allow many roles that are currently fulfilled by manual record-keeping and central authorities to instead be maintained on a decentralised ledger. The Ethereum blockchain pioneered the implementation of smart contracts [2] but these currently have limited applicability because of their high cost and limited access to computational resources. The Fetch.AI scalable ledger provides a blueprint for reducing the cost of smart contracts [3] while this article is concerned with increasing the computational power that is available for contract execution. This enhanced capability can be used to solve complex optimisation problems that arise in smart markets and facilitate new types of economic exchange.

Smart markets are auctions that allow complex bids involving, for example, logical relations such as "and" or "or" to be lodged for multiple independent items or resources. These are periodically cleared by a market manager [4, 5, 6], who is responsible for matching and clearing the auction at a specified point in time. Smart markets enable trades that would otherwise be subject to additional cost and inefficiencies on participants in the market. For example, a person may wish to bid for two different models of a particular item, such as a camera, but only wish to purchase one of the two items. Combinatorial auctions allow these preferences to be expressed in a single bid, whereas simple auctions would leave the bidder exposed to an increased risk of the undesirable outcomes of winning an auction for both or neither items. An issue associated with smart markets is that they face additional computational demands, not present in simple auctions, that arise from the difficult computational problem that must be solved to decide which participants succeeded in making the winning bids [4].

To enable a market coordinator to be implemented in a decentralised fashion, a ledger should possess: (i) an in-built mechanism for conducting auctions [7], and (ii) a means to solve computationally intensive optimisation problems [8] in a way that is sufficiently decentralised for the process to be trusted by all participants. In this paper, we develop a framework that enables smart markets to be conducted within a ledger. We refer to any ledger that deploys this solution as a *smart ledger*, and we refer to the market coordinators as *synergetic contracts*. These contracts are multi-component contracts that can operate in a decentralised fashion to enable

smart markets.

An important question in designing a smart market system lies in determining the difficulty, and therefore the payments that should be given to miners for solving a particular problem. The complexity of estimating the difficulty of computational tasks is illustrated by the example of randomly generated combinatorial search problems [8], where the time taken to solve the 10% of easiest instances and the 10% of the hardest instances of a particular problem can easily differ by a couple of orders of magnitude. Moreover, some problems might be computationally easy and it may be practically impossible to evaluate their difficulty in advance of attempting to solve them. To circumvent these issues, we instead define usefulness in terms of the economic value that is derived from the solution of any computational problem. We further restrict the types of problems that are allowed in our synergetic smart contract framework to those where the quality of different solutions can easily be computed and compared with alternatives.

Our strategy is as follows: miners act as maintainers of the market by performing timestamping of the auction pool as well as providing the necessary computational power that is needed to solve the decision problems to enable smart markets to be cleared. The second objective will be attained by creating smart contract functionality that enables the optimisation problems to be specified. The auction pool itself will be managed through as a DAG which is notarised using a blockchain. The DAG contains all data needed to finalise the auction including solutions and bids. The blockchain itself will be implemented using an extended Proof-of-Stake (PoS) consensus.

The rest of this paper is arranged as follows: we first define synergetic smart contracts. We then explain how solutions to the problems specified in these synergetic smart contracts are delivered and stored on a DAG that is used to define a temporal ordering. We then demonstrate that some classes of heuristic optimisation methods can be improved considerably through multiple independent repetitions. This is an important result as it means that, in some cases, collective optimisation can be used without any communication overhead, and therefore solve problems as efficiently as in a centralised system.

2 Synergetic Smart Contracts

Directed Acyclic Graphs (DAGs) [9, 10] have been proposed and deployed as an alternative to traditional blockchains. These data structures contain vertex elements that are chained to other vertices by references, which in the case of blockchains are simply digests (also known as hashes) of the data that is contained within the

vertex. DAGs are intrinsically parallel in that they allow elements to be added and communicated across the peer-to-peer network of processing nodes. The advantage of DAGs is that they can easily scale to record very large numbers of events, and that this scaling is not impeded by the more processing nodes joining the network. The disadvantages of DAGs are that their ordering of events is only partial and that performing search operations is expensive because of their disordered structure.

In the following section we explore how we can combine blockchains with DAGs to enable nodes in the network to construct *contests* that provide market clearing functionality on a decentralised network. We use the blockchain as a mechanism to time-stamp the DAG and thereby specify the start and end points of both the bidding and market clearing phases of the smart marketplace. We refer to processing nodes that provide solutions to the market clearing problems as *miners* to reflect the similarity of these contests to those involving the solution of hash-puzzles that are used as the consensus mechanism in Proof-of-Work blockchains.

2.1 Definition of first and second level transactions

The properties of transactions used in a contest are different from standard blockchain transactions. This is best understood by considering a contest that is carried out over a single block period, where block n defines the starting point of the contest and block $n+1$ defines its end. Transactions that are entered into the contest are submitted and certified within this period but the order of transactions is unimportant. We refer to the two different transaction types as conditional and imperative respectively.

Definition 1 *We define conditional transactions as transactions that are used in a contest.*

A contest entry, which could be a bid in an auction or the solution to an optimisation problems, can modify the world state depending on whether it is a winning entry[1]. We therefore consider a conditional transaction as one that *might* alter the state database, but where this is not guaranteed even if the transaction is valid. Ordinary blockchain transactions, on the other hand, are guaranteed to modify the state database, even if they fail, as a transaction ID is recorded along with the return value of the transaction. This leads to following definition:

Definition 2 *We define imperative transactions as transactions that are guaranteed to modify the state database.*

[1]This does not exclude there being multiple winners or varying rewards for different contributions.

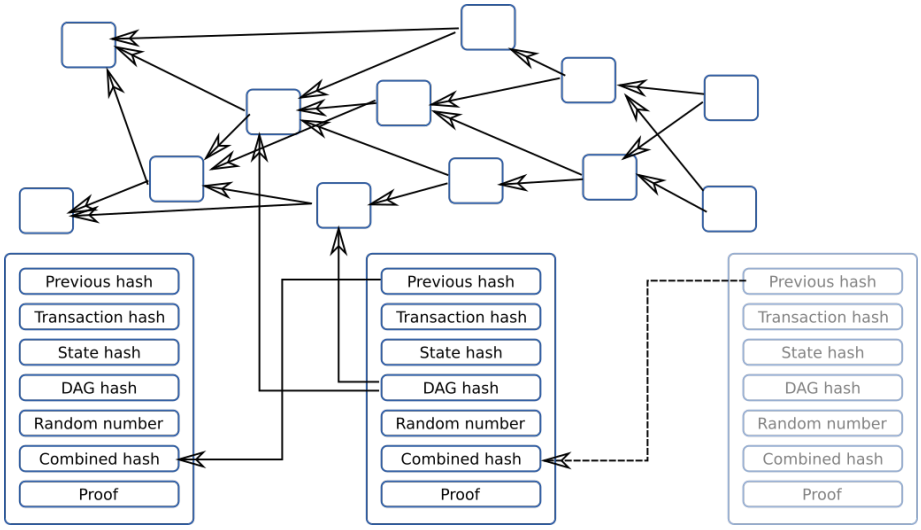

Figure 1: **Notarisation of DAG.** Each block certifies a number of DAG nodes. Any nodes referred to by any of the certified nodes implicitly becomes certified.

If the conditional transactions were to be removed from the ledger, the system would revert to being a traditional blockchain.

2.2 Notarising DAGs using Blockchains

We certify the DAG by including a number of hashes of the DAG nodes into the block. This is illustrated in Fig. 1. This serves the purpose that a consensus is created regarding which nodes exist in the DAG and provides additional ordering to the DAG structure. We further impose a strict ordering on our DAG by using a deterministic algorithm that could make use of the decentralised random beacon (DRB) [11]. This serves the purpose of deciding on a competition winner in cases where there is a tie in the value of the objective function.

3 Decentralised Solution of Optimisation Problems

In the following sections we will illustrate the different aspects of the synergetic computing framework with an example. We will consider variations of the Travelling Salesman Problem (TSP) to illustrate the types of problems that can be solved using

this approach, and will conclude by outlining a smart contract platform for these problems.

3.1 Defining a Quality Metric for Optimisation Problems

In our day-to-day lives, we have an implicit understanding of the value of services that are provided to us. Typically we have a task that needs solving and the required effort to achieve an acceptable solution is non-zero. We can divide tasks into classes and within each class we have instances of problems. For example the class of "cleaning a kitchen" can be applied to many kitchens with each kitchen constituting a problem instance. Similarly, we can define classes of problems, problem instances and solutions. Solutions indirectly demonstrate that some effort has been expended. In addition, one may consider the quality of the solution, which in the case of cleaning a kitchen loosely translates into the ratio of clean areas to dirty areas. We could refer to this ratio as an objective measure to compare different solutions. For the purpose of this paper we define classes of problems in terms of objective functions that are defined by an instance of a problem.

Definition 3 *We de ne an objective as a function* $Q_P : S \; ! \; \mathbb{R}$ *that is de ned through a set of problem parameters* $P = f p_1, p_2, \ldots, p_N g$*, and returns the value* $Q_P(s)$ *for any solution* $s \; 2 \; S = f s_1, \ldots, s_M g$*. We refer to the set* P *as the problem instance and the value* $Q_P(s)$ *as the quality of a solution* $s \; 2 \; S$*. We further refer to* $s \; 2 \; S$ *as a piece of work.*

A consequence of this definition is that any optimisation problem constitutes a class of work. Additionally, we can define an objective function for standard proof-of-work (PoW) and as you would expect, PoW fulfils the requirements to be classified as work. The objective function allows us to quantify the quality of the solution and consequently compare different solutions to the same problem instance.

An example that we will discuss in more detail in Section 4.2 is the scheduling of parcel pick-ups. We will consider a sub-problem in which a given set of parcels have already been assigned to a vehicle and that the vehicle must plan a route to collect all of the parcels. Performing the collections in the most economically efficient way translates directly into the well-known TSP problem. We illustrate the problem in Fig. 2 where a single vehicle needs to collect 29 parcels around town and return to the depot.

In this example the computation to be carried out is the discovery of a suitable route for the vehicle and the value of the objective function is measured by the total length of the journey.

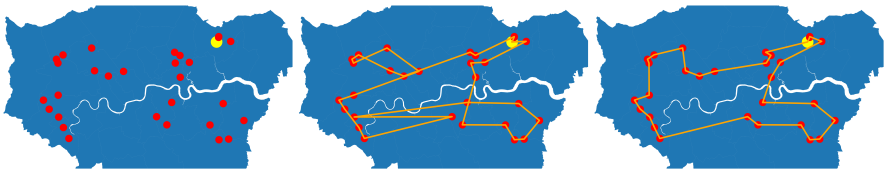

Figure 2: **Scheduling parcel pickup in London.** Left we show the original problem: a number of parcels are scattered around town (red dots) and need to be collected by a single vehicle and transported back to the depot (yellow dot). In the middle we show a suboptimal solution and to the right we show the optimal solution. The difference amounts to a 27% reduction in the distance travelled.

In the figure we illustrate two possible solutions: the first solution gets the job done, but is clearly suboptimal. The second solution requires substantially more computational work to find, but is also more efficient by 27%. Clearly the latter solution is of higher quality in this example. Under the assumption that all roads are equal and that no traffic exists in the system, the economic impact of the improved schedule is a roughly 27% reduction in fuel expenditure.

Considering a single vehicle in traffic is not very interesting because this problem could already be solved using Amazon Mechanical Turk or similar platforms. The true value lies not in solving a few particular problem instances but in designing a multi-stakeholder collaborative protocol for solving arbitrary problem instances. This key aspect is treated in much more details from an application perspective in Sec. 4.2.

For any ledger with strict ordering we can now define the best piece of work as one piece of work $s \in S^0$ that maximises $Q_P(s)$ over the set of work pieces $S^0 \subseteq S$ found within the ledger.

3.2 A Market Solution to Valuing Computation

Having established the concept of solution quality, we next define what we understand by the term useful in the context of the synergetic computing platform: loosely we refer to a solution as being *useful* if it carries any economic value. More precisely we define work as useful if it solves a problem such as traffic scheduling and the

corresponding solution is used inside a smart contract to make collective decisions. The requirement for using smart contracts arises from the need to determine the problem's economic value.

As an example, consider a smart contract for meeting a doctor at a clinic with the requirement that only one patient can be present at any time. A single person may request a meeting of varying length and may suggest multiple time periods that suit his or her schedule. This leads to a multi-stakeholder scheduling problem where the best solution provides the highest patient throughput.

With this example in mind, we can specify the following requirements for our smart contracts, which must include

1. A method to specify the problem instance.
2. A method to solve the problem.
3. A method to reach agreement on which of the proposed solutions has the highest quality.

In terms of our example, the first item implies that the network participants need a method to submit proposals for attending their doctor's practice. The second item involves miners solving the actual scheduling problem and the third item involves every node updating the world state based on the solution that is found. As a part of step three, the highest quality solution is determined, and transaction fees are awarded to the corresponding miner.

An important benefit of this approach is that it enables comparison of the usefulness of problems across classes and instances by financial means. It further removes the need to control the problem difficulty while ensuring that only problems with a positive economic impact are considered useful. This leads us to following definition:

Definition 4 *We de ne any solution to a problem that is sold through a contest as useful.*

In the following subsections we explore the mechanisms by which work is mined and sold.

3.3 Mining and verification functions

A key property of the "synergistic computing" platform is the ability to engrave the miner's public key into the work. This is done through a randomised search for a

solution to the problem where the randomised search is initialised with the public key of the miner in such a way that changing it becomes infeasible.

This has several similarities to PoW in Bitcoin, where a miner performs a brute-force search of the hashing function double-SHA256 for a result that starts with at least a number of zeros. The hashing function is used to hash the block header that contains the miner's public key and a nonce. Reversing the calculation in order to change the public key is known as pre-image attack and is a difficult task for secure hash functions.

The algorithms that are best-suited to the synergetic computing framework have many similarities to PoW, which allow them to be distributed effectively across a network of processing nodes without requiring any communication between the nodes. These properties that should apply are therefore that the algorithms should be

> Randomised - making it unlikely that miners will carry out identical computations.
>
> Heuristic - Approximate solutions should be appropriate.
>
> Finite runtime - since the distributed ledger cannot wait indefinitely for the algorithms to be executed.

The subclass of randomised heuristic algorithms known as Markov Chain Monte Carlo (MCMC) solvers are known to have these useful properties. An important benefit of MCMC lies in the independence of the solutions that they produce. Indeed, since all solutions generated by MCMC are sampled from the same probability distribution, they are by definition uncorrelated with one another. This feature makes them particularly appropriate for parallel execution. This property is discussed further in Appendix A where we describe an example of how tuning of these algorithms allows them to be very efficiently executed by a trivial parallelisation technique. In the rest of this document, we will restrict the discussion of mining functions to MCMC algorithms, although the computing framework is general and it is possible that other types of algorithm may also be suitable for different use-cases.

We will require that any mining function in the Fetch ecosystem has the following signature: the mining function takes a single (large) number as function argument and returns a data structure representing a solution to the optimisation problem. In addition to the mining function we also require an objective function as described in the previous sections. This function takes a single solution and returns a value (e.g. a double-precision floating point number) representing the quality of the solution (corresponding to $Q_P(s)$ for any solution s, as previously described).

Let us first examine how the mining function is executed. In order to run the mining function, one first constructs a DAG header. Then the hash of the header is computed, resulting in a large integer. This integer is used to initialise the mining function. Once the solution is known, the quality can be established by using the objective function to convert the solution into a single number. This means that the miner's public key will be engraved into the solution verification similarly to the hash-puzzles used in Bitcoin.

The verification function takes a DAG header that claims to have a quality of x and re-runs the mining function. This provides the solution of the problem which can then be passed to the objective evaluation function to verify the quality of the solution.

3.4 Incentivised Mining

The other important part of synergistic computing lies in the way to incentivize miners to provide solutions. A simple example consists in running a contest among miners such that the one with the highest-value proposal wins a fixed reward.[2] Such a reward is directly funded by the users that will eventually benefit from the solution. To be more specific, the procedure of the contest consists of the following sequential phases: (1) task specification, (2) problem instantiation, (3) solution mining, and (4) reward distribution.

Initially, upon the creation of a synergetic smart contract, the task is specified along with the constraints associated with every transaction (such as the identity or location of a person sending the parcel, and another receiving it). Such constraints include fees extracted from any future related transaction, which could be specified in various ways (e.g., the same flat fee for any transaction, or a fee proportional to the transaction value). Another constraint relates to the timing of the contest, i.e., when does it officially start and end. Those two pieces of information (fee and time) are both important as they implicitly specify the reward that miners can expect to receive from participating in the contest. Importantly, those constraints should be defined in relation with the miners' expected cost for participating. More concretely, the cost (e.g., in computational resources), denoted C, for any miner to participate in the contest from start to finish may be assumed to be fixed. The reward R corresponding to the sum of fees across all transactions is then offered as a prize to the winner of the contest. Assuming M miners are participating in the contest, and since every miner is equally likely to win the contest, every miner's expected

[2]If several proposals share the same highest value, then one is randomly chosen (with equal probability) and the reward goes to the corresponding miner.

payoff is R/M. It can therefore be inferred that a rational self-interested miner will participate as long as the expected benefit outweighs the expected cost, i.e., R/M C (assuming such a miner knows about C and M[3]). In other words, the above constraints specify the maximum number of miners that are expected to participate in the contest, i.e., $M_{max} = R/C$. Additional constraints need to be specified in the contract regarding the mining activity, including the objective function determining the score of any solution submitted (i.e., how efficient is the proposed solution). Note that the objective function may also specify some minimum requirement in the proposed solution (e.g., a solution to the traveling salesman problem must minimize the overall distance while covering every node).

Once such a contract has been created and recorded, the problem instantiation begins through all related transactions that are submitted, and considered to incrementally define the optimisation problem (e.g., organising parcel delivery). Note that such transactions are conditional as they may not modify the world state (if they are somehow ignored by the optimisation method that is yet to be executed). When the starting time of the contest is reached, the problem is assumed to be fully specified and miners can then engage in finding the most optimal solution (e.g., delivery schedule). Note that since the contest has a finite period, no miner has any incentive to submit more than one solution during the contest. They can simply submit their best solution to the DAG at the very last moment before the deadline, after which the winning solution can be determined. The final stage of the contest consists of creating an additional imperative transaction transferring the full reward (i.e., sum of transaction fees) to the winning miner. If there exist multiple winning proposals, then only one is randomly chosen (through the DRB) to be the implemented solution, and the corresponding miner receives the full reward.

Any transaction from the problem instantiation phase that somehow could not be considered by the implemented solution (e.g., because the corresponding parcel delivery is too distant from any other parcel location) could optionally be refunded the corresponding fee unless being considered in any subsequently created contest (as initially specified in the smart contract).

To manage such a contest, the DAG is certified by blocks in the blockchain. In this case, the blockchain serves as a timestamping mechanism and allows us to define the duration of the contest using the *block time* as a measure. The purpose of the DAG is then to record data related to the contest. The lifecycle of a contest normally consists of a period of at least three blocks. This framework can easily be extended to N blocks, but for the simplicity of this document, we will consider three blocks.

[3]Disclosing information about the number of miners participating in the contest can considerably help miners make more knowledgeable optimal decisions.

During the first block period from n to $n+1$, transactions (problem specification and instantiation) are submitted to the DAG. In the second block period from $n+1$ to $n+2$ transactions are revealed enabling the miners to submit work (proposed solutions) to the DAG. Finally, in the section block period from $n+2$ to $n+3$, a winner is selected and the corresponding solution is entered into the world state.

4 Applications

In this section we will first discuss a couple of applications. We will then turn our attention towards the design of the contract language and discuss an outline of the Fetch smart contract system. This includes preliminary source code examples from the Fetch VM (including features that have not yet been developed). The considerations in this section impose some requirements to the smart contracts in addition to those already described in the previous sections.

The goal of this document is not to serve as a complete reference of the Fetch smart contract language, but to provide some indication of how smart contracts should be specified. Unlike other smart contract systems, the Fetch VM allows for conditional as well as imperative transactions alongside objective and mining functions. These are specified as follows:

```
contract SmartContractFunction()
  // This is a standard smart contract function
endcontract

competition AuctionFunction()
  // This defines a function that is invoked at the end of an
  // auction.
endcompetition

objective ObjectiveFunction(SolutionType type) : FixedPoint64
  // This function defines an objective function
endobjective

miner MiningFunctionName() : SolutionType
  // This function defines a mining function
endminer
```

We will be using these in the following sections outline certain aspects of the applications and to discuss the design requirements for the virtual machine (VM) language. The contract functions, defined above, are standard smart contract functions that can be invoked by issuing imperative transactions. Function decorators are an important feature of the language. The competition function is by default

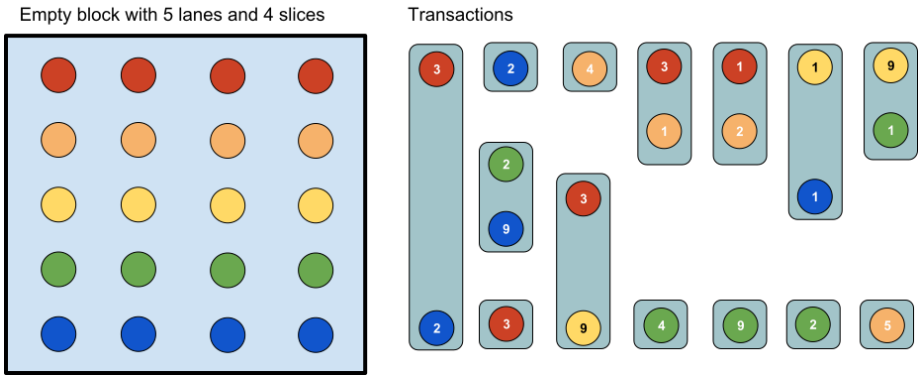

Figure 3: **Execution scheduling problem for the sharded ledger.** In this image, we show a block on the left hand side. On the right hand side we show transactions. The problem is to fit as many transactions into a block as possible.

decorated with `contest(1)`. This decorator tells the smart contract system that the contest is settled during every block, and where alternative decorator arguments lead to settlement after a different interval of blocks.

4.1 Optimising Parallel Transaction Execution

The first and most important use-case of this smart contract technology is to use miners for the purpose of optimising parallel transaction execution. As explained in our first yellow paper [3], the Fetch ledger uses structures known as lanes to store transactions and state shards. If transactions include information about which resources they are affecting, a block can be constructed such that it arranges transactions that can be executed in parallel.

We illustrate the problem of creating blocks in figure 3 below where we assume that the ledger has 5 lanes available (red, beige, yellow, green and blue). Each transaction will need to access at least one of these lanes and will pay a fee accordingly. We illustrate this through vertical boxes with coloured circles (each circle has a fee associated with it).

An important bottleneck in the sharded ledger design lies in optimising execution of cross-chain transactions so that can be executed in parallel. Practically this means arranging them in a block such that the transaction fee is maximised. This problem has two key properties: 1) as the system size increases, the problem becomes increasingly difficult to solve and 2) given any two solutions we can straightforwardly evaluate which one is better. An example of a solution to the problem above is

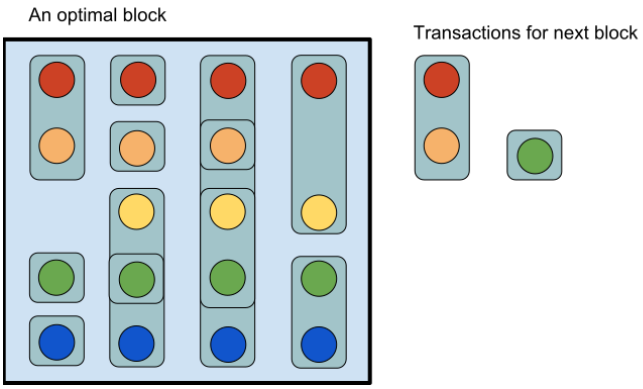

Figure 4: **Optimal solution to the block-packing problem.** Here we show what the optimal solution to the problem in Fig. 3.

illustrated in figure 4.

To solve this problem, we use the MCMC approach described in our yellow paper on the scalable ledger[3], and summarise the approach here for completeness. First we map the lane problem into a quadratic binary optimisation problems (QUBO) problem as depicted in Fig. 5

This leads to a cost function

$$C(\vec{b}) = \sum_{i<j}^{N} P_{ij} b_i b_j + \sum_{i=0}^{N} r_i b_i. \tag{1}$$

where r_i is the reward (transaction fee) for adding a transaction to the block slice, and P_{ij} is the penalty for activating two incompatible transactions. The vector $\vec{b}$ is a binary vector where entry i is 1 if the transaction is included in a slice of the puzzle and 0 if it is not. This cost function can then be minimised by using simulated annealing (SA) which fulfils the criteria that we earlier proposed for a mining function, and the puzzle can be solved by running this solver as many times as there are slices in the puzzle. For the full details on this aspect of the ledger, the reader is referred to the scalable ledger yellow paper[3].

In order to use the contract system to pick a winning block, we enable the block producer to set up a contest where the miner providing the best solution is incentivised with a fixed reward.

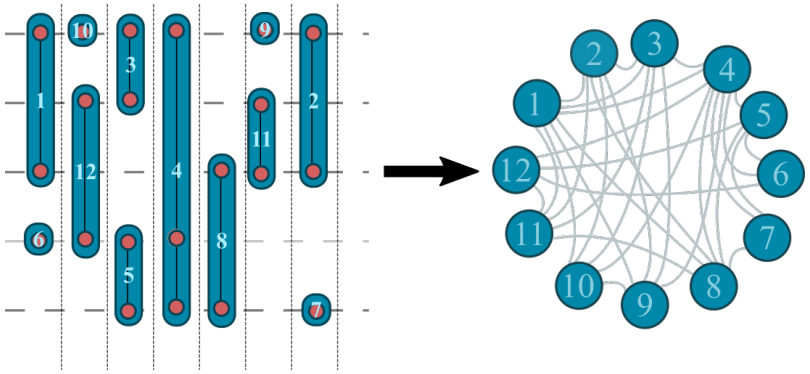

Figure 5: **Mapping the transaction organisation problem to a QUBO problem**. On the left we show the original resource lane concept with transactions added to the stack as they arrive. On the right we show a corresponding QUBO problem that contains information about mutual exclusivity: Any two connected nodes in this graph are given a penalty if they are activated simultaneously.

4.2 Parcel delivery

In this application, K delivery men have to pick N parcels up and deliver them to a depot. We want to design a smart contract system such that this can be done without a central authority. To archive this task, we make use of the synergetic computing framework. In order to do that, we first need to divide the problems into parts that are "easy" or "hard". For the difficult part of this problem we then need to describe an objective function that allows pieces of work to be executed by miners and compared against each other by the ledger to find the best piece.

Given K cars, our strategy is to first divide the set of parcels into K sets and then compute the optimal round-trip for each these sets. For simplicity, we will assume that the parcels are picked up by drones taking direct paths from parcel to parcel. Segmenting the data set is done using a K-means algorithm while computing the optimal routes for each of the sets is done using a heuristic TSP algorithm.

Let X be the set of all parcels and let $\vec{A}_i$ for 1 i K be a vector of points. We refer to $\vec{A}_i$ as a route and $A_{i,j}$ is the j'th point on the i'th route. Let $j\vec{A}_ij$ denote the number of elements in $\vec{A}_i$. Then the objective function is given by

$$E = \sum_{i=1}^{K} \sum_{j=2}^{j\vec{A}_ij} jjA_{i,j} \quad A_{i,j} \quad {}_1jj^2. \tag{2}$$

This is sum of the length of all routes. Minimising this function minimises the col-

lective travel distance and hence the energy expended, assuming that this is uniform over the parcel space.

We first consider the problem of segmenting the data set. Let X be the set of all parcels and let $\vec{A} = \{\vec{A}_1, \ldots, \vec{A}_N\}$ be N lists such that each parcel $\vec{x}_j \in X$ only occur in a single list $\vec{A}_i$. Then the first part of the strategy is to minimise the function

$$E_K = \sum_{i=1}^{K} \sum_{\vec{x} \in \vec{A}_i} \|\vec{x} \quad \mu_i\|^2 \tag{3}$$

where μ_i is the mean value of $\vec{A}_i$.

To optimise the function in Eq. (3), we deploy a Monte Carlo search technique: Initially all parcels are assigned random groups between 1 and K. Next, pairs of parcels are selected at random. The pair is swapped and if the configuration is improved, the move is accepted and otherwise, it is rejected with a probability from a Boltzmann distribution.

Having segmented the dataset, we next optimise the routes individually, we optimise the individual list of points by reordering them to minimise the function

$$E_T(\vec{A}_i) = \sum_{j=2}^{|\vec{A}_i|} \|A_{i,j} \quad A_{i,j \ 1}\|^2. \tag{4}$$

To this end, we deploy a simple SA algorithm where we propose moves consisting of swapping $A_{i,j}$ with $A_{i,k}$ to obtain $\vec{A}_i^{\prime}$. We then accept the move using the Metropolis-Hastings algorithm based on the probability given by

$$P = \min\left(1, \exp\left[\quad \beta(E_T(\vec{A}_i^{\prime}) \quad E_T(\vec{A}_i))\right]\right). \tag{5}$$

The parameter β is referred to as the inverse temperature. We refer to a sweep as one attempted swap per point in the list $\vec{A}_i$ and we typically run the algorithm for a number of sweeps. Over this period of time, the temperature is lowered to the point where only moves that lower the value of Eq. (4) are accepted. For the purpose of this document we have used a schedule that is linear in β and we discuss how to run these types of algorithm optimally in Appendix A.

Similar to the previous application, we can now establish a competition to provide the solution.

4.3 Smart Markets

An interesting potential application of this technology are combinatorial auctions, which we will illustrate with an example; a car rental company in Cambridge has

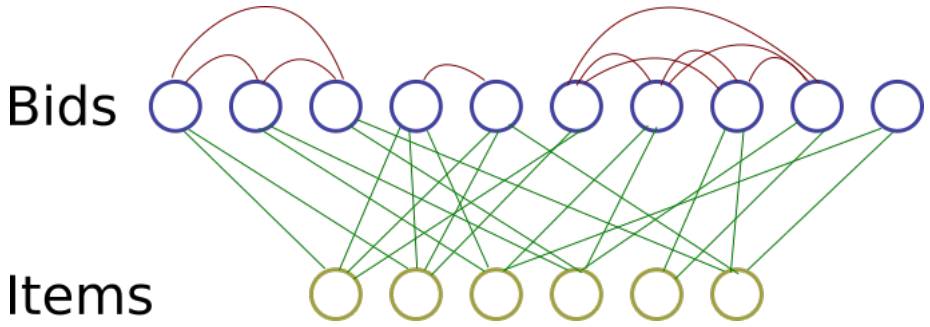

Figure 6: **Graph for smart market showing bid combinations.** Blue circles are bids and yellow circles are items for sale. Red edges illustrate exclusiveness between to bids and green lines illustrate items associated with bids. The above graph shows 6 items for sale, 4 bidders and and each of the bidders placing between 1 and 4 mutually exclusive bids.

a single car that they have available to rent, a hotel in London has a single room and a cinema in London is selling a single ticket. Three bidders make bids on these objects; bidder one places a bid on the car in Cambridge of 10 tokens, the second bidder places a bid on the hotel room for 5 tokens and the last bidder places a bid on all three items for 10 tokens for the car, 4 tokens for the hotel room and 3 tokens for the cinema ticket. Next each of the sellers specify a reserve price, which in this example could be 8 for the car, 3 for the hotel and 0.5 for the cinema ticket. There are now two possible outcomes: either the two first buyers get the hotel and the car, or the third buyer gets all three. As with the previous applications we can create an objective function that would optimise which of the configurations are better. If, for instance, we optimise for the number of buyers, then the first option would be the preferred combination whereas if we are optimising for the total sale value the second option would be better would be better.

The property that makes it difficult to find the optimal solution to these auctions is the dependencies between the bids. One could imagine that buyer 3 only wants to buy if the buyer gets all the items while, for instance, buyer three might want to rent the car and the hotel provided they receive the cinema ticket and a restaurant reservation at a nearby pizza house. We can express the combined query of all participants in terms of *or*, *exclusive-or* and *and* operations with the constraint that no item can be sold more than once. We illustrate an example smart market problem in Fig. 6.

Let x_i *2* $X = \mathit{f}x_1, \ldots, x_N\mathit{g}$ denote the i'th item for sale and let l_i denote the minimum price that the seller is willing to accept for his or her item. Let each of the M bids be represented by set B_j of tuples, each tuple containing an item index and

a bid for that item. Then define the objective function as

$$E_V = \sum_{j=1}^{M} a_j \sum_{(s,v) 2 B_j} (v \quad l_s), \tag{6}$$

where a_j is a binary variable which is 1 if the bid is accepted and otherwise 0. This function adds up to the total economic value unlocked by the auction under the assumption that the auction vector $\vec{a}$ is valid.

We note that to complete the objective function, we need to add a penalty for bids which are under the seller's reserve price. This can be done as follows

$$E_P = \quad P_1 \sum_{j=1}^{M} a_j \sum_{(s,v) 2 B_j} H(l_s \quad v), \tag{7}$$

where H is the heaviside function and P_1 is the penalty.

Likewise we need to develop a penalty for items with more than one winning bids.

$$E_B = \quad P_2 \sum_{i=1}^{N} H \left(\quad 1 + \sum_{j=1}^{M} a_j \sum_{(s,v) 2 B_j} \delta_{i,s} \right) \tag{8}$$

where P_2 is the magnitude of the penalty for each item with more than one bid. The final objective function becomes

$$E = E_V + E_P + E_B. \tag{9}$$

which is the function that could be used to define the objective function of the smart contract. Provided that the objective function and mining function are supplied by the same person and/or organisation, it may suffice to just include the term E_V in the objective function, but if the contract involves third-party mining algorithms, the full objective expression should be used to prevent possible malicious outcomes.

Having established the objective function, a mining function can be constructed in a straightforward fashion using SA. The algorithmic steps are as follows: let one sweep be an attempt to instantiate a bid, then the algorithm is run for a number of sweeps. During each sweep, a random bid is picked and this bid is agreed at the cost of clearing all conflicting bids. The new total economic reward is computed and, based on the difference between the old and the new reward, the step is either rejected or accepted. This is done in accordance with the normal SA probability that depends on the inverse "temperature" β *via* the relation

$$P = \min\left(1, \exp\left(\beta \Delta E\right)\right). \tag{10}$$

This algorithm is simple to implement in the Fetch smart contract system and allows for truly smart contracts.

4.4 Outline of Fetch Smart Contracts

With the above considerations, we turn our attention to next-generation smart contracts. We will do this through the example of implementing the parcel delivery service described in section 4.2. In this example, we consider an extended version of the TSP, where K parcels should be collected by L cars, which can be performed with a synergetic contract with the following pseudocode.

```
import KMeans, TSPSolver from heuristics.statistics;
import Euclidian from math.metrics;

struct Location
  FixedPoint64 latitude;
  FixedPoint64 longitude;
endstruct

global all_parcels : List< Location >;
global pickup_cars : Map< ByteArray, Location >;
global current_schedule : List< List< Location > > ;
global pick_routes = Map< ByteArray, List< Location > >;

contract AddParcel(Location loc)
  all_parcels.Append( loc );
endcontract

contract RegisterPickupCar(PublicKey public_key, Location loc)
  pickup_cars[public_key] = loc;
endcontract

// This function is automatically invoked upon the end of a
// contest
competition CommitToRoute(PublicKey public_key, UInt32 route)
  pick_routes[public_key] = current_schedule[route];

  // Unregistering car as it now has work to do.
  delete current_schedule[route];
endcompetition

objective RouteCost(List< List< Location > > routes)
```

```
32  var cost : Int64 = 0;
33
34  // The objective of this contract is to minimise
35  // the total travelled distance
36  for(route in routes)
37    var p1 = route[0];
38
39    // Computes the length of all routes
40    for (var i in 1:route.size())
41       var p2 = route[i];
42       var diff = p1   p2;
43       cost = cost + diff   diff;
44    endfor
45  endfor
46
47  return cost;
48 endobjective
49
50 miner MineRoutes()
51   var parcel_data = List< List< FixedPoint64 > >;
52
53   // We first convert the data format to one compatible
54   // with KMeans
55   for(var pair in all_parcels)
56     parcel_data.Append(pair.second);
57   endfor
58
59   // First we cluster the data
60   var clusters : List< List< List< FixedPoint > > >;
61   clusters = KMeans( parcel_data, pickup_cars.size(), Euclidian);
62
63   // Then we compute the routes for each of them.
64   var routes : List< List< Location > >;
65
66   for(var cluster in clusters)
67     var best_route = TSP(cluster);
68     var route: List< Location >;
69
70     for(var point : route)
71       var loc : Location(point[0], point[1]);
72       route.Append(loc);
73     endfor
74
75     routes.Append(route);
76   endfor
77
```

```
78   return routes;
79 endminer
```

The above is a minimal example the contract system focusing on essential functionality. The first function allows agents to add packages to the system. The second function allows cars to register themselves as being available for parcel delivery. The third function allows cars to commit to a given schedule. The fourth function describes the objective of the contract and the fifth function provides the mining functionality.

Delivery vehicles can now register to collect parcels and customers can add parcels to be collected. Once this data has been added to the contract, miners compute schedules that lead to the parcels being delivered.

In a more advanced version of the above contract, one can imagine how a decentralised autonomous organisation would use a similar mechanism to schedule delivery optimising all routes to ensure a fair, yet optimal fund distribution between the participants. In addition to the synergetic functionality, Fetch smart contracts also allow for more complicated operations such as matrix-matrix multiplication. This functionality is required for the smart contract, described above, and also allows smart contracts to be used to interpret and process complex multi-dimensional data.

5 Conclusion

We have presented a framework for deploying general purpose contests within a ledger and argued how these can be used to solve complex multi-stakeholder optimisation problems. We also applied this framework to a number of different applications, including optimisation of transactions in a ledger, parcel delivery and to combinatorial auctions. In future work we will explore new types types of smart markets that can be deployed within this framework. An interesting possible extension of this work is to use the framework for collective distributed problem solving, where a computational task is broken-up into sub-problems that can be solved by different participants in the network for their collective benefit.

Acknowledgements

The authors would like to thank Tian Huey Teh for inspirational thoughts on double auctions and smart markets. We would further like to thank Toby Simpson, Jonathan Ward and Frederic Moisan for useful comments and proof reading.

A Properties of Stochastic Search Algorithms

To support our proposal to use probabilistic heuristic search functions initialised with a random seed as outline in section, we investigate the the efficiency of MCMC algorithms. These considerations also apply to other types of probabilistic optimisation, but we choose to limit the scope to this class of optimisers here, for illustration. In the following, we will demonstrate how the massively parallel search technique used in Bitcoin to find solutions to PoW problems also can be used for MCMC algorithms. We will show that these algorithms greatly benefit from being run multiple times in parallel as opposed to just performing a single run for a very long time.

Let us consider the problem of collecting parcels: Given N parcels scattered across the city, find the optimal route to collect them and bring them back to the depot. In this case, the cost function C determines the sum of expenses (e.g., distance, time, gas) between any subsequent pair of parcels in the sequence $(p_0, \ldots, p_N)$ (where p_i represents the ith parcel to be picked up):

$$C(p_0, \ldots, c_N) = d(p_N, p_0) + \sum_{i=0}^{N \quad 1} d(p_i, p_{i+1}) \tag{11}$$

Where function $d(a, b)$ captures the cost of picking parcel b right after picking parcel a. The task at hand is to minimise function C.

The above cost function can be optimised using heuristic optimisation methods. Let F be such a method and let it be a function of C, T and K where T is the target runtime for a single run and K is the seed for the heuristics. Assuming that any pair of search paths is initiated with different seeds and uncorrelated, we can define the probability p of finding a solution with a cost no larger than $\overline{C}$ to the above problem. We define such a solution as optimal. In this case the probability P of finding such an optimal solution after R repetitions using different random seeds is

$$P = 1 \quad (1 \quad p)^R. \tag{12}$$

Letting $P = 0.99$ and isolating R we get the number of repetitions required to find the optimal solution with 99% probability is

$$R = \frac{\log(0.01)}{\log(1 \quad p)} = \quad \frac{\log(100)}{\log(1 \quad p)}. \tag{13}$$

The average total computational cycles required is then $R \quad G(F, C, T, K)$ where G is the average number of cycles needed for a single repetition with objective using method F for the cost function C with runtime T.

If we consider the simulated annealing algorithm discussed in the scalable ledger yellow paper[3], the function G is given as $G_{SA}(N,T)$ N T where N is a problem parameter indicating the size of the problem and T is an algorithm parameter. With these definitions, the algorithm's time-to-solution is proportional to R G by a constant c and the relation can be found by simply measuring the constant c,

$$T_{optimal} = cRG = \quad c\,G\frac{\log(100)}{\log\left(1 \quad p\right)}. \tag{14}$$

For simulated annealing we refer to c as the spinflip rate if the time T is measured in sweeps. This parameter indicates how efficient an implementation of the algorithm is and makes any scaling analysis implementation independent. For simulated annealing, the final time-to-solution is a non-trivial function of the algorithm parameter S and the problem K

$$T^{(SA)}_{optimal} = \quad c\,S\,N\frac{\log(100)}{\log\left(1 \quad p\left(K,S\right)\right)}. \tag{15}$$

To perform an analysis of the time-to-solution, we measure p for a set of problem instances.

We used the above analysis for a 1000 different problems of the form in Eq. (11). We illustrate the general dependency of the time to solution as a function of the algorithm run time in Fig. 7. As seen the time-to-solution initially decreases as the algorithm run time is increased. This may be counter-intuitive, but the reason is as follows: as the run time is increased so is the probability of finding the ground state. When the run time is low enough the number of repetitions becomes very high, but as it increases the required number of repetitions converge towards one as one would expect. However, in the other limit (infinite run time) the total time-to-solution again increases. The reason for this is that the total time spent on optimising is far greater than what is needed. In between these two extremes there is an optimal algorithm run time that minimises the total time-to-solution.

It turns out that as the problem grows large enough, the most efficient strategy is to do multiple runs rather than increasing the run time of the algorithm. We show our findings in Fig. 8. This is a very powerful result as it means that for at least some classes of heuristic optimisation, trivial parallelisation is preferable over long run times. The consequence is that this type of optimisation problem is particularly well-suited for an infrastructure that has some level of replicated effort.

In conclusion, the replicated effort that is natural to the proposed framework is desirable, at least in the case where the search algorithm is based on heuristic search functionality. For other problems with deterministic search algorithms this is less so, as everybody is executing the exact same code with exact same parameters to solve

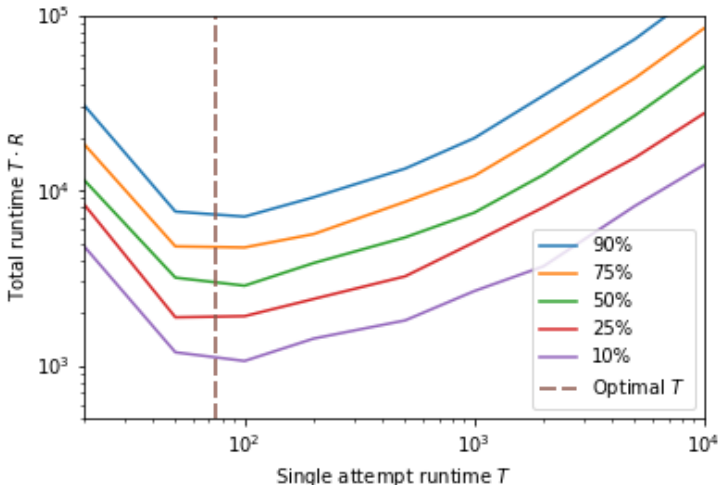

Figure 7: **Optimisation of the total computational effort for heuristic search.** We show the total computational effort that is needed to find a solution with 99% probability for various run time parameters. The search algorithms run time grows linear in the run time parameter T. The number of repetitions needed to find the solution with 99% probability depends on the run time and as a result there is always an optimal number of repetitions $R > 1$ for difficult problems. We show the results for various percentiles of 1000 randomly selected problems. The conclusion is consistent for the studied problem set.

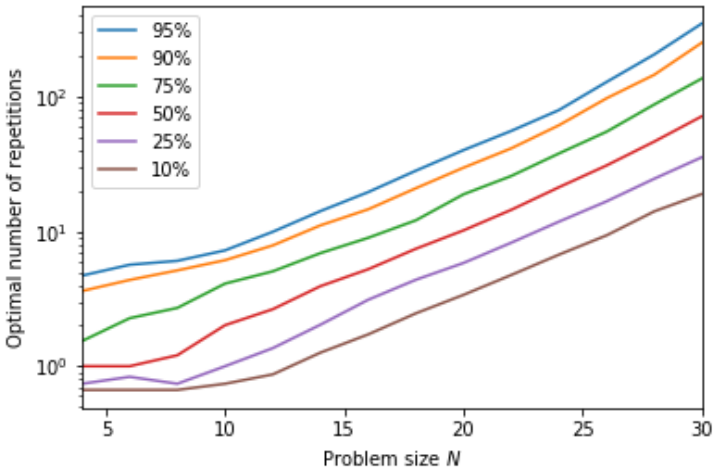

Figure 8: **Optimal number of search repetitions as a function of problem size for different quantiles.** We show how the number of repetitions needed to reach an optimal solution grows as the problem size grows. The need for repetitions grows exponentially as the problem size grows. For example, the median problem needs 10 repetitions for 20 parcels and 50 repetitions for 30 parcels in order for the search to be computationally optimal.

the problem. Hence, the proposed framework is expected to excel for heuristic search problems.

Glossary

DAG Directed Acyclic Graph. 1, 3, 5–7, 12–14

DAO Decentralised Autonomous Organisation. 1, 2

DLT distributed ledger technology. 1

MCMC Markov Chain Monte Carlo. 11, 16, 24

PoS Proof-of-Stake. 5

PoW proof-of-work. 8, 11, 24

QUBO quadratic binary optimisation problems. 16, 17

SA simulated annealing. 16, 18, 20

TSP Travelling Salesman Problem. 7, 8, 17, 21

References

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[2] G. Wood, "Ethereum yellow paper," *Internet: https://github.com/ethereum/yellowpaper,[Oct. 30, 2018]*, 2014.

[3] Hutton N., and Maloberti J., and Nickel S., and Rønnow T., and Ward J., and Weeks M., "Design of a Scalable Distributed Ledger." `https://fetch.ai/public/pdf/Fetch-Ledger-Yellow-Paper.pdf`, 2018.

[4] S. Rassenti, V. Smith, and R. Bluffing, "A combinatorial auction mechanism for airport time slot allocation," *Bell J. of Economics*, vol. 13, pp. 402–417, 1982.

[5] K. McCabe, S. Rassenti, and V. Smith, "Smart computer-assisted markets," *Science*, vol. 254, pp. 534–538, 1991.

[6] A. Pekec and M. H. Rothkopf, "Combinatorial auction design," *Management Science*, vol. 49, pp. 1485–1503, 2003.

[7] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. New York, NY, USA: Cambridge University Press, 2007.

[8] Z. Michalewicz, *How to Solve It: Modern Heuristics 2e*. Berlin, Heidelberg: Springer-Verlag, 2010.

[9] S. Popov, "The Tangle." `http://iotatoken.com/IOTA_Whitepaper.pdf`. [Online; accessed 30 September 2018].

[10] L. Baird, M. Harmon, and P. Madsen, "Hedera: A Governing Council & Public Hashgraph Network." `https://www.hedera.com/hh-whitepaper-v1.4-181017.pdf`. [whitepaper V.1.4 Last updated 17 OC T 2018].

[11] Galindo, D. and Ward J., "A Minimal Agency Scheme for Proof-of-Stake Consensus." `https://fetch.ai/public/pdf/Fetch-Ledger-Yellow-Paper.pdf`, 2018.