Abstract

The question of how to design unpermissioned ledgers that can scale to accommodate millions of transactions without compromising their decentralisation, cost or security remains unresolved. In this article we describe a scheme where resources are partitioned into groups. Transactions involving distinct groups can safely be executed in parallel, thereby enabling scaling of the ledger’s throughput. This is achieved by a novel block architecture that uses structures known as resource lanes, which enable synchronisation of events across and between different groups. We discuss how miners must solve an NP-hard problem to maximise the fees that they receive for publishing a block. At the same time, wallets managing multiple resources have an incentive to minimise the fees they pay to miners. We show that the combination of these two processes can lead to high transaction throughput under a wide variety of conditions.
Contents

1 Introduction .................................................. 3
  1.1 Grouping of Resources into Lanes ............................ 4

2 Architecture .................................................. 5
  2.1 Block Data Structure ........................................ 6
  2.2 Node Architecture ........................................ 7
    2.2.1 Main Chain ........................................... 7
    2.2.2 Storage and Execution Unit .......................... 8
    2.2.3 Wallet interface ..................................... 9
  2.3 The Block Packing Problem ................................. 10

3 Models of Transaction Generation and their Effect on System Throughput 10
  3.1 A Naive Transaction Model .................................. 11
  3.2 Exploiting the Ledger’s Lane Architecture with Smart Wallets .......... 13
  3.3 Decentralised Load-Balancing with Wallet Policies .................. 15

4 Creating Blocks .............................................. 18
  4.1 Greedy Search ............................................. 18
  4.2 Constraint Satisfaction Problems ........................... 18
  4.3 Simulated Annealing ....................................... 19
  4.4 Improving System Performance with Miner Incentives ............... 19

5 Summary ..................................................... 21

6 Outlook ...................................................... 21

A Strategies for Generating Blocks ............................... 22
  A.1 Slice-by-Slice Optimisation .................................. 22
    A.1.1 Simulated Annealing ................................... 22
  A.2 Full Block Optimisation .................................... 23
  A.3 Constraint Satisfaction Problems ............................ 23

B Formulation of Block Optimisation as a QUBO Problem .......... 24

C Heuristic Binary Optimisation using Simulated Annealing .......... 25
  C.1 Basic Simulated Annealing using Markov Chain Monte Carlo .......... 26
  C.2 Discrete Quadratic Binary Optimisation Problems ................. 27
1 Introduction

The scaling of distributed ledger technologies (DLTs) has been the subject of intense innovation ever since blockchains became established as secure, reliable mediums for economic exchange. It is clear that for any ledger to be adopted as a future standard, it must be capable of scaling to accommodate the millions of transactions that arise from widespread deployment of IoT-enabled devices. In this white paper we describe the design of a scalable distributed ledger.

One of the well-known disadvantages of the original unpermissioned distributed ledger, Bitcoin [1], is the inherent limit on the rate of transactions it can process, which arises from blocks of a specific size (1 MB) being generated at a constant rate (10 minutes). Although the block-creation parameters could be increased to raise transaction throughput, these changes have downstream effects on other aspects of the system such as security, orphan block production rate, and the minimum hardware requirements for processing nodes. As a result, the current maximal transaction throughput of Bitcoin could increase by perhaps two orders of magnitude over its current rate ($3 - 7 \text{ tx/s}$), but reaching the throughput of even the current Visa transaction system (24,000 tx/s) would require a change in its protocol [2].

The origin of the limited throughput of Bitcoin and other conventional DLTs is the sequential organisation of blocks in a chain. This means that all full processing nodes must keep a copy of the ledger and that blocks must be distributed across the peer-to-peer network in their entirety. The principal novelty of the Fetch ledger is that it allows transactions to be distributed and processed in parallel to enable scaling of its throughput in proportion to the computational power of the most powerful processing nodes on the network\(^1\).

Although the serial nature of blockchains limits their throughput, it is also crucial to one of their most important features. While most abuses of distributed payment systems can be prevented by cryptographic techniques it is unclear how to prevent an attacker from modifying the transactions recorded on the blockchain. The serial ordering of blocks in a chain means that the famous double-spending attack [1], which essentially involves altering the temporal ordering of two conflicting events, is difficult to accomplish. To succeed in inserting a later conflicting event into the global consensus, the attacker must re-write the entire history of the ledger that has been recorded since the earlier event, which becomes more difficult as time progresses. To retain the security of Bitcoin, modern ledgers must therefore be able to ensure the strict and immutable temporal ordering of transactions that they record.

Ledgers based on directed acyclic graphs (DAGs) offer increased parallelism by allowing events, which can be individual transactions or transaction blocks, to be recorded with more general connectivity to existing events stored on the ledger\(^2\). The advantage of these data structures is that they can scale to record an infinitely large number of events as they arrive

\(^1\)This reflects multiple hardware properties such as network bandwidth, CPU processing power, memory and storage.

\(^2\)Connections are typically references to earlier block hashes. It should also be noted that a blockchain is a special type of DAG. In discussing the properties of DAGs, we are referring to data structures with general connectivity and not special cases such as linked lists and trees.
asynchronously across a distributed network [3].

The disadvantage of using DAGs to record transactions is that they do not provide a clearly defined ordering of the events that they record. This makes them unsuitable as platforms for smart contracts, as it complicates detection of double-spending attacks and can lead to variability in the times that transactions take to reach finality. The Fetch ledger system uses a DAG as part of its consensus mechanism, but also maintains a strict ordering of transactions, thereby combining the advantages of blockchains and DAG-based ledger systems.

Double-spending attacks rely on submitting “conflicting” transactions; that is, transactions that attempt to read or modify the same resource at the same time. The presence of such a conflict is known as a data race condition. Double spending can be prevented by executing transactions in a strict, sequential order, thus ensuring that access to any given resource is strictly sequential. On a distributed system, this ordering of transactions must be identical across nodes that replicate the process: otherwise, the state of a resource can become inconsistent across nodes, which is the objective of a double-spending attack.

The Fetch ledger relaxes the requirement on sequential execution by partitioning resources into mutually disjoint resource groups. Transactions that affect resources from different groups are then handled by separate resource lanes. Resource lanes are a novel component of the Fetch ledger architecture. A complete ordering of transactions is defined within, but not across lanes.

The ledger enables transactions between resources belonging to different groups by entering each transaction that involves multiple resource groups into all of the appropriate resource lanes. This serves as a cross-lane synchronisation mechanism that is resolved by a novel block organisation algorithm. Transactions that do not belong to the same lane affect resources belonging to different groups, and thus can safely be executed in parallel. These features allow the ledger to scale its throughput to accommodate an arbitrary number of transactions, as described in greater detail in the following sections.

1.1 Grouping of Resources into Lanes

We use the term resource to refer to any object that has a mutable state and a unique address. In the classical use case of monetary transactions, the resources hold integers that represent currency, analogous to a bank account balance. In any ledger, a transaction will affect at least two different resources. For accounting purposes, these would be the sender and recipient of the transaction. In this context, a pair of transactions that do not have a common sender or recipient cannot, by definition, arise from a double-spending attack and can therefore safely be executed at the same time (in parallel). This feature is exploited in the Fetch ledger by partitioning the set of all resources into mutually disjoint subsets, called resource groups.

The ledger enters transactions involving resources that are drawn exclusively from one of these groups into a novel architecture component that we refer to as a resource lane (RL). The ledger system defines a strict time-ordering of the transactions belonging to any given lane. Cross-lane transactions, which involve resources from two or more lanes, are recorded
Figure 1: **Resource lane concept.** Dashed horizontal lines represent RLs. Vertical cyan bars denote transactions that involve resources (magenta circles) from one or more lanes. Groups of compatible transactions are arranged into block slices, demarcated by vertical lines, and can be executed simultaneously. For example, in the first block slice, the transaction involving resource groups 1 and 2 can be executed at the same time as the transaction that involves groups 3 and 5. The bold vertical lines represent the putative boundaries of blocks that are to be entered into the blockchain. Each block contains a fixed number of slices, which we refer to as the *slice number*. The *lane number*, which specifies the other dimension of the block undergoes a doubling after the boundary of the second block, which leads to a concomitant doubling of the transaction throughput.

in all of the relevant RLs. These cross-lane transactions enable monetary exchange between distinct resource groups and provide a mechanism for synchronising events between the RLs. A diagram of transactions organised into RLs is shown in Fig. 1. This diagram also shows compatible transactions (i.e. not involving the same resource groups) arranged into block slices.

RLs serve a similar purpose as sharding in conventional databases, and reduce the minimum storage requirements on the smallest processing node on the network. An important difference, compared to conventional sharding schemes, is that a transaction may be entered into several RLs, depending on the resources it uses. An advantage of this design is that independent peer-to-peer networks can be created for each RL. This provides a means of scaling the ledger’s execution and transaction distribution rates, since the *lane number*, i.e. the number of RLs, can be adjusted according to the transaction load.

A strict temporal ordering of transactions, including simultaneous execution of non-conflicting events, is achieved by entering transactions into a novel block structure. These blocks are connected in series and the overall organisation constitutes a blockchain. The organisation of transactions within a block is described in the following section.

## 2 Architecture

The previous subsection has outlined the motivation for introducing resource lanes and block slices. This section will describe how these structures are implemented. We first introduce
Figure 2: Illustration of the Fetch Blockchain. The leaves of the Merkle tree each reference a list of transactions rather than a single transaction. These lists are referred to as block slices. For a block to be valid, it is required that no two transactions inside a given slice involve the same resource group.

the block data structure used in the Fetch ledger, and then describe the software architecture that is run on Fetch ledger nodes.

2.1 Block Data Structure

The block data structure used in the Fetch ledger is similar in most respects to that used in Bitcoin and other blockchains. Blocks consist of a header and a body, where the body is a Merkle tree that references a number of transactions. The header contains the block hash and a hash pointer to the previous block, so that, collectively, the block headers form a cryptographically secured linked list (the blockchain). The block’s body is referenced from the header by the Merkle hash root. The only important way in which Fetch block headers differ from Bitcoin block headers is in the method used for representing and storing a proof of work. This will be explored in a future technical paper describing the Fetch consensus mechanism. The only consequence on the block architecture is that the headers contain a “proof” field which holds a digital signature. Note that the Fetch ledger design is equally compatible with traditional proof-of-work and proof-of-stake protocols.

The novel aspects of the block data structure which are designed to make the ledger more scalable are found in the block body. Whereas the leaves of the Merkle tree used in a conventional blockchain each reference a single transaction, each leaf of the Merkle tree inside a Fetch ledger block references a list of transactions. We refer to these transaction lists as block slices. This is illustrated in figure 2.

The idea of arranging transactions into block slices was first introduced in section 1.1 and illustrated in figure 1. In order for a block slice to be considered valid, it is required that no two transactions inside a given slice involve resources belonging to the same resource group. This guarantees that no race condition is present among the transactions inside the same
Because no two transactions in the same block slice can affect the same resource group, the number of transactions in a slice can never exceed the number of resource groups. As mentioned in section 1.1, each resource group is managed by a separate resource lane, which we will describe in more detail in the following subsection. The number of resource lanes is therefore necessarily the same as the number of resource groups. We refer to this number as the \textit{lane number}. The lane number constitutes one of two key parameters that can be adjusted to relieve different bottlenecks as the transaction load increases. The second key parameter is the number of block slices per block, which we refer to as the \textit{slice number}. All blocks are required to contain this fixed number of slices, and are otherwise considered invalid.

We define the \textit{block size} as the product of the lane number and the slice number. This corresponds to the maximum number of transactions that can be packed into a single block. This maximum is reached if each transaction in the block involves resources from only one group, and each slice contains one transaction per resource group. Increasing either one of the two key parameters increases the expected number of transactions per block, but they have differing effects on throughput. The slice number controls the number of transactions that can be placed inside a given block in a more deterministic manner than the lane number. This parameter can therefore be tuned to the rate at which blocks can be generated by the consensus mechanism and then synchronised across the network. As will be described in more detail in the following section, the lane number controls the number of transactions that can be executed in parallel to maintain the state database. In summary, the slice number will be tuned to the block creation rate, while the lane number will be tuned to transaction volumes.

\section{2.2 Node Architecture}

We will now describe how the different parts of the block data structure are stored and maintained across different components of the ledger node architecture. This architecture consists of three layers, illustrated in Figure 3. The lowest layer maintains the \textit{main chain}, and, as such, is the layer responsible for maintaining temporal ordering. On top of this main chain layer, the \textit{storage and execution unit (SEU)} maintains transactions and resource states. In this sense, it is the layer responsible for the "contents" of the ledger system. Finally, the \textit{wallet interface} provides a means for submitting new transactions to the ledger system in order to make changes to resource states.

\subsection{2.2.1 Main Chain}

The main chain layer is made up of components of two different types: The \textit{blockchain} component maintains the linked list of block headers. Each header contains a Merkle hash root which references the list of block slices as described in section 2.1. These block slices are stored by the remaining components of the main chain layer, which we refer to as the \textit{slice keepers}. It should be noted that the block slices contain hash pointers to transactions, and
that it is the next layer of the architecture that is responsible for recording the transactions themselves.

The number of slice keepers is equal to the slice number. Each slice keeper is associated with a certain slice index. For any given slice index \( j \), the slice keeper associated with this index maintains a time-ordered list of slices with index \( j \), each belonging to a different block.

Each of the components of the main chain layer is connected to a separate network: the blockchain component is part of a network together with all the blockchain components of all the nodes in the Fetch ledger system. And, for each slice index \( j \), the “slice \( j \) keeper” is part of a network involving all ”slice \( j \) keepers” from all nodes. The components of each type synchronise their data across Fetch ledger nodes using these networks.

Each slice keeper maintains two data structures: one to store the slices, and one to keep track of their ordering. The first of these is called the slice store. This is a map with slice hashes as keys and the actual slices as values. The latter is called the slice sub-chain, and consists of a linked list of slice hashes. Slice keepers rely on the linked list of block headers, stored by the blockchain component, to build their respective slice sub-chains.

### 2.2.2 Storage and Execution Unit

This layer manages the state of the resources by tracking and executing transactions. It is made up of components of two different types: resource lanes and contract executors.

Having introduced resource lanes in conceptual terms in section 1, we now describe these architecture components in more specific detail. Each resource lane maintains three data structures, which we call the state shard, the transaction store, and the sidechain. Recall that each resource lane maintains the state of one of the resource groups. In fact, each

![Figure 3: Architecture of a Fetch ledger node.](image)
resource lane stores the node’s local copy of the resource group whose state it maintains. This is the \textit{state shard}: a key-value store with resource addresses as keys and the resources’ states as values.

The remaining two data structures are used to maintain the history of transactions affecting the resources in the state shard. The \textit{transaction store} holds the transactions, indexed by their respective hashes. The \textit{sidechain} is a linked list of transaction hashes: this is where the ordering of transactions belonging to the lane is defined. Resource lanes each build their respective sidechains based on the ordering of the block slices, which is defined by the main chain layer. Each resource lane with a given lane ID is part of a separate network that connects all resource lanes with this same ID, each running on a separate ledger node. The lanes use these networks to synchronise transactions and their ordering across nodes.

Transactions can be thought of as instructions to make specific changes to resources held in the state shards. These instructions are executed by the \textit{contract executors}. Since only transactions belonging to different lanes can be executed in parallel, the number of contract executors is no higher than the lane number.

The executors work in parallel to execute all transactions referenced from a given block slice. The block slices themselves are processed sequentially, according to the complete ordering of slices defined by blocks’ Merkle trees and by the linked list of block headers. When a given slice is next in line, each contract executor pulls a copy of this slice from the corresponding slice keeper in the main chain layer. A simple index-based scheme determines which executor will execute which transaction. As the number of transactions referenced from a slice can be lower than the lane number (due to the presence of cross-lane transactions), some of the executors may be idle during the execution of a particular block slice. It remains to be decided whether contract executors will persist even when they are idle, or be spawned and retired as required.

Having been assigned a specific transaction, a given executor connects to each resource lane that the transaction belongs to. It pulls the actual transaction from the transaction store of one of these lanes. The executor then pulls the resources affected by the transaction from the respective state shards, and determines whether the transaction is executable (that is, whether it is consistent with the current resource states). If so, it executes the transaction, and then pushes the resulting modified resource states to the respective state shards.

\subsection{2.2.3 Wallet interface}

This topmost layer of the node architecture is responsible for providing a way of submitting new transactions to the Fetch ledger system. This involves three simple tasks: Exposing an Application Programming Interface (API), doing basic checks on incoming transactions to filter out invalid submissions, and routing transactions to the appropriate lane(s) in the SEU. The first of these tasks is carried out by the \textit{wallet API} component, and the latter two are carried out by the \textit{pre-evaluation unit}.

The wallet API consists of a standard HTTP interface with support for WebSockets. This enables the wallet interface to interact with a variety of different devices and applications that host users. These could include weakly powered devices such as Arduino boards and
Raspberry Pi chips, as well as smartphones and desktop computers. The HTTP API also has the advantage of being programming-language agnostic, which facilitates third party application development.

2.3 The Block Packing Problem

Having defined the block architecture and the two parameters, the lane number, $m$, and the slice number, $s$, that determine its maximal capacity, $C = ms$, we now discuss the problem of how to decide which transactions should be entered into a new block. From the miner’s perspective, the objective is to maximise the fees that they receive. If we begin by assuming that the fee for a transaction is directly proportional to the number of RLs that it involves, which we refer to as its cardinality, $\kappa$, the fee optimisation problem is equivalent to arranging the transactions so they occupy the maximum number of free sites in the block (Fig. 4). We refer to the proportion of sites that contain transactions as the block occupancy while the block density is defined as the ratio of the number of transactions in the block to the maximal block capacity, $C$. Both numbers reach unity when the block is completely filled with transactions of cardinality $\kappa = 1$. We further define the lane occupancy as the proportion of slices that contain a transaction belonging to this lane.

The miner’s objective of filling valid block slices can also be viewed as identifying and removing data race conditions, since this allows the transactions to be executed in parallel. Solving the block packing problem therefore aligns the miner’s objective of fee maximisation with the system’s goal of optimising transaction throughput.

If we consider a system consisting of $m$ resource lanes and a backlog of $n$ transactions as depicted in figure 4, we see that there are a number of valid configurations for the next block slice we propose.

3 Models of Transaction Generation and their Effect on System Throughput

In the previous sections we outlined the block optimisation problem. An important question in designing an implementation of the ledger concerns what type of transactions are likely to be encountered in the real world. In addressing this question, it is necessary to take account of the economic incentives for both miners and transactors within the system, and the effect that these incentives have upon the system’s throughput. In this section, we present several models for transaction generation and investigate their effect on the performance of the ledger.

We investigate the behaviour of the system when transactions are submitted from random resource lanes, and find that the relatively high cardinalities and the stochastic variability in the usage of lanes leads to sub-optimal throughput. We attempt to resolve this issue with a node policy that charges transaction fees that are proportional to the number of distinct lanes utilised by a particular transaction (i.e. its cardinality) and a smart wallet policy that attempts to minimise these fees. We find that while this discrete pricing model does
Figure 4: **The block optimisation problem.** Confirmed transactions, coloured in blue, were published in the previous block and were arranged to maximise the fee that the miner receives. Prior to publishing the second block, the miner has access to a collection of $n$ unexecuted transactions, coloured in sea green, which are candidates for entry into the subsequent block. In this example, the slice number, $s$, is 4. The first four slices containing unexecuted transactions could thus form the contents of a valid block. However, the miner could move the cardinality $\kappa = 1$ transaction, shown in the final slice, to one of the two vacant slices within the block, and thereby earn a higher fee for mining it.

Initially increase throughput by decreasing transaction cardinalities (therefore increasing the ledger’s parallelism), it also eventually causes greater variability in RL usage than the stochastic model, and has an overall deleterious effect on the ledger’s performance. Finally, we investigate a *continuous* pricing mechanism where transaction fees rise according to the demand on each of the RLs that it uses. This pricing model leads to high transaction throughput and stable transaction finality times.

### 3.1 A Naive Transaction Model

We assume that the system consists of a single node that is running one instance of the ledger and that it interacts with a large population of wallets. The model could easily be extended to a decentralised ledger, but this simplified representation captures most of the essential elements of the system. All of the transactions involve resources associated with the source and target addresses, along with a fee payment to the node.

The block’s size parameters are fixed, with slice number $s = 32$ and lane number $m = 64$ and the only tunable parameter is the number of transactions that the node receives in the interval between each block being published. In this context, “publication” of a block entails removing the transactions that could be packed into a block of the specified size.

The system’s performance is quantified using two parameters. The first of these is the transaction density (i.e. the number of transactions per published block) and the second is the mean waiting time, measured in blocks, between a transaction being received by the node and it being published. These metrics can be seen, respectively, as reflecting the system’s throughput and the confirmation time for a particular block size. We begin by considering
an analytical model where each wallet address is allocated to a random RL. We then evaluate
the load on each RL that would result under this scenario, as well as the effect on system
performance.

Our first task is to estimate the relative rates at which transactions arrive in the different
resource lanes. We begin with a naive model where we assume that a miner has access to a
total of \( n \) transactions, and that the ledger contains \( m \) resource lanes, where \( m = 2^S \) and \( S \)
is the integer sharding parameter.

The probability of a transaction affecting \( \kappa \) resources that are selected uniformly at
random being associated with a particular lane is given by

\[
p = 1 - \left( \frac{m-1}{m} \right)^\kappa = 1 - \frac{1}{m^\kappa} \sum_{i=0}^{\kappa} (-1)^i \binom{\kappa}{i} m^{\kappa-i}, \tag{1}
\]

and, assuming that the lane number far exceeds the cardinality of any transaction \( (m \gg \kappa) \),

\[
p \approx 1 - \frac{m^\kappa - \kappa m^{\kappa-1}}{m^\kappa} = \frac{\kappa}{m} \tag{2}
\]

The entry of a transaction into a particular lane therefore constitutes a Bernoulli trial. If we assume that all transactions have the same cardinality, \( \kappa \), and the total number of
transactions in the system is far greater than the lane number \( (n \gg m) \) then the number of
transactions in each lane (which we refer to as the load) approximately follows a binomial
distribution whose mean, \( \mu \), and variance, \( \sigma^2 \), are equal to each other (Fano factor \( F = \sigma^2/\mu = 1 \)) under the large resource lane number approximation.

\[
\mu = n p \approx \frac{n \kappa}{m} \tag{3}
\]

\[
\sigma^2 = n p (1 - p) \approx \frac{n \kappa}{m^2} (m - \kappa) \approx \frac{n \kappa}{m} \tag{4}
\]

These properties arise from the resource lane dynamics following a 1-dimensional advection-
diffusion process with time indexed by the transaction number \( t = n \), velocity \( v = \kappa/m \) and
diffusion coefficient \( D = \sqrt{\kappa/4m} \). Intuitively, this implies that the number of transactions
averaged over all lanes increases linearly over time, while the relative number of transactions
in each lane undergoes a random walk with respect to the mean.

This model predicts that differences in the average number of transactions in each lane
increase over time. This is undesirable, as these stochastic effects lead to some resource lanes
having greater loads than others, and would cause either a reduced density of transactions in
each block or a loss of synchronisation. To further investigate this possibility, we introduce a
control parameter, \( \lambda \), which modifies the number of transactions that arrive before a block is
published. Although, under this model, miners will always attempt to maximise the number
of transactions in a block (i.e. its occupancy), we assume that blocks arrive at a uniform
rate and that \( \lambda \) is controlled at the level of the protocol. This parameter has the following
relationship to the slice number, \( s \),
\[ \lambda = \frac{s - \mu}{\sigma} \]  

(5)

and therefore controls the normalised margin between the mean per-lane transaction occupancy, \( \mu \), and the slice number, \( s \). Substituting the values from equations 3 and 4 into equation 5 gives the following expression for the number of transactions that arrive in the time interval between each block being published

\[ n_{\text{block}} = \frac{1}{2p} \left[ 2s + \lambda^2 (1 - p) - \lambda \sqrt{(1 - p) (4s + \lambda^2 (1 - p))} \right] \]  

(6)

where \( p = \kappa/m \).

In order to verify the predictions of this analytical model and its effect on the ledger’s performance, we developed a Monte Carlo simulation of the single instance of the ledger interacting with a population of wallets. We then simulated block dynamics by generating binary transactions (between a sender and recipient so that \( \kappa = 2 \)) with the RLs sampled uniformly at random. Potential double-spends are prevented by “locking” the sender’s wallet after they submit a transaction to the ledger. This lock is then released after the transaction has been entered into a block, published and executed.

After an interval in which exactly \( n_{\text{block}} \) transactions are added to the system, a simulated annealing algorithm is used to optimise the number of transactions that can be entered into a block and the block is published. (Techniques for solving the block optimisation problem are discussed further in section 4.3). Any transactions that cannot be packed into the block are retained in a queue that prioritises them for entry into a subsequent block.

The waiting time and the average density of transactions per block are shown in figure 5. This shows that for, \( \lambda = 0.4 \), the transaction density is well below the theoretical maximum (of around 0.5) for transactions with cardinality \( \kappa \approx 2 \) but that the waiting time is stable at around 0.5 blocks. As \( \lambda \) is decreased towards zero the block density increases but at the expense of the average transaction waiting time becoming unbounded.

### 3.2 Exploiting the Ledger’s Lane Architecture with Smart Wallets

These difficulties can be resolved in a decentralised manner by implementing certain policies on the ledger and the wallet software. The first policy is for miners to make transaction fees proportional to the number of resource lanes, \( \kappa \), that are utilised by a particular transaction. This payment scheme is consistent with the externalities of entering a transaction into a block, as a transaction with cardinality \( \kappa = 2 \), for example, can always be substituted by a pair of single-cardinality transactions and should therefore be subject to at least the same fee\(^3\).

The RL fee encourages wallets to submit transactions that minimise the number of distinct RLs that they involve, and thereby increases the system’s parallelism. Another policy is to enable wallet software to be associated with accounts/addresses in an arbitrary number of

---

\(^3\)In general, the fee should be a super-linear function of the cardinality to reflect the additional difficulty of fitting higher-cardinality transactions into a block.
lanes (Fig. 6). The practical consequence of this policy is that there is no intrinsic restriction on the lanes into which a particular user can receive funds. However, there is an incentive to keep resources in a small number of lanes for subsequent payments, as illustrated in Fig. 6.

The effect of wallets controlling resources in multiple RLs can be seen by considering the example shown in Figure 6 where wallet X can choose to accept payment in a new (b) or existing lane (c). The transfer in (b) involves a lower initial transaction fee. However, if wallet X wishes to make a subsequent payment of 15 units to another wallet, in configuration (b) the payment must be made from a minimum of three different lanes with a higher associated cost than the two-lane transaction that can be used in configuration (c). This property discourages users from holding very small amounts across multiple resource lanes.

To investigate the effect of fee optimisation on transaction cardinality, we constructed a model containing $N_a = 1000$ independent users interacting with one instance of the ledger that is running on a single node. Each user starts off with an account balance of exactly 100 units in a single randomly-selected resource lane, and makes transactions with a value of exactly 10 units to another randomly selected user. A transaction fee of 1 unit per RL is subtracted from the value of the transaction to model the payment of a fee, and we therefore refer to this as a discrete pricing model. The node returns transaction fees from its highest-valued lane to a random user from the population after an average of 10 transactions to maintain a roughly constant volume of currency in circulation among the users. These transactions from the node to a user are always intra-lane and thus have unit cardinality ($\kappa = 1$).

In the first instance, we investigated the effect of the fee minimisation policy on the mean transaction cardinality and the eventual steady-state reached by the system. This initial model does not include the ledger and the block generation dynamics, but these are introduced in the following section. The results of this simulation are shown in Fig. 7, which shows that the early transactions have a mean cardinality that is close to unity and that
Figure 6: **Transactions between wallets containing a collection of resources associated with different RLs.** (a) Wallets controlled by two different individuals, X and Y, can contain accounts that are associated with multiple resource lanes. In this example, the wallets do not share any common resource lanes. There are several possible routes to making a payment of 4 units from wallet Y to wallet X. The payment can involve setting up another address in wallet X to enable receipt of a single-lane transaction, as shown in (b). The alternative is to make a cross-lane transaction to an existing address in wallet X, as shown in (c). The transaction in (c) has a higher fee as it involves two resource lanes (the fee is assumed to be fixed at 0.1 units per lane).

this increases up to a maximum that depends on the lane number before again decreasing to one. The initial increase is caused by each users’ funds becoming dispersed over multiple lanes, from which they must make withdrawals in order to meet the transaction cost. The subsequent decrease is caused by the funds in all wallets becoming concentrated in a single RL. This configuration is a stable equilibrium from a fee-payment perspective as it minimises the fees paid by all users but would have negative consequences for the ledger’s throughput.

The simulation involving proportional lane fees displayed in Fig. 7, demonstrates that a simple rule for optimising RL usage can decrease the transaction cardinality compared with the random model. This initially confers a high degree of parallelism and efficiency on the system, however, this policy eventually leads to all-but-one of the lanes becoming unoccupied. This effect is reminiscent of Gresham’s law and the tendency for more expensive forms of currency to be displaced by cheaper alternatives. In this case, transactions that use lanes containing fewer active wallet addresses accrue higher fees, on average, and this further reinforces the flow of currency into lanes more densely populated with funds. Even in the early stages of the simulation, the stochastic variability in resource lane usage, described by Eqs. 3 and 4 still applies (except with $\kappa \approx 1$) and would lead to uneven lane usage and sub-optimal efficiency. In the following section we investigate how a market-based mechanism, where the fees charged for a transaction depend on the current demand for the lanes that it uses, can be used to resolve both of these issues.

### 3.3 Decentralised Load-Balancing with Wallet Policies

In this section we investigate the effect of a simple market-pricing mechanism on the ledger’s performance. This involves making a small modification to the model so that the fee charged
by the node increases according to the demand for each of the RLs that is used by a particular transaction. We further assume that agents assign equal priority to all transactions and that their only objective in transacting with each other is to minimise their fees\textsuperscript{4}. We also assume that agents are prepared to pay an arbitrarily high fee, and the only constraint on them transacting is in having sufficient funds to make a transfer to the other party.

The transaction fee, $f_T$, in the lane pricing model has two terms. The first term increases linearly with the cardinality, and has a coefficient, $b = 1$, identically to the *discrete* transaction fee structure described in the previous section. The second term $d_i$ increases with the demand on each RL affected by the transaction. This *continuous* transaction fee has the form

$$f_T = kb + \sum_{i \in T} d_i \quad (7)$$

where $T$ is the set of RL indices affected by the transaction, $\kappa = |T|$, and the demand term, $d_i$, has the following dependence on the number of unexecuted transactions, $u_i$, that affect lane, $i$,

$$d_i = \begin{cases} 
0 & \text{if } u_i \leq s \\
\kappa(u_i - s) & \text{otherwise}
\end{cases} \quad (8)$$

\textsuperscript{4}In reality, fees will be proposed by the payee with low fee-paying transactions assigned a lower priority for block inclusion by miners, but this requires additional assumptions that are beyond the scope of this simple model.
where $d_c$ is the demand penalty and $s$ is the block slice number.

Results of simulations with $d_c = 0$ and $d_c = 1$ are shown in Fig. 8. The simulation with $d_c = 0$ is equivalent to the model described in the previous section, but includes the additional interaction with the ledger. This shows that even for $\lambda = 0.8$, where there is a small increase in transaction density compared with the naive model, that the uneven lane usage leads to a continuous increase in waiting times. This indicates that under this model an even lower value of $\lambda$ than under the naive model is required to avoid detrimental effects on the ledger’s throughput. The equivalent plot with $d_c = 1$ shows that market pricing resolves these issues as high block density and low waiting times are achieved, even with very low values of $\lambda$.  

Figure 8: Decentralized load-balancing with demand pricing. Waiting time (left column) and block density (right column) for different values of $\lambda$ obtained from running 50 independent Monte Carlo simulations with $w = 32$ and $m = 64$. The top row shows results without market pricing ($d_c = 0$) while the lower row includes this mechanism with $d_c = 1$. Simulations were terminated if the number of unspent transactions exceeded four blocks. All simulations were initialised using an identical procedure to Fig. 7 with a total of 25 transactions per agent in order to reach the peak cardinality and a quasi steady-state.
4 Creating Blocks

In the previous section we discussed the economic incentives for miners to maximise the fees they receive from publishing a block. These incentives are likely to stimulate the development of extremely optimized code for solving the block-packing problem. However, as an important first step in bootstrapping this process, Fetch will provide algorithm with functionality that represents a challenging benchmark for the community. In doing so, we outline optimization strategies that could be competitive in solving these types of problems depending on the conditions in which the ledger operates.

4.1 Greedy Search

The defining characteristic of “greedy” algorithms is that at each iteration they take the step that leads to the largest increase in the objective function. This means that they are typically fast but give solutions that are sub-optimal. For the full block optimisation problem, greedy search involves sorting the transactions according to their fee divided by their respective cardinality (i.e. the fee paid per lane), which we refer to as their score. The algorithm starts by selecting the highest-scoring transaction and placing it in the first slice of the block. The transaction with the second-highest score is then selected and placed in the first slice of the block if it is compatible with the first transaction, and in the second slice otherwise. This proceeds with transactions of sequentially lower scores placed in the first slice with which they are compatible. The algorithm terminates when every transaction has been considered for inclusion in the block, or when the block is fully occupied. Since the greedy search has a complexity of $O(su)$ (i.e. the slice number multiplied by the total unspent transactions) it is likely to be the fastest of the full-block optimisation algorithms but cannot easily be parallelised and therefore may not be faster than slice-by-slice simulated annealing.

4.2 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) occur widely in scientific and engineering disciplines with the most famous real-world example being Sudoku puzzles. In order to formulate an optimisation problem as a CSP it is necessary to specify a set of discrete variables $V$, a set of constraints $C$ and a set of domains of values $D$. The full block optimisation problem can be mapped to a CSP where the variables are transactions, Rls provide the constraints and block slices are the set of domains. Solving the block optimisation as a CSP involves iteratively searching for assignments of the variables that meet the constraints. However, unlike greedy approaches, back-tracking is used to search over a larger number of configurations of the variables which leads to better solutions but at the expense of greater computational complexity.
4.3 Simulated Annealing

Simulated annealing is an optimisation technique used to minimise cost functions [4, 5, 6]. This technique can be applied to quadratic binary optimisation problems (QUBO). We explain simulated annealing in detail in App. C.1. In App. B we can formulate the transaction packing problem as a QUBO problem by associating each transaction with a binary variable $b_i$. In the following section we go through the translation from RLs to the full QUBO problem and explain how this can be solved. We then explain how to map a binary problem into a spin glass problem in App. C.2.

4.4 Improving System Performance with Miner Incentives

In this section we discuss the miner’s role in organising transactions within blocks. The results were generated by mining over a quarter of a billion block slices with varying computational effort and with varying numbers of RLs. In the first part of the analysis, we investigated how the fees that block miners are able to recover depends on the number of available transactions. One example of this is shown in Fig. 9.

The figure demonstrates that the mining fee is strongly correlated with the transaction availability. This is a very desirable feature, as it implicitly incentivises miners to obtain as many unspent transactions as possible and, hence, ensure that a high transaction exchange rate between nodes is achieved. However, encouraging a high transaction exchange is not sufficient for the system to operate optimally.

The other necessary feature is that the number of transactions executed in parallel should increase as the transaction loads increase. In the second part of our analysis we investigated the relationship between lane occupancy and the fees collected by the miner. To accomplish
Figure 10: **Lane occupancy as a function of mining fee.** We show a density plot of occupancy as a function of mining fee. Of note is the piecewise linear behaviour and the strong correlation between mining fee and occupancy. The general trend shows that miners who earn the highest fees also generate blocks with a higher block occupancy.

This, we calculated the lane occupancy for 25,000,000 mined block slices and compared them with the corresponding fees. This procedure was carried out on problems with 64, 128, 256 and 512 lanes, with results summarised in Fig. 10.

It can be seen that the general tendency is the same, regardless of the number of lanes. One interesting feature of Fig. 10 are the “clusters” seen in the histogram. These clusters arise from differences in the number of unspent transactions that are available to the miner as we saw in Fig. 9. Indeed, these discs move towards being centered around higher mean occupancy as the pressure on the system increases. What this means effectively is that our design of blocks responds positively to an increase in pressure.
5 Summary

In the previous sections we have discussed two mechanisms that may limit transaction throughput. These are: 1) The consensus mechanism for ordering the transactions and 2) the rate of transaction execution. The former primarily depends on how many transactions are contained within a single block whereas the latter is dependent on how many transactions are executed in parallel. We have presented a ledger architecture that takes both of these features into consideration. The first property is vital to the Fetch vision in accommodating the vast economic activity we anticipate will take place on the Fetch network, and the latter becomes important as the number of cycles needed to execute a smart contract increases.

The Fetch ledger, described in this paper, introduces resource lanes to ensure scalability in the number of transactions and block slices to ensure that we can accommodate parallel execution of transactions on the ledger. The architecture presented will be able to make use of dynamic variables whose value are agreed upon through the consensus mechanism to dynamically ensure that the ledger has capacity to meet transaction demand.

We have further demonstrated how the idea of having transactions associated with lanes creates the need for smart wallets and how these smart wallets will be incentivised to improve the performance and throughput of the system.

Finally, we have discussed three different methods that can be used to construct new blocks. By implementing a number of these methods, we have shown that the miners’ “selfish” behaviour helps improve the overall economy by increasing the transaction throughput. The block structure can also be used to improve transaction throughput by adjusting the number of lanes in the system, and the number of slices in a block.

6 Outlook

While this paper has outlined the architecture of the Fetch ledger and investigated idealised models for the behaviour of participants in the network, we have not yet released estimates of the transaction throughput that the system is capable of attaining in practice. Over the coming months, the Fetch team is planning to release performance metrics on several different aspects of the system including serialization/deserialization rates, network latency, consensus, mining and proof-of-work. At this point in time, we have demonstrated synchronisation of 40k transactions per second between nodes, and can mine blocks at a rate greater than 30k transactions per second using a parallelized simulated annealing algorithm.
A Strategies for Generating Blocks

In order to create new blocks, different strategies can be applied depending on the objective. The cost of running a node will increase with the number of resource lanes, as each lane typically will need its own process and networking connections etc. One objective is therefore to balance lane occupancy to ensure that the system is utilising its existing resources to the fullest. As explained in the previous section, each transaction has lane fees associated with it. A consequence is that the fee for utilising different lanes may vary from lane to lane. This affects miners’ strategies for their objective of maximising the sum of fees within a given block.

In the following sections we will explore solving these two objectives using various programming techniques. The algorithms suggested in this whitepaper are meant as initial algorithms to optimise the lane utilisation, and we expect that as the ecosystem grows, the community will develop additional algorithms to further improve the efficiency of our system.

Generally we can apply two strategies to generate a new block. In the first and simplest strategy, we create and optimise the individual slices independently. In the second approach, we consider the entire block as an optimisation problem and look for a global solution. We start by exploring the slice-by-slice optimisation and then turn to full block optimisation.

A.1 Slice-by-Slice Optimisation

In this strategy we optimise each block slice independently. This has the advantage that we can compute the slices in parallel, and is expected to be particularly beneficial for large transaction volumes. In this section we consider the formation of block slices using heuristic search algorithms. To this end, we first reformulate the problem into a QUBO problem.

A.1.1 Simulated Annealing

Simulated annealing is an optimisation technique used to minimise energy functionals of the form[4, 5, 6]

\[ E = \sum_{i<j} J_{ij} s_i s_j + \sum h_i s_i. \]  

(9)

For completeness of this work we explain the details in App. C.1. In order to solve this problem using simulated annealing, we first reformulate the problem in terms of spin variables. The details can be found in App. C.2 and lead to

\[ J_{ij} = \frac{P_{ij}}{4}, \]

\[ h_i = -\frac{1}{2} \left( \frac{1}{2} \sum_j P_{ij} + I_i \right) \]

where \( P_{ij} \) are the penalties defined in the previous section and \( I_i \) are transaction incentives.
A.2 Full Block Optimisation

The advantage of solving the block as multiple independent slice optimisation problems is that the algorithms can be trivially parallelised to deal with very high transaction volumes. However, it is likely that algorithms optimising the entire block will provide better solutions, and may be more appropriate when transaction volumes are low. At very high volumes, it is also possible that even efficient, parallel simulated annealing code may not execute quickly enough, which would necessitate the use of fast but sub-optimal “greedy” approaches. However, we anticipate that the financial rewards for mining will stimulate the development of block optimisation algorithms by the community. We have also explored several alternative approaches that can be used to solve this problem.

A.3 Constraint Satisfaction Problems

A CSP is defined as a triple \((V, D, C)\) where \(V = \{V_1, \ldots, V_n\}\) is a set of variables, \(D = \{D_1, \ldots, D_n\}\) is a set of the respective domains of values, and \(C = \{C_1, \ldots, C_m\}\) is a set of constraints.

Each variable \(V_i\) can take on the values in the non-empty domain \(D_i\). A constraint \(C_j \in C\) has a scope \(S\), which is a subset of \(V\) containing \(k\) variables, and a \(k\)-ary relation that restricts the values that can be simultaneously assigned to the variables in \(S\). A solution is an assignment of all variables in \(V\) that is consistent, i.e. that does not violate any of the constraints in \(C\).

Any instance of the block formation problem can be converted to a CSP where: \(V\) is the set of transactions, \(D\) is the set of block slices in the block, and \(C\) contains one constraint for each transaction lane, which is satisfied if all transactions belonging to that lane are assigned different slices. A solution to the CSP is then an assignment of the transactions to the slices of the block.

If a transaction belongs to more than one lane, its assignment will have to satisfy more than one constraint. On the other hand, a transaction belonging to only one lane only needs to take a value that does not conflict with the other transactions of the same lane.

If there exists a solution, i.e. all transactions can fit in the block, there is a polynomial algorithm that can find it:

1. select a lane with unassigned variables
2. find an assignment for each variable that satisfies the constraint of the lane and the current domains of the variables
3. for each assigned variable \(V_i\), remove its value from the domains of all the variables sharing a lane with \(V_i\)
4. if some variables are not yet assigned, go to 1

Since all permutations of block slices are equivalent, the order in which variables are assigned to values is not important as long as all constraints are satisfied. Since steps 2 and
3 ensure that all values in the domains are consistent with the constraints and the assigned variables at any step, the algorithm never needs to backtrack. The algorithm is executed at most once for each lane, and is therefore polynomial.

This approach is similar to solving a Sudoku puzzle with an empty grid.

B Formulation of Block Optimisation as a QUBO Problem

To reformulate the formation of a block slice as a QUBO problem, we enumerate through the unmined transaction \( T_i \) where \( 1 \leq i \leq N \) and \( N \) is all unmined transactions. We then associate each transaction with a binary variable \( b_i \), which is 1 if the transaction should be added to the next block slice and 0 otherwise. We can then formulate the formation of a block slice as an optimisation problem with the objective function

\[
C(\vec{b}) = \sum_{i<j}^{N} P_{ij}b_ib_j + \sum_{i=0}^{N} r_ib_i. \tag{10}
\]

where \( r_i \) is the reward for adding a transaction to the block slice, and \( P_{ij} \) is the penalty for activating two transactions that could cause a race condition. This is illustrated in Fig. 11.

We can now optimise Eq. (10) to find solutions that form block slices, and then form a block using these solutions. We illustrate this in Fig. 12 where we show four solutions found for the previous problem. These four solutions are then hashed and added into a block as can be seen. This is done by having the first solution belong to the first leaf of the Merkle tree, the second solution belong to the second leaf etc. We note that the order in which the transactions within a single block slice are ordered does not matter as these can be executed
Here we suggest ordering them according to lowest affected lane using normal SHA256 for order verification. It should be noted that it is guaranteed that this order exists and is unique as each lane is represented exactly once.

As a result of the block formation above, the updated state of the system is depicted in Fig. 13. In the example scenario given here, two leftover transactions did not make it into the proposed block. We note that it would be suboptimal to then attempt to publish them in their own block as this would leave four lanes empty in each slice and hence, the overall value of the block both from a miner perspective and a system performance perspective would be poor.

Various heuristic methods can be used to solve Eq. (10). We explore the application of simulated annealing to this problem in the following subsection.

C Heuristic Binary Optimisation using Simulated Annealing

In this section we review the application of simulated annealing algorithms to solving QUBO problems. We begin by describing simulated annealing (SA) and then explain how it can be used to perform non-deterministic polynomial-time hard (NP-hard) binary optimisations.
C.1 Basic Simulated Annealing using Markov Chain Monte Carlo

Ising spin glass models have been studied for more than a century, and are central to a wide variety of scientific and engineering disciplines. The model originates from the study of interactions between atoms with discrete angular momenta (spins) arranged on a regular lattice. Analytical calculations of the free energy are difficult to evaluate for many Ising models, due to their very large number of degrees of freedom. This difficulty stimulated the development of Markov Chain Monte Carlo techniques, which can be used to estimate the physical parameters of Ising models computationally.

The energy, $E$ of an Ising model is given by the equation

$$E = \sum_{i<j} J_{ij} s_i s_j + \sum_i h_i s_i \quad (11)$$

where $J_{ij}$ are the couplings that define the potential between pairs of spins, $s_i$ and $s_j$ and the terms $h_i$ define the local field. An instance of the Ising model is specified by the set of couplings and set of local fields $\{J_{ij}\}$ and $\{h_i\}$. The spin variables $s_i \in \{-1, 1\}$.

A common method for optimising the spin configuration in Eq. (11) is inspired by a 7000 year-old method for material optimisation, known as annealing. As in reality, simulations of the annealing process involve initialising the system at a high effective temperature, where it can freely explore the energy surface, and then slowly lowering the effective temperature until it converges on a low-energy configuration. If the cooling is performed slowly enough, then the system is guaranteed to reach the ground state.

The first step of simulated annealing involves randomly initialising each spin variable to one of the available states. Markov Chain Monte Carlo (MCMC) is then used to sample a new state. This involves randomly picking one of the spins $s_i$ and proposing a new state $s_i' = -s_i$. This new state is accepted with a probability given by the Boltzmann distribution, which means that it is accepted if the energy difference $\Delta E = E(s') - E(s)$ is less than zero,
and will be accepted with a probability \( p = e^{-\beta \Delta E} \) if \( \Delta E > 0 \), where \( \beta = T^{-1} \) is the inverse temperature.

For a particular spin variable \( s_k \), the energy difference is given by the equation

\[
\Delta E_i = -2s_i \sum_j s_j J_{ij} - 2s_i h_i
\]  

(12)

where the sum is over the neighbouring interactions, where \( J_{ik} > 0 \). For negative energies, a random number uniformly distributed between zero and one is sampled, and the new state accepted if it is below the probability threshold, \( p \).

### C.2 Discrete Quadratic Binary Optimisation Problems

In this section, we show the equivalence between QUBO problems and the more conventional spin glass representation. For a QUBO problem instance described by the parameters \( \{W_{ij}\} \) and \( \{f_i\} \) the cost is given by

\[
C(\vec{b}) = \sum_{i<j} W_{ij} b_i b_j + \sum_i f_i b_i.
\]  

(13)

The objective of a QUBO algorithm is to find a configuration that minimises the cost function \( C(\vec{b}) \), and we refer to this as the optimal solution, or as the ground state.

It will often be impossible (or at least hard) to find the optimal solution due to the size of the solution space and one must settle for a good solution. In order to use simulated annealing to obtain a good solution we need to reformulate the above problem in terms of spin variables. Letting \( b_i = \frac{1}{2}(1 - s_i) \) we find

\[
C(\vec{b}) = \sum_{i<j} W_{ij} b_i b_j + \sum_i f_i b_i
\]

\[
= \frac{1}{4} \sum_{i<j} W_{ij} + \frac{1}{2} \sum_i h_i
\]

\[
+ \frac{1}{4} \sum_{i<j} W_{ij} s_i s_j - \frac{1}{2} \sum_i s_i \left( \frac{1}{2} \sum_j W_{ij} + f_i \right)
\]

\[
= E_{\text{offset}} + \sum_{i<j} J_{ij} s_i s_j + \sum_i h_i s_i
\]

with

\[
J_{ij} = \frac{W_{ij}}{4}
\]

\[
h_i = -\frac{1}{2} \left( \frac{1}{2} \sum_j W_{ij} + f_i \right)
\]

\[
E = C - E_{\text{offset}}
\]
and

\[ E_{\text{offset}} = \frac{1}{4} \sum_{i<j} W_{ij} + \frac{1}{2} \sum f_i. \]  

(14)
Glossary

$C(\vec{b})$ Objective function for which a minimum is to be found. 27

$E$ Spin glass energy. 26, 28

$J$ Coupling between spins. 22, 26–28

$S$ The sharding parameter. 12

$\Delta E_i$ Local spin energy. 27

$\kappa$ The cardinality of a transaction. 12–16

$\mu$ Mean value. 12

$\sigma^2$ Variance. 12

$b$ Binary variable that tells whether transaction $i$ should be added to slice or not. 19, 27

$h$ Local field for a spin variable. 22, 26–28

$m$ The number of lanes. 11, 12

$n$ The number of unmined transactions. 12

$s$ The number of slices. 11

$s$ Spin variable associated with the $i$’th transaction. 26, 27

API Application Programming Interface. 9

CSP Constraint Satisfaction Problems. 18, 23

DAG directed acyclic graph. 3, 4

DLT distributed ledger technology. 3

MCMC Markov Chain Monte Carlo. 27

NP-hard non-deterministic polynomial-time hard. 25

QUBO quadratic binary optimisation problems. 19, 22, 25, 27

RL resource lane. 4, 5, 10, 18, 19

SA simulated annealing. 25

SEU storage and execution unit. 7–9
References


