University of Groningen

Bachelor Thesis

A Matchmaking Algorithm for a Blockchain-based Trading Platform

Author:
Alexander Daffurn-Lewis
S2906031

Supervisor:
Dr. Vasilios Andrikopoulos
Dr Dimka Karastoyanova

August 15, 2018

university of groningen
faculty of science and engineering

UNIVERSITY OF GRONINGEN

Abstract

Faculty of Science and Engineering

A Matchmaking Algorithm for a Blockchain-based Trading Platform

by Alexander Daffurn-Lewis
S2906031

This bachelor thesis presents an implementation of a matchmaking algorithm for use in a blockchain-based trading platform. The creation of the platform was a collaborative effort and was split into two parts, with the first being the creation and implementation of a blockchain application that can support such a trading platform. The second part involves the design and implementation of a matchmaking algorithm that can automatically match offers to sell goods and demands to buy goods to create sales agreements between the two parties, which is the subject of this thesis. The main contribution of this work is an algorithm that takes in both offers and demands, and outputs the matches in $\mathcal{O}(o * b)$ time where: o = number of offers and b = number of bids. This complexity is shown to be sufficiently fast for the identified use case.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Background

Blockchain is a technology that allows a network of computers to agree at regular intervals on the true state of a distributed ledger. Such ledgers can contain different types of shared data, such as transaction records, attributes of transactions, credentials, or other pieces of information. The ledger is often secured through a clever mix of cryptography and game theory, and does not require trusted nodes like traditional networks (Catalini, 2017). Blockchain's first application Bitcoin, was created to serve as a electronic payment system that did not rely on a trusted third party for regulation, yet was still protected from users with malicious intent (Nakamoto, 2009). Since that first innovation all manner of potential blockchain based applications have been suggested, from digital property where ownership rights to assets are stored in a blockchain where they cannot be disputed[1], to online voting where votes are cast as transactions to prevent corruption and vote manipulation[2]. Using blockchain for marketplaces is a commonly suggested use case[3], however currently there very few implementations of this, and all are quite specific to their domain.

Initially when we started this project we wanted to focus on a solution for the local energy domain, however as it turned out this has been one of the areas with the most activity which I will cover later in the related work. Instead we decided to focus on making a generalized solution for a simple automated market that can then in the future be adapted for more complex markets with a specific domain in mind and to explore whether the use of blockchain in such an application is viable. We chose this because as mentioned previously all the other work in this area is very domain specific and a general solution and proof-of-concept was certainly missing.

1.2 Application Outline

We envision to the application to work as follows: Participants can create offers to sell goods and demands to buy goods which list the amount of goods that want to buy or sell along with the price they want to pay for them. Periodically a routine will run that will take all of the offers and bids and match offers and bids that

[1] https://www.ibm.com/developerworks/cloud/library/cl-adopting-blockchain-for-enterprise-asset-management-eam/index.html

[2] https://followmyvote.com/blockchain-voting-the-end-to-end-process/

[3] https://www.forbes.com/sites/rogeraitken/2017/10/24/whats-the-future-of-online-marketplaces-blockchains-technology-impact/

are seen to be suitable to be paired (the definition of a suitable pairing is covered later in this thesis) and a sale agreement between the two parties will be created. This sales agreement acts as an obligation for the offering party to supply the bidding party with an amount of goods for a certain price, which is agreeable to both parties. Once the offering party supplies the bidding party with the goods, the bidding party can notify the sales agreement that they have been received and the sales agreement will then automatically create a payment agreement which is an obligation for the bidding party to pay the offering party the amount owed, which is calculated on the details specified in the sales agreement. Once this has been satisfied the trade can be concluded.

The offers, bids and sales and payment agreements all exist on the blockchain as smart contracts, and their creation and modification are handled using transactions. To create the next block in the chain and apply these transaction they must be mined as you would expect in a blockchain application. However the traditional proof-of-work mechanism for mining where all nodes race to create the next block is not suitable for our application, and as such it was decided that instead we would implement a different mechanism based on the proof-of-stake concept. Rather than having everyone mining at the same time, instead we elect a leader who is the only one allowed to mine transactions. The leader rotates in a round robin fashion through nodes that have been deemed to have a stake in performing honest work. This leader can then also run the matching routine and include the transactions that create the sales agreement contracts in the blocks they produce while they are leader.

1.3 Methodology

As outlined above one of the core components of the application is the matchmaking routine which is ran by the leader with the purpose taking offers to sell goods and bids to buy goods, and creating a sales agreement between two compatible parties. This must be an automatic process which takes no input from either the leader or any other third party, as doing otherwise would defeat the purpose of using blockchain in the first place. While the purpose of matchmaking and its position within the application is clear, the question remaining its how exactly should it fulfill its purpose. The other similar projects in the blockchain-market domain make very little, if any, mention of how they perform matchmaking (discussed in more detail in the next section) and as such I will also attempt to provide some discussion as to how traditional matchmaking ideas can be applied to the blockchain domain.

This thesis will therefore attempt to answer the following research questions:

> What factors influence the development of a matchmaking algorithm for use in a blockchain-supported market?
>
> What does a matchmaking algorithm that takes into account these factors look like?

To answer these questions we will first look into the matchmaking domain, especially where there has already been overlap with the blockchain domain, and attempt to learn what needs to be taken into consideration to design a algorithm

that is fit for use in our application. Then we will present an algorithm based on requirements elicited from the project documentation and what we have learnt from the related work.

This thesis shall contribute a key ingredient in our efforts to develop a generalized proof-of-concept solution for a simple blockchain-based market application, as well as providing a start point for any future development in the blockchain-matchmaking domain.

1.4 Thesis Outline and Task Distribution

This bachelor thesis is part of a collaborative effort between myself and Pieter Dekker to create such an application. The requirements and design for the platform itself are not covered in this thesis, instead they can be found here[4][5] and I will make references to them as necessary. Pieter Dekker handled the creation of the blockchain platform - which has been created using Ethereum as a basis - that will support the market and all information about that process is covered in his thesis. My thesis will cover the design and implementation of a matchmaking algorithm that will run as a protocol on top of the blockchain to enable it to function as a automated market. The final algorithm and a link to the final application can be found here[6].

This thesis will be organized as follows: First we will explore the related work around the blockchain-matchmaking domain to find out what factors will influence the creation of the algorithm (Chapter 2). Next we will set out the requirements for the algorithm drawing from the requirements of the project as a whole (Chapter 3). Then the design and implementation of the algorithm will be covered in detail utilizing what has been learnt from the related work (Chapter 4). Finally we will evaluate the algorithm with respect to its requirements (Chapter 5) and state our conclusion along with any future work we have identified along the way (Chapter 6).

[4] Project Requirements
[5] Project Design
[6] https://github.com/Daffurn/BlockchainMatchmaker

Chapter 2

Related Work

2.1 Blockchain Marketplaces and Matchmaking

As mentioned earlier the use of blockchain technology in marketplaces is a very commonly suggested idea. Hemang Subramanian discusses the advantages of a decentralized marketplace over the traditional e-marketplace (Subramanian, 2018), particularly identifying how a blockchain network can enforce contracts of sale through distributed validation whereas traditional e-marketplaces rely on a third party (usually controlled by the platform) to ensure transaction validation, which is not always so reliable. Subramanian also identifies that blockchain marketplaces can have significant advantages when it comes to transaction speeds due to fast network validation, compared to the often slow and cumbersome methods that the traditional e-marketplaces use.

Due to the popularity of this idea a large number of projects and applications and been proposed and developed in this domain. The most common idea is a blockchain-based e-commerce platform such as OpenBazaar[1] and Origami[2]. These function similarly to their traditional counterparts like Amazon where sellers can post listings which can be browsed by users who then have full autonomy to select which products they may be interested in. In these cases there is no automated matchmaking taking place and therefore there is not much to learn from these kinds of projects in regards to how we should approach implementing matchmaking in our project.

Another common implementation is for usage in microgrids in the local energy domain. A microgrid is a discrete energy system consisting of distributed energy sources (including demand management, storage, and generation) and loads capable of operating in parallel with, or independently from, the main power grid. The primary purpose is to ensure local, reliable, and affordable energy security for urban and rural communities[3] For example in a local community that receives a lot of sunlight a few of the residents may decide to invest in solar panels to meet their energy needs instead of buying energy from a bigger provider. On especially sunny days they may even generate more energy than they need, in which case usually the only option is to give this energy back to the grid in exchange for compensation from the provider, the amount of which is specified by a feed in tariff. Feed in tariffs however are not available everywhere (Klein et al., 2008) in which case the resident has to invest in costly energy storage solutions or the production capability goes to waste. In this case it may be desirable to be able

[1] https://openbazaar.org/
[2] https://ori.network/
[3] https://www.generalmicrogrids.com/about-microgrids

to sell the excess energy to your neighbours who may not have their own production capabilities. A microgrid can then be formed where residents who are also producers (prosumers) can transfer their excess energy to the microgrid, and residents who are not can pay to take energy out of this grid. A blockchain and cryptocurrency system can then be used to enable this: prosumers can gain coin by producing electricity for the grid, and this coin can then be exchanged for electricity from the grid, or sold to others who can then exchange it for electricity themselves. One example of this is the Brooklyn Microgrid[4] and while it seems that their application does perform automated matchmaking of some sort, they do not discuss in their documentation how this is done. Mihaylov et al., 2014 and Mengelkamp et al., 2018 both propose solutions for the local energy domain, however in their papers neither seem to touch on the topic of matchmaking.

Apart from these domains there are a good number of other applications designed for differing types of marketplaces, such as Zenodys[5] for trading digital assets, and Augur[6] which has built a platform for a prediction market. However most of these application again do not implement any type of automated matchmaking. There were, however, a couple of applications that did have a system for automatic matching of offers and bids. Powerledger[7] is developing a peer-to-peer renewable energy trading platform, and their whitepaper states that consumers and prosumers buy and sell energy using their own cyptocurrency Sparkz, and while it is not explicitly stated how exactly this processes happens, they do also state that they "...have developed their own unique trade matching algorithms which transact available power equitably, between Prosumers and Consumers, without favoring any of the Participants."[8] Unfortunately this is as much detail as is given, and as such there is not much to learn. Another project, iExec[9], have created a platform for trading computing resources. This project seems to have the most similarities to what we are attempting, they state that "Applications and users place “work orders” for their works to be executed based on pre-selected criteria" and then "Workers place a sell order defining the price at which they are willing to execute works." which is very similar to how we envision our application to run. When it comes to matchmaking iExec state in their whitepaper "iExec team plans to design and adapt a simplified version of the well-known and tested ClassAd that powers the CondorHTC distributed system..."[10] HTCondor is "...a specialized workload management system for compute-intensive jobs" and they state that their ClassAd mechanisim "provides an extremely flexible and expressive framework for matching resource requests (jobs) with resource offers (machines)."[11] It is apparent from this that iExec’s approach to matchmaking is to utilize an existing algorithm in the computing resource domain, which unfortunately does not seem to have any overlap with our generalized approach, however if in future we decide that our application should support this domain then their approach may be of use.

[4] http://www.brooklynmicrogrid.com/
[5] https://www.zenodys.com/
[6] https://www.augur.net/
[7] https://powerledger.io/
[8] https://powerledger.io/media/Power-Ledger-Whitepaper-v8.pdf
[9] https://iex.ec/
[10] https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf
[11] https://research.cs.wisc.edu/htcondor/description.html

In conclusion it is apparent that there is very little discussion in the blockchain-marketplace space about how automated matchmaking should take place and what discussion is available is not relevant to our approach to a marketplace platform. However we can learn that applications seem to be taking unique and domain specific approaches to how they perform matchmaking, so for our project perhaps it would be best to follow suit and try to create a generalized matchmaking algorithm for a generalized implementation.

2.2 Traditional State of the Art in Matchmaking

For matching of offers and bids where price and quantity are the only variables we can look to stock markets for inspiration. The CME group which operates a number of different exchanges around the world list a number of different algorithms that they use in their exchanges[12]. From this it is apparent that there are a good number of options available.

According to Janecek and Kabrhel the most widely used matching algorithms are Price/Time (also known as First In First Out (FIFO)) and Pro-Rata with Price/-Time being the most commonly used (Janecek and Kabrhel, 2007). We will not go into detail on the workings of the algorithms here, but we note that the Janecek and Kabrhel found that Price/Time "motivates participates to narrow the spread", meaning keeping the difference between the highest price a buyer is willing to pay and the lowest price a seller is willing to offer as minimal as possible. However they also put forward Price/Time can demotivate participants to place orders, as their orders will be placed last in the queue. For Pro-Rata they state that conversely it "does not motivate to narrow the spread in the natural way" however it does motivate participants to place large orders.

For this project Price/Time is a more suitable influence for our matchmaking algorithm. Keeping the spread narrow is a desirable trait to have in a market and due to the speed at which blockchain process transactions even if an offer or bid is submitted at the back of the queue, it should not be long before it is considered.

2.3 Quality of Service-Based Service Selection

So far we have only looked into matching based on the properties of quantity and price, and this follows as our general solution will require users to provide this information in their offers and bids. However later we will discuss the future work that can follow our project, part of which is expanding the application to support goods that may have more than just these two properties. For example matchmaking in the cloud computing service domain will need to consider characteristics such as disk space, amount of RAM required and GPU runtime requirements[13]. In this case an algorithm that we design to support just quantity and price would not be sufficient, so here we will briefly look into the state of the art in matchmaking in domains where there is more to consider than just these two characteristics.

[12] https://www.cmegroup.com/confluence/display/EPICSANDBOX/Matching+Algorithms
[13] https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf

Yu et al present a number of efficient algorithms for service selection (Yu, Zhang, and Lin, 2007). We learn from their work that the suitability of these algorithms depends on heavily on the domain and the model that can be made from the services provided in the domain. Jula et al comprehensively cover the Cloud Computing service domain (Jula, Sundararajan, and Othman, 2014), and cover a number of solutions in regards to service composition. Once again their solutions are very much dependant on the model that can be made of the domain and the service provided.

From this it can extrapolated that solutions to matchmaking and service selection in more complex domains than ones where the only characteristics are quantity and price require algorithms that are very domain specific. The algorithm that we shall design to meet our general solution will certainly not be suitable, and if we are to consider supporting more complex domains then effort must be put in to modelling that domain so we can make an informed choice into what methods may be suitable to use for matchmaking.

Chapter 3

Requirements

The requirements for our algorithm follow directly from functional requirements 2 and 2a and non-functional requirements 1 and 3 from our project domain and requirements document[1] which are:

1. The application must run reasonably fast
2. Offers and demands are paired to form a match agreement. This agreement acts as an obligation for the offering party to supply the demanding party with an amount of goods for a certain price.
   (a) Offers and demands are only paired if the quantity and price are agreeable to both parties. This should be calculated from the information in the bids
3. The source code of the application must be available openly as participants are placing their trust in the system

Functional requirements 2 and 2a provide us with the need to create a matchmaking algorithm that takes in bids and offers and outputs match agreements between a agreeable bid and offer. Therefore the algorithm will need to calculate all the information needed to create a match agreement. We still need to define exactly what is meant by agreeable, and this will be covered in the design section.

Non-functional requirement 3 applies directly to our algorithm in that its source code should be freely available to view. In addition it should be clearly published exactly how the algorithm works and what is taken into account when it performs matchmaking. Also the algorithm should perform in such a manner that it always gives the same results given the same input. This is so that there is no unfair advantage given to participants who study the algorithm in an attempt to predict how the results will look, which would be possible if there are any elements of randomness within the source code. These requirements will ensure an even playing field for all participants.

Non-functional requirement 1 is a point of more discussion however. While in the project requirements we stated that what is defined as reasonable is mostly based on the domain, there are some other implications of speed that are not considered. Functional requirement 2b states that matchmaking is performed by the leader who is selected to form the next block, and from the project design documentation[2] we can see that matchmaking occurs during block creation, and that

[1] Project Requirements
[2] Project Design

the transactions that create the match agreements are to be included in the block that is to be created. Therefore it follows that the matchmaking process must be faster
than the time it takes to create the block. As our selected blockchain technology for this project is Ethereum, we now need to calculate the minimum time it takes to form a block in Ethereum, and this time will then dictate the maximum run time for our matchmaking algorithm. We can see from Figure 2.1 that on the main Ethereum network the block creation time is usually between 10-20 seconds. However our application may be ran privately as this can help satisfy requirements 4 and 4a which state that energy costs of running the chain should be kept to a minimum and that there should be no additional costs to participating in the network. In this case the difficultly of mining a block will be set as low as possible to keep transaction fees as low as possible. In this case the time it can take to mine a block is usually between 3-10 seconds. Therefore it is suitable to set 1 second as an upper limit to the execution time of our algorithm, with expectations that the network size is not overly large, in which case the mining time should increase as well.

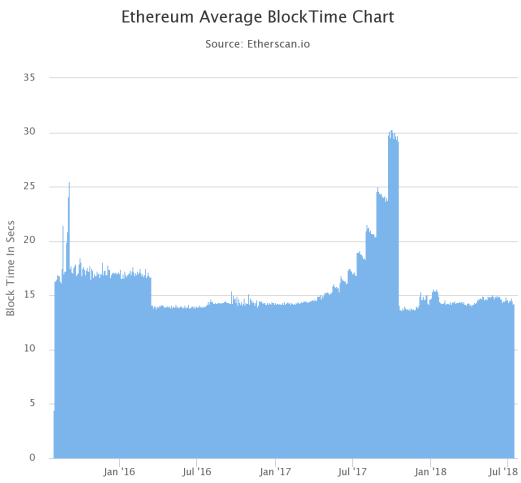


Figure 3.1: Ethereum Mainnet BlockTime

We should also consider the following: As we are using Ethereum as our blockchain technology we should consider integration with the Ethereum development environment when writing our algorithm. The majority of Ethereum application development is done in JavaScript using the node.js environment and as such it would be advantageous if our algorithm is written in the same language. It should also be integrated in the application is such a way that it can be ran by any node in the network that is running the application. There should be no other requirements for it to run in terms of technology needed outside what is provided by the application. By delivering the algorithm as a node.js module this should also be satisfied.

From this we can now elicit the following requirements for our algorithm:

3.1 Functional Requirements

1. The algorithm should take in a list of offers and bids, and output a list of matches between suitable bids and offers along with the all the information necessary to create a match agreement.

2. If there are no suitable offers and bids the algorithm should output an empty list

3. The algorithm should be delivered as a node.js module for ease of integration with the rest of the application

4. The algorithm can be executed by any node running the application in the network

3.2 Non-functional Requirements

1. The algorithm's runtime must not exceed 1 second, up to a reasonably large number of nodes, bids and offers in the network
2. The algorithm source code should be available openly
3. What factors the algorithm takes into consideration when performing match-making should be available openly
4. Given a specific input the algorithm should always give the same output

Chapter 4

Design and Implementation

4.1 Algorithm Input

Our first consideration when it comes to designing our algorithm is the what data we receive from the offers and bids and in what format we receive it. From the project design documentation[1] we can see that these contracts have the following data fields:

- `minimal_volume`
- `maximal_volume`
- `minimal_unit_price`
- `maximal_unit_price`
- `expiration_date_time`
- `owner` (which holds the address of the owner)
- `used` (a boolean to signify whether it has been matched or not)

From these we only require the minimal and maximal volume and the price which is read from either `minimal_unit_price` or `maximal_unit_price` depending on whether the contract is an offer or bid (discussed in the next section). We can also extrapolate whether the contract is a bid or an offer from this, however all this information is neatly provided in the *getDetails* method from the contract. Also due to how the contracts are gathered from the blockchain we receive them ordered from oldest to newest as with the Price/Time algorithm discussed earlier, so we do not have to concern ourselves with having to sort them ourselves. As we are working with JavaScript we collect all this information as a JSON object as this format is nice to work with. An example of a valid algorithm input can be seen in Listing 4.1.

[1] Project Design

Listing 4.1: Example Algorithm Input

```
[
    {
        address: "0x123f681646d4a755815f9cb19e1acc8565a0c2ab",
        type: "offer",
        minQuantity: 10,
        maxQuantity: 15,
        price: 3
    },
    {
        address: "0x5ed8cee6b53b1c6arce3ad7c92f4fd7e1b8fad9f",
        type: "bid",
        minQuantity: 12,
        maxQuantity: 13,
        price: 4
    }
]
```

4.2 Format of Price and Quantity

Before we move on to the next step we will first explain our decisions in regards to the format we require for the price and quantity characteristics. For our general solution we decided upon a one-to-one matching relationship between offers and bids, meaning that only one bid can be matched with only one offer. We chose this as our aim was to provide the simplest solution, and while it may be beneficial to allow bids and offers to be split so that, for example: one bid for 100 units can be matched with two offers for 30 and 70 units; this is considered a domain-specific requirement and will be treated and discussed as a possible extension to our application in the future.

This one-to-one relationship in matching is very relevant when deciding which format to ask users to provide for quantity and price when creating bids and offers. On one end of the spectrum we could ask users to provide in their bids and offers one value for both price and quantity and then only match with bids or offers that have the exact same value. However this is too strict of a limitation and would mean finding a match could take a long time and will potentially lead to many bids and offers expiring due to the lack of perfectly matching pairs. Alternatively we could ask for just one value and than widen this to form a range of values either side of the given value. In this case matching would be easier and quicker, as to calculate if a bid and an offer are agreeable we just need to see if their ranges for quantity and price intersect. The problem with this approach however is the widening of the provided value into a range would have to be done by the application, which leads to the question of finding an appropriate method for widening. In the end we decided that the best way to do this is to ask the user to provide the range themselves. In this case matchmaking is not limited to finding corresponding bids and offers with the exact same values which will speed up the overall process and we do not have to make any assumptions about what is acceptable to the user by implementing a widening process. If the user

so desires they can even set the minimum and maximum values of the range to the same value, essentially providing a single value, and while it may take more time for their bid or offer to be matched than if they provided a larger range, the option is there if that is what the user requires.

4.3 Subtyping Quantity and Price Ranges to Enable Matchmaking

This section relies heavily on the previous work by Vasilios Andrikopoulos *et al* in their paper on QoS Contracts Formation and Evaluation (Andrikopoulos et al., 2010). In particular we will make use the following definitions described in their work:

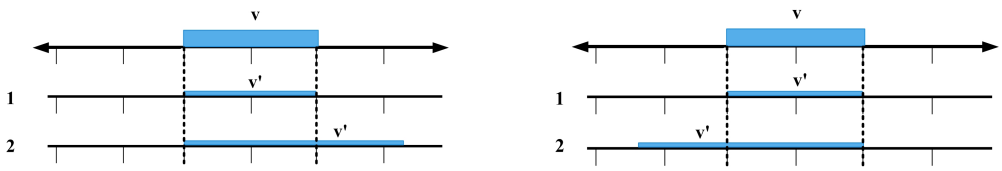
- A characteristic is said to be *monotonic* if larger values are considered "better"
- A characteristic is said be be *antitonic* if smaller values are consider "better"
- The value range of a monotonic or antitonic characteristic is said to be *strict* if it is not acceptable to extend to the boundaries of the range towards "better" values
- The value range of a monotonic or antitonic characteristic is said to be *relaxed* if it is acceptable to extend to the boundaries of the range towards "better" values

This is illustrated in Figures 4.1 and 4.2, where v is the original range and v' is a range that also acceptable as a match based on the properties of v.

We can now apply these properties to our characteristics of quantity and price to define what is acceptable as a match.

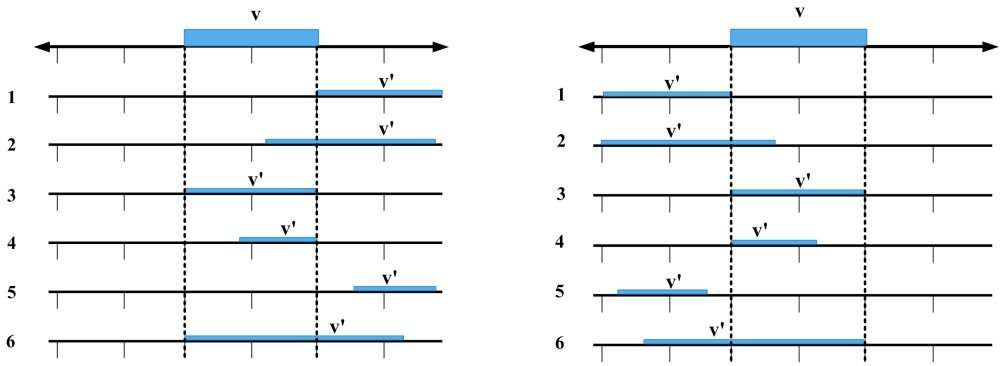
4.3.1 Quantity

We define quantity as *monotonic* and *strict* with a small extension. It is monotonic as indeed for both parties it is desirable that "more is better". For example if a producer makes an offers to sell 5-10 units of their goods, they would rather sell 10 units than 7 units and the same applies for consumers. Therefore if two ranges from an offer and a bid are deemed acceptable, we will take the larges value that lies in the overlap of both ranges for our match agreement value. As for strict, if a user puts up an offer to sell 5-10 units, then they most likely would not be happy to accept a bid for 11-15 units, as perhaps their production facilities only allows for a maximum of 10 units produced at a time. However we do slightly relax the strict definition in one case: If the two ranges overlap at all then it is acceptable to match these, as for example if the offer is for 12-15 units, and the bid is for 10-14, both parties can happily agree on an amount of 12 for the match agreement. Table 4.1 gives some examples of what is acceptable for matching of quantity:



(a) Monotonic (b) Antitonic

Figure 4.1: v is Strict



(a) Monotonic (b) Antitonic

Figure 4.2: v is Relaxed

4.3.2 Price

Price is a little trickier than quantity to subtype, because while it is always *relaxed* it can be *antitonic* from the perspective of the consumers, but *monotonic* from the perspective of the producer, in essence the producer wants to get a higher price for their product, but the consumer wants to pay a lower price. We deal with this by doing the following: When creating an offer we ask the producer for the minimum price that they would want for goods and only the minimum price. We then use this provided value as the lower boundary of a range from that value to $+\infty$. This works as, for example, if a producer offers their goods for €3 per unit, they would certainly be happy to accept any price that is above €3. For consumers we do the following, when they create a bid we ask them for the maximum price they would be willing to pay. We then use this value to become the upper boundary of a range from 0 to that value. Again this follows as if a consumer is willing to pay €4 a unit for their goods, they would be happy with paying €3 per unit.

Will still need to decide what value to choose when these ranges intersect. Currently we decided to favour the consumer and choose the lowest value that lies in both ranges as we feels that keeping prices low within the market is desirable. However this decision is most likely very domain dependant and can very easily be tweaked in the algorithm if required, but for now we will continue with this decision. Table 4.2 illustrates price matching:

Bid Quantity Range	Acceptable	Matched Value
$[6,8]$	Yes	8
$[1,4]$	No	-
$[11,15]$	No	-
$[3,6]$	Yes	6
$[7,15]$	Yes	10

Table 4.1: Table to show Acceptable Matches for a Offer with Range $[5,10]$

Offer Price Range	Bid Price Range	Acceptable	Matched Value
$[3,\infty]$	$[0,4]$	Yes	3
$[5,\infty]$	$[0,3]$	No	-
$[2,\infty]$	$[0,2]$	Yes	2

Table 4.2: Table to show Acceptable Price Matches

4.3.3 Characteristic Matching Functions

We can now create our helper functions that calculate if the characteristics of a offer and bid are acceptable.

Algorithm 1: *calcQuantity* - A function to calculate if two quantity ranges are agreeable

```
Data: offer, bid
if bid.maxQuantity < offer.minQuantity
OR offer.maxQuantity < bid.minQuantity then
  | Return 0;
end
if offer.maxQuantity < bid.maxQuantity then
  | Return offer.maxQuantity;
else
  | Return bid.maxQuantity;
end
```

Algorithm 2: *calcPrice* - A function to calculate if two price ranges are agreeable

```
Data: offer, bid
if bid.maxPrice < offer.minPrice
OR offer.maxPrice < bid.minPrice then
  | Return 0;
end
if offer.minPrice < bid.minPrice then
  | Return bid.minPrice;
else
  | Return offer.minPrice;
end
```

4.4 Algorithm Design

Now we have established the format of our input and helper functions to calculate if the quantity and price of a bid and an offer are acceptable we can start to create the algorithm.

The first task we need to do is sort our input into two separate arrays, one of bids and one for offers. While we do this we also add some additional fields and data to our input which we will use later. If element is an offer we set the *minPrice* field to the value of the *price* field from the input and we set *maxPrice* to $+\infty$. If the element is a bid the *minPrice* field is set to 0 and we set *maxPrice* to the value of the *price* field from the input. To both we also add the *status* field which is initially set to *incomplete*. We use this field to check if a bid or offer has already been matched this round as we can skip over it if it has.

The rest of the algorithm is now fairly straight forward. Using a nested for loop we iterate over both the offers and bids, skipping over ones that have already been matched. We use our helper functions to calculate the agreement values, and if both of them return a value greater than 0 then our current bid and offer can be matched. We then update the status field of both the offer and bid to *complete* and save the calculated value for quanitity and price, along with all the other information we need to create a match agreement. Once the loops have exited we return our list of matches. An example of the algorithm output is shown in Listing 4.4 and the full psuedocode is shown in Algorithm 3. The full node.js module code can be found in Apprendix A.

The time complexity of the algorithm simple to calculate: In a worst case scenario the algorithm will loop b times for each o where $b =$ the number of bids and $o =$ the number offers. The complexity is therefore:

$$\mathcal{O}(b * o)$$

When the distribution of offers and bids in the input is 50-50 this reduces to the following with $n =$ the combined number of offers and bids:

$$\mathcal{O}((0.5 * n) * (0.5 * n)) \equiv \mathcal{O}(0.25 * n^2)$$

Our complexity can therefore be described as polynomial with order 2.

Listing 4.2: Example Algorithm Output

```
[
    {
        offerAddress: "0x123f681646d4a755815f9cb19e1acc8565a0c2ab",
        bidAddress: "0x5ed8cee6b53b1c6arce3ad7c92f4fd7e1b8fad9f",
        quantity: "13",
        price: "3"
    }
]
```

Algorithm 3: *Matchmaking*

```
Data: offersAndDemands - A list of offers and demands

Let offers = Empty List;
Let bids = Empty List;
Let matches = Empty List;
Let matchQuantity = 0;
Let matchPrice = 0;
for x ∈ offersAndDemands do
    if x.type=="offer" then
        offers.push({
            address: x.address,
            type: x.type,
            minQuantity: x.minQuantity,
            maxQuantity: x.maxQuantity,
            minPrice: x.price,
            maxPrice: +∞,
            status: "incomplete"});
    else
        bids.push({
            address: x.address,
            type: x.type,
            minQuantity: x.minQuantity,
            maxQuantity: x.maxQuantity,
            minPrice: 0,
            maxPrice: x.price,
            status: "incomplete"});
    end
end
for o ∈ offers do
    for b ∈ bids do
        if o.status=="complete" then
            Move to next o in offers;
        else
            if b.status=="complete" then
                Move to next b in bids;
            end
        end
        matchQuantity = calcQuantity(o, b);
        matchPrice = calcPrice(o, b);
        if matchQuantity > 0 AND bidPrice > 0 then
            o.status = "complete";
            b.status = "complete";
            matches.push({
                offerAddress: o.address,
                bidAddress: b.address,
                quantity: matchQuantity,
                price: matchPrice});
        end
    end
end
Return matches;
```

Chapter 5

Evaluation

5.1 Performance

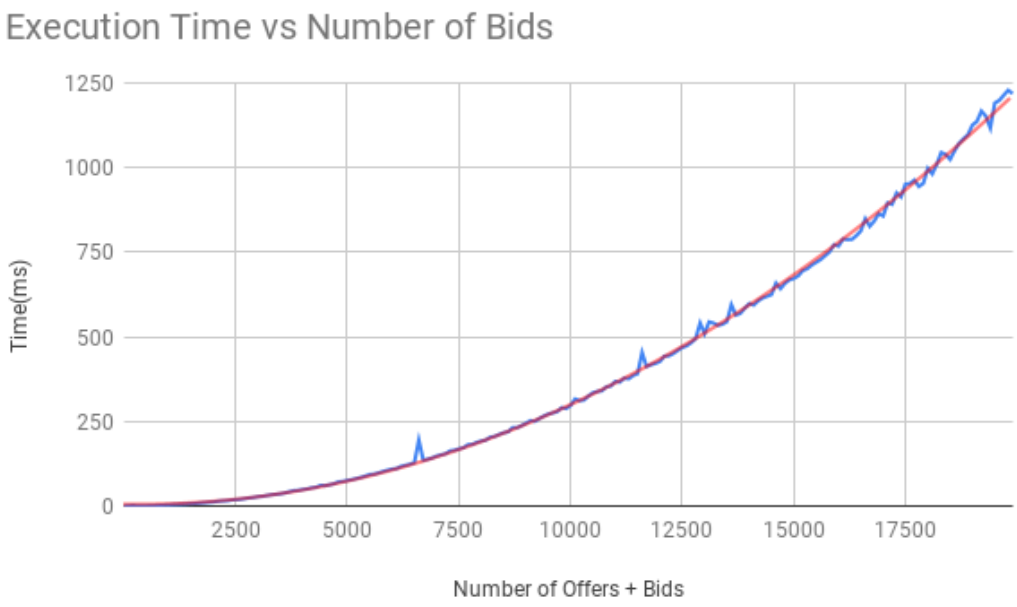


Figure 5.1: Plot of Algorithm Performance vs Number of Offers and Bids

To test the performance of our algorithm we generated test cases using the function in Appendix B. The function simply generates offers and bids with a 50-50 distribution with a random quantity between 0 - 1000 and a random price between 1-20. These tests are ran on a machine with a duel-core Intel i5 mobile CPU with a max clock speed of 3200 MHz and 4GB of RAM running Linux Mint 8.1. At the time of writing these specs are fairly below average.

The first test to run would be to see how execution time increases with the number of offers and bids. Test inputs were generated with the number of bids and offers ranging from 10-20000 in intervals of 100 and the execution time measured. The results can be seen in Figure 5.1. You can see that around 18000 offers and bids the execution time exceeds our limit of 1 second. However in reality the number of offers and bids might be several orders of magnitude below this number as we discuss in the following section, and as such the algorithm seems to be sufficient in this regard.

It would also be beneficial to evaluate the algorithms performance compared with an implementation that does not widen the values of quantity and price and instead matches only if the values are equal. The algorithm was modified slightly to the exact matching functions in Appendix B and tested with 10 inputs of various numbers of offers and bids with the number of matches made being recorded. The same technique and input was used on our algorithm and the results are displayed in Table 5.1.

	Number of Offers and Bids									
	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
A	3	13	24	44	70	125	132	175	236	287
B	24	88	201	292	430	568	665	841	1010	1178

Table 5.1: Number of matches for exact matching (A) and our implementation (B)

It is clear that our implementation creates far more matches than the exact matching and as such will create a higher throughput of bids and offers through the application leading to less expirations and a more responsive market.

5.2 Fit for Purpose

Our final algorithm meets the requirements set out for it in Chapter 3. It can successful handle an input of a list of offers and bids and provide as output a list of matches between suitable offers and demands, and if there are no matches it will output an empty list. From Figure 5.1 you can see that there must be over 18000 offers and demands before the algorithm takes more than a second to run, and realistically there will never be this number of offers and bids submitted even if the number of nodes is very large. We have also delivered the algorithm in a node.js module which can be found in Appendix A, to satisfy functional requirement 3, and this format also partly satisfies requirement 4 as the algorithm can easily be ran within the application without requiring any other technological requirements. We've also demonstrated that the algorithm can be ran on a low specification machine, proving that there are no excessive hardware requirements. This shows that any participant in the network should be able to execute the algorithm. Finally we have publicly published the source code with the rest of the application and detailed exactly how it works in this thesis to ensure full transparency.

Apart from meeting the requirements the algorithm seems suitable for its purpose. By drawing inspiration from existing real world examples such as the Price/Time algorithm used in financial markets around the world, we feel that by taking balanced approach during development we have indeed created an algorithm that works well without being too optimized towards any specific domain. Instead it strikes a nice balance of matching without too much restriction as to ensure speed, while not being so unrestricted that it would discourage participation in the market by users who want to get a good deal for their goods. This fits in with our generalized approach to the whole application, and helps in proving that the concept of a blockchain-based market is viable.

This algorithm does have some limitations however. We have discussed that it is only suitable for specific domains where quantity and price are the only characteristics. We have also made some assumptions about is what is optimal when matching. With quantity we value higher quantities and price we value lower quantities and while in some domains the should indeed be the case, there will definitely be some domains when it is not rendering our algorithm unsuitable.

Chapter 6

Conclusion

6.1 Summary

In conclusion we have looked into the factors that influence the development of a matchmaking algorithm for use in a blockchain-supported market and found that while there has been very little previous work in the blockchain-matchmaking domain, the information that we have found in the wider matchmaking domain is still relevant to blockchain. The key lesson that we have learnt is that the suitability of a matchmaking algorithm is also wholly dependant on the market in which it is to operate in and designing a good algorithm for a complex domain requires first modelling the market itself before making decisions as to what strategy is best to take in matchmaking design.

We have also successfully created a matchmaking algorithm that is suitable for our application. Using what we learnt from the related work we tried to create a generalized algorithm to match our generalized domain, and we have been successful in doing so. Our algorithm meets all the requirements set out for it and performs the task in hand efficiently even with inputs larger that it will ever have to handle.

6.2 Future Work

While the algorithm works and is appropriate for our application, as discussed in Chapter 2 the suitability of a matchmaking algorithm is mostly dependant on the domain of the market that it is to work in. Our implementation is suitable for a general market situation with goods that the only characteristics that matter are quantity and price. However as soon as more characteristics related to the goods are introduced, or the conditions that the market operates upon start to differ from the conditions we designed for, our algorithm quickly becomes unsuitable.

As such the next step in making our application more widely usable would be to offer the ability to tweak how the algorithm works, or change it out entirely. Due to the algorithms position within the application as a routine ran by the leader during block creation, it should be simple to offer a variety of algorithms which the parties that set up the application can chose between. It would also be possible to extend our solution so that its behavior is modifiable, for example it would be very easy to add an additional flag that would change the price matching so that it chooses a highest or average price that lies in the ranges of both the offer and the bid.

Appendix A

Matchmaking Algorithm node.js Module Code

Note than when this was written we had not standardized our use of terminology. Therefore in this code demand refers to a bid to buy, and a bid refers collectively to offers and demands

Matchmaking.js

```
//Made by Alex Daffurn-Lewis

/*
 * This function expects an array of bids and offers in the form:
 * [
 *   {address: address, type: type, minQuantity: minQuantity,
 *    maxQuantity: maxQuantity, price: price},
 *   ...
 * ]
 * Where type is either "offer" or "bid"
 * It will return an array of matched offers and demands in the form:
 * [
 *   {offerAddress: addressOfOffer, bidAddress: addressOfBid, quantity: quantity,
 *    price: price},
 *   ...
 * ]
 */
module.exports = function(bids) {
        var offers = new Array();
        var demands = new Array();
        var matches = new Array();
        var bidQuantity;
        var bidPrice;
        /*
         * If the bid is an offer we ask them for the minimum price they will
         * want for their goods, we then set their price range to [price, inf)
         * If the bid is a demand we ask them for the maximum price they are
         * willing to pay for the goods, we then set their price range
         * to [0, price]
         */
```

```
        for (var i = 0; i < bids.length; i++) {
                if(bids[i].type == "offer") {
                        offers.push({address: bids[i].address,
                          type: bids[i].type,
                          minQuantity: bids[i].minQuantity,
                          maxQuantity: bids[i].maxQuantity,
                          minPrice: bids[i].price,
                          maxPrice: Number.MAX_SAFE_INTEGER,
                          status: "incomplete"});
                } else {
                        demands.push({address: bids[i].address,
                          type: bids[i].type,
                          minQuantity: bids[i].minQuantity,
                          maxQuantity: bids[i].maxQuantity,
                          minPrice: 0, maxPrice: bids[i].price,
                          status: "incomplete"});
                }
        }
        for (var i = 0; i < offers.length; i++) {
                for (var j = 0; j < demands.length; j++) {
                        if(offers[i].status == "complete") {
                                break;
                        } else if (demands[j].status == "complete") {
                                continue;
                        }
                        bidQuantity = calcQuantity(offers[i], demands[j]);
                        bidPrice = calcPrice(offers[i], demands[j]);
                        if(bidQuantity > 0 && bidPrice > 0) {
                                updateStatus(offers[i]);
                                updateStatus(demands[j]);
                                matches.push({offerAddress: offers[i].address,
                                  bidAddress: demands[j].address,
                                  quantity: bidQuantity, price: bidPrice});
                        }
                }
        }
        return matches;
}
```

```
/*
 * Checks if the two ranges of both bids meet returning 0 if they don't.
 * If they do then it returns the HIGHEST value that lies in both of the ranges.
 */
function calcQuantity(offer, demand) {
        if(demand.maxQuantity < offer.minQuantity
        || offer.maxQuantity < demand.minQuantity) {
                return 0;
        }
        if (offer.maxQuantity < demand.maxQuantity) {
                return offer.maxQuantity;
        }
        return demand.maxQuantity;
}
/*
 * Checks if the two ranges of both bids meet returning 0 if they do not.
 * If they do then it returns the LOWEST value that lies in both of the ranges.
 */
function calcPrice(offer, demand) {
        if(demand.maxPrice < offer.minPrice
        || offer.maxPrice < demand.minPrice) {
                return 0;
        }
        if (offer.minPrice < demand.minPrice) {
                return demand.minPrice;
        }
        return offer.minPrice;
}

function updateStatus(bid) {
        bid.status = "complete";
}
```

Appendix B

Testing Functions

```
function calcQuantityExact(offer, demand) {
        if(demand.quantity == offer.quantity) {
                return demand.quantity;
        }
        return 0;
}
function calcPriceExact(offer, demand) {
        if(demand.price == offer.price) {
                return demand.price;
        }
        return 0;
}
```

Bibliography

Andrikopoulos, Vasilios et al. (2010). "QoS Contract Formation and Evolution". In: *International Conference on Electronic Commerce and Web Technologies*, pp. 119–130. url: https://link.springer.com/chapter/10.1007/978-3-642-15208-5_11.

Catalini, C (2017). "Blockchain, Explained". In: url: http://mitsloan.mit.edu/newsroom/articles/blockchain-explained/.

Janecek, Karel and Martin Kabrhel (2007). "Matching Algorithms of International Exchanges". In: url: https://pdfs.semanticscholar.org/6d92/0528fc5a3a25cb7a627b93ae3e7d5789bde8.pdf.

Jula, Amin, Elankovan Sundararajan, and Zalinda Othman (2014). "Cloud computing service composition: A systematic literature review". In: *Expert Systems with Applications* 41 (8), pp. 3809–3824. url: https://www.sciencedirect.com/science/article/pii/S0957417413009925.

Klein, Arne et al. (2008). "Evaluation of different feed-in tariff design options". In: url: http://www.feed-in-cooperation.org/wDefault_7/wDefault_7/download-files/research/best_practice_paper_2nd_edition_final.pdf.

Mengelkamp, Esther et al. (2018). "Decentralized blockchain-based electronic marketplaces". In: *Computer Science - Research and Development* 33 (1-2), pp. 207–214. url: https://link.springer.com/article/10.1007/s00450-017-0360-9.

Mihaylov, Mihail Emilov et al. (2014). "NRG-X-Change: A Novel Mechanism for Trading of Renewable Energy in Smart Grids". In: *3rd International Conference on Smart Grids and Green IT Systems*, pp. 101–106. url: https://www.researchgate.net/publication/287377465_NRG-X-change_a_novel_mechanism_for_trading_of_renewable_energy_in_smart_grids.

Nakamoto, Satoshi (2009). "Bitcoin: A Peer-to-Peer Electronic Cash System". In: p. 8. url: https://bitcoin.org/bitcoin.pdf.

Subramanian, Hemang (2018). "Decentralized blockchain-based electronic marketplaces". In: *Communications of the ACM* 61, pp. 78–84. url: https://dl.acm.org/citation.cfm?id=3158333.

Yu, Tao, Yue Zhang, and Kwei-Jay Lin (2007). "Efficient algorithms for Web services selection with end-to-end QoS constraints". In: *ACM Transactions on the Web* 1 (1). url: https://dl.acm.org/citation.cfm?id=1232728.