Dissertation Type: enterprise

DEPARTMENT OF COMPUTER SCIENCE

Extending the Bristol Stock Exchange with a Dark Pool

George Church

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Sunday 30th June, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

George Church, Sunday 30th June, 2019

Contents

List of Figures

List of Listings

Executive Summary

Most modern financial exchanges today are *lit pool* exchanges. This means that orders placed by traders are made publicly visible on a *Limit-Order-Book*. In recent decades, electronic financial exchanges have become heavily populated with automated trading algorithms. Extensive research has gone into the development of sophisticated automated trading algorithms with the goal of turning a profit. The *Bristol Stock Exchange* is a simulator of a lit pool exchange, written in Python, that facilitates experimentation with automated trading algorithms.

Market impact is the effect that a trader has on the market when they place an order to buy or sell a tradeable asset. When a trader places a large order, they can have a significant effect on the market which can result in the trader getting a bad deal for their trade. *Dark pool* exchanges were created as a way to avoid market impact. This is achieved by not making orders placed by traders publicly visible.

The aim of my project is to extend the Bristol Stock Exchange to offer a dark pool alongside the currently offered lit pool. This will allow market simulations to be run with automated trading algorithms in a coupled dark and lit pool.

During my project I have done the following:

- I researched the functionality of Turquoise Plato$^{\text{TM}}$: a dark pool trading venue offered by the London Stock Exchange.
- I created a minimal simulator of the Turquoise Plato trading venue. This includes a reputational scoring system that monitors trader behaviour. The simulator was written in Python.
- I ran example market simulations with the implemented dark pool.

Supporting Technologies

- I used the *Bristol Stock Exchange* (BSE), an open-source simulator of a LOB based financial exchange created by D. Cliff. BSE is publicly available on GitHub. The repository can be found online at https://github.com/davecliff/BristolStockExchange.
- I used the *Python* programming language (version 2.7) to implement the dark pool simulator. Unit tests were created with the *unittest* module.

Notation and Acronyms

BDN	:	Block Discovery Notification
BDS	:	Block Discovery Service
BI	:	Block Indication
BSE	:	Bristol Stock Exchange
CRS	:	Composite Reputational Score
ERS	:	Event Reputational Score
LOB	:	Limit-Order-Book
MES	:	Minimum Execution Size
MIV	:	Minimum Indication Value
MNV	:	Minimum Notification Value
OSR	:	Order Submission Request
QBO	:	Qualifying Block Order
RST	:	Reputational Score Threshold

Chapter 1

Contextual Background

1.1 Exchanges

Exchanges are venues where traders meet to buy and sell assets. Traders place offers to buy or sell some quantity of a tradeable asset for a specified price and hope to find counterparties that are willing to accept their offers and trade with them. Modern financial exchanges are electronic and highly automated with hundreds of thousands of trades occurring each day with billions of pounds worth of assets being traded. The London Stock Exchange reports that in April of 2019, their UK order book executed 16 million trades with £88 billion worth of assets traded [8]. A *Lit pool* exchange is an exchange where the offers made by traders to buy and sell are publicly displayed in a *Limit-Order-Book* (LOB). This is the most common type of exchange.

1.2 Automated trading algorithms

Only a few decades ago, financial exchanges were exclusively populated by human traders. With the advent of electronic exchanges came a new opportunity for making money. Automated trading algorithms were developed and deployed into these exchanges with the goal of making trades to gain as much profit as possible. These automated trading algorithms very quickly started to out-performed human traders. This is because the trading algorithms can react to trading opportunities much faster than their human counterparts. As a result, human traders have become a rare breed.

Many trading algorithms have been published in academic papers and are in the public domain. Financial institutions have simultaneously been developing their own automated trading algorithms and keep their designs as tightly kept secrets. Examples of some popular automated trading algorithms in the published literature are ZIP [1], Kaplan's Sniper [6] and AA [9].

1.3 The Bristol Stock Exchange

The Bristol Stock Exchange (BSE) is a open-source minimal simulator of a LOB based, lit pool exchange [3]. BSE is a platform for running market simulations with automated trading algorithms. BSE contains reference implementations of the trading algorithms mentioned in Section 1.2 as well as some others. BSE was created for the purpose of teaching masters-level students at the University of Bristol about algorithmic trading. BSE has since been used in a number of academic studies that have resulted in published work. For example, [7] makes use of BSE to demonstrate that the use of deep learning neural networks can mimic the behaviour of the ZIP trading algorithm.

1.4 Market impact

A trader may want to trade a very large quantity of a particular tradable asset. To do this, the trader can place a single, very large order onto a lit pool exchange. An order to trade a very large quantity is known as a *block order*. The block order placed by the trader will be visible on the LOB to all the other traders in the exchange. The arrival of a block order into the market is likely to cause the other traders to move their price of the tradable asset against the trader. This effect is known as *market impact*. Traders

placing block orders want to avoid market impact as this effect will result in them getting worse deals for their trades.

1.5 Dark Pools

Traders that trade in block orders are known as *block traders*. Block traders want to avoid market impact as much as possible. *Dark pool* exchanges were introduced to allow block traders to trade with each other without the negative effects of market impact.

The primary feature of a dark pool is that the LOB is not made visible to participating traders. This means that block traders can place block orders without their intentions to trade being disclosed to other traders. This avoids the effects of market impact. Trades that occur in dark pools will be disclosed after they happen. The trades that take place in dark pools are executed at the current price of the asset in the lit pool exchange.

1.6 Turquoise

Turquoise is a trading venue offered by the London Stock Exchange [5]. In April 2019, Turquoise executed 1.3 million trades with £16 billion worth of assets being traded [8]. Turquoise offers a LOB based, lit pool exchange called *Turquoise Lit*^{TM}. Turquoise also offers a coupled dark pool exchange called *Turquoise Plato*^{TM}. Turquoise Plato offers a novel service for block traders called Turquoise Plato Block Discovery. This service allows allows block traders to first identify if a counterparty can match their block order before actually submitting the block order. The Turquoise trading venue is described in detail in Section 2.6.1.

1.7 Project Description

The aim of this project is to create a dark pool simulator that is based on the the Turquoise Plato trading venue. The dark pool simulator is intended to be an extension to BSE. This will allow BSE to offer both a lit and dark pool simultaneously, thereby simulating the coupled trading venues of Turquoise Lit and Turquoise Plato. The simulator will allow for the study of automated trading algorithms in coupled dark/lit exchanges. I know of no other publications in the literature that describe such a simulator. There is also a lack of studies of automated trading strategies that perform in a dual lit/dark trading venue. The release of the code as open source on GitHub will provide a tool for further research and education.

1.8 Aims and Objectives

The high level objectives of this project are:

1. Creating a minimal simulator of a dark pool exchange which is based on the London Stock Exchange's Turquoise Plato trading venue.
2. Integrating the dark pool with BSE, thereby creating a simulator of a coupled lit and dark pool.
3. Running some market simulations with the implemented dark pool.

Chapter 2

Technical Background

2.1 Continuous Double Auctions

Exchanges are an example of the *continuous double auction* (CDA). A CDA takes place for each tradable asset. In a CDA, traders are free to place *orders* for the tradable asset whenever they like. An order is either a *bid* or an *ask*. A bid is an offer to buy a specified quantity at a specified price and an ask is an offer to sell a specified quantity at a specified price. Traders are free to accept orders whenever they like.

2.2 The Limit-Order-Book

A *limit order* is an order in which the specified price is the trader's price-limit. For a bid order, this means the highest price that the order can execute at. For a sell order, this means the lowest price that the order can execute at.

In *lit* exchanges, limit orders that have been submitted to the exchange but have not yet been executed are displayed in the Limit-Order-Book (LOB). There are two sides to the LOB: the bid side and the ask side. The bid side of the LOB displays information about the current bid orders and the ask side of the LOB displays information about the current ask orders. Each side of the LOB has a column for quantity and price. Orders that have the same limit price will be aggregated together to give an overall quantity for that price. For example, if there are currently two ask orders with a limit price of £120 awaiting execution, one with a quantity of 2 and another with a quantity of 4, then the LOB will display the price of £120 with a quantity of 6. The LOB indicates the current supply and demand for the tradable asset.

Both sides of the LOB are displayed in best-to-worst order from top-to-bottom. For the bid side, *best* means the orders with the highest price. This is considered the best price because it represents the best deal for counterparties that want to trade. This means that on the bid side of the LOB, the price is descending from top-to-bottom. For the ask side, *best* means the orders with the lowest price. So on the ask side of the LOB, the price is ascending from top-to-bottom. An example LOB is shown in Table 2.1.

Ask		Bid	
Quantity	Price (£)	Price (£)	Quantity
2	100	120	6
4	98	123	3
5	97	125	2
6	96	126	5
1	94	128	1
3	93	130	4
2	92	131	2

Table 2.1: An example LOB.

The LOB is split into the ask side and the bid side. The best ask order has a price of £100 and a quantity of 2. The best bid order has a price of £120 and a quantity of 6. The spread is equal to £20. The midprice is equal to £110. The microprice is equal to £105.

The *spread* is the difference between the best bid price and the best ask price on the LOB. For example, if the current best bid price is £95 and the current best ask price is £105, then the spread is

equal to £10. For an order to execute, the limit price of the order needs to cross the spread. For a bid order, this means that the price needs to be greater then the best ask order price on the LOB. For an ask order, this means that the price needs to be less than the best bid order price on the LOB.

A *market order* is an order that immediately executes and accepts whatever the best price is on the LOB at that moment in time. For a bid order, this means that the order is executed with the current lowest priced ask orders. For a ask order, this means that order is executed with the current highest priced bid orders.

2.2.1 The Midprice

The *midprice* can be thought of as the general market value of an asset. The midprice is given as the arithmetic mean of the best bid price and the best ask price currently on the LOB. If the best bid price is P_b and the best ask price is P_a, then the midprice P_m is given by the equation:

$$P_m = (P_a + P_b)/2$$

For example, if the current best bid price on the LOB is £22.50 and the current best ask price on the LOB is £27.50, then the current midprice is equal to

$$(£22.50 + £27.50)/2 = £25$$

.

2.2.2 The Microprice

The *microprice* is another way to value an asset. The microprice takes into account the limit prices of the best ask and best bid orders as well as their quantities. If the current best bid price and associated quantity are P_b and Q_b respectively, and the current best ask price and associated quantity are P_a and Q_b respectively, then the microprice P_μ is given by the equation:

$$P_\mu = P_a(Q_b/(Q_a + Q_b)) + P_b(Q_a/(Q_a + Q_b))$$

For example, if the best bid price on the LOB is £24 with a quantity of 14 and the best ask price on the LOB is £26 with a quantity of 2, then the microprice is equal to:

$$£26 \times (14/(2 + 14)) + £24 \times (2/(2 + 14)) = £25.75$$

2.3 Market Impact

A problem facing traders on conventional lit pool exchanges is market impact. Market impact can be described as the effect that placing a *block order* has on the market price. A *block order* is an order to trade a very large quantity of an asset. This is a big problem for *block traders*, who wish to trade in *block orders*. Since the LOB is visible on lit pool exchanges, all traders will be able to see when a block order is placed. Potential counterparties will move their price to a less favourable position as they anticipate the price of the tradable asset changing. This effect is due to the traders understanding about the laws of supply and demand. When there is excess supply in the market, then price of an asset is likely to move down. If there is excess demand in the market then the price of an asset is likely to move upwards.

As an example of market impact, say that there is a tradable asset with ticker symbol XYZ. Say that the normal quantity for trades of XYZ is 10 and that currently the midprice of XYZ is £10. Now say that a trader called Alice places an order to sell a quantity of 1000 units of XYZ at a price of £10.05. This definitely qualifies as a block order. Let us assume that Alice's order is not able to immediately execute. This means that Alice's potential counterparties that are placing bid orders are able to see this block order on the LOB. As a result, the bidders are likely to lower their prices. Lets say that the market impact effect causes the midprice of XYZ to move down to £7.00. This means that the price of XYZ will have taken a 30% drop before Alice's order has even been executed. As a result, Alice will not be very happy. One way that Alice could get around the effects of market impact is by slicing up her order up into many smaller orders, and then placing those individual orders into the lit pool over an extended period of time. This is likely to minimize the effects of market impact for Alice; however, this approach is not very efficient and each separate order will incur a processing fee.

Say that Alice wants to place an ask order of 1000 units of XYZ but this time she finds out that another trader called Bob wants to place an bid order of 1000 units of XYZ at around the same time. Alice and Bob decide to not put their orders into the lit exchange and instead trade directly with each other off exchange. Alice and Bob agree to trade at current midprice of XYZ on the lit exchange. Alice and Bob will not only avoid the problems of market impact, but will also get a better price than if they had placed a market order on the exchange. Alice and Bob will also avoid the order slicing method, and instead will execute their orders in one large trade. Bob and Alice notify the exchange of their trade after it is executed.

The example with Alice and Bob is what block traders would have had to do to avoid market impact before *dark pools* came along. Dark pools were originally created for these block traders to trade with each other anonymously while avoiding market impact.

2.4 Dark Pools

In lit pool exchanges, the LOB is made publicly available. Traders can use the information in the LOB to determine the prices that they want to place their orders at. In dark pool exchanges, the LOB is not made publicly available. This allows block traders to avoid market impact. The price that trades execute at is equal to the midprice of the asset in the lit pool. It is important to note that, even for small orders, trading on the dark pool will yield a better price than trading with a market order in the lit pool. Since the trade executes at the midprice, the price of the trade will be better by half of the spread.

2.5 The Bristol Stock Exchange

2.5.1 Introduction

The Bristol Stock Exchange is a open-source minimal simulator of a lit order book exchange created by D. Cliff [3]. It is written in Python, and the source code can be found online at the public GitHub repository available at `https://github.com/davecliff/BristolStockExchange`. The repository includes a guide [2] which thoroughly explains how BSE works. BSE is a platform that facilitates the simulation of market experiments with automated trading algorithms.

2.5.2 Limit-Order-Book and Tape

BSE contains a LOB for a single tradable asset. Orders that have been placed by the traders and which are still active will be visible on the LOB. The traders have access to the LOB data, and can use this information to determine the order prices that they wish the submit. An example LOB taken from BSE is shown in Figure 2.1.

Figure 2.1: An example LOB taken from BSE.
The LOB is displayed on the left hand side and the demand and supply curves are displayed on the right hand side . [3, p3].

BSE also produces a tape which contains details of all of the transactions that have taken place within the market simulation. For each transaction, the tape details the time the transaction occurred, the quantity of the trade and the price that the trade executed at. The tape is saved to a CSV file at the end of the simulation, and is useful for analysing results.

2.5.3 Automated Trading Algorithms

Automatic Execution Systems

The traders used in BSE fall into the class of traders known as *automatic execution systems*. This type of trader does the job that human traders were doing for many years before they were recently made redundant by machines. This type of trader receives a *customer order* specifying the quantity that a customer wants to buy/sell and a price that the customer wants for each unit. The trader is then tasked with carrying out the customer order whilst trying to maximize its own *margin* on the trade. The margin is the difference between the price specified in the customer order and the price that the trader manages to trade at.

For example, let's say that a trader receives a customer order specifying that they should buy a quantity of 1000 units of a tradeable asset at a limit price of £5 per unit. This means that the trader will be given £5000 to carry out this order. The trader wants to buy each unit at a price less than or equal to £5. The lower the price the trader manages to acquire the units, the better. Let's say that the trader manages to buy the 1000 units at a price of £4.75 each. This means that the trader spent £4750 on acquiring the 1000 units. In this case the margin for the trader would be $£5000 - £4750 = £250$. The trader gives the customer the units that they asked for and the trader is able keep a portion of the £250 as profit.

Similarly, let's say that a trader receives a customer order specifying that they should sell a quantity of 2000 units of a tradable asset at a price of £4.50 per unit. This means that the trader is given the 2000 units of the tradable asset and has to sell them for a total value of £9000. This means that the trader wants to sell each unit at a price greater than or equal to £4.50. The greater the price, the better. Let's say that the trader manages to sell the units at a price of £5 each. This means that the total price of sales is £10,000. In this case the margin for the trader is $£10,000 - £9000 = £1000$. The trader gives the £9000 back to the customer and keeps a portion of the £1000 as profit.

Each trader in BSE extends the `Trader` class. Each trader needs to create a function called `getorder()` which BSE uses to obtain an order from the trader. Within this function, the trader will specify the algorithm that they use to carry out customer orders. The traders will be given the latest information on the LOB and also the amount of time remaining in the market simulation. Each trader keeps track of how much money it has made from the margins of customer orders. This is the trader's profit. The profit of each type of trader is aggregated together and written to a CSV at the end of a simulation.

Example Implementations

BSE contains reference implementations of some automated trading algorithms. The simplest among these is the Giveaway (GVWY) trader. This trader simply quotes the same price as that which is specified in the customer order that it receives. This means that the trader doesn't hope to make any money from the customer order, but simply wants to make sure the trade goes through without making a loss.

BSE also contains implementations of some illustrative automated trading algorithms which have been published in the literature. One of these is the ZIP trading algorithm [1]. The ZIP (which stands for Zero Intelligence Plus) trader uses machine learning techniques in order to determine the price that it quotes at.

2.5.4 Market Simulations

BSE allows its users to perform market simulations with automated trading algorithms. The market simulations are performed using the `market_session()` function. For a market session, a user can specify the start time and end time, the traders to populate the market with and how the customer orders are scheduled.

Populating the market with traders is done by using the `populate_market()` function. Each trader is either a buyer or a seller. A user can select the amount of buyers and sellers of each type of trader that they want in the market simulation.

The scheduling of customer orders is performed using the `customer_orders()` function. You can schedule the supply (i.e. sell orders) and the demand (i.e. buy orders) separately. There are a few different scheduling modes that can be selected for the arrival of customer orders. You can also specify the price range that the customer orders will have. For more detail see the BSE guide [2].

2.6 Turquoise

The dark pool trading venue that I will be implementing is directly inspired by the Turquoise Trading Service [5]. Turquoise was founded in 2008 and is majority owned by the London Stock Exchange Group. Turquoise features a lit order book trading service called Turquoise Lit^{TM} and a dark order book trading service called Turquoise Plato^{TM}.

2.6.1 Turquoise Plato

Turquoise Plato is the the dark pool service offered by Turquoise. Orders that are submitted to Turquoise Plato are stored in the Turquoise Plato Order Book. The Turquoise Plato Order Book is not made visible to the users. All orders submitted to the Turquoise Plato Order Book will execute at the current midprice of the LOB on Turquoise Lit. All trades that take place within Turquoise Plato are published after they occur. Orders to buy are called *buy orders* and orders to sell are called *sell orders*. The buy orders rest on the *buy side* of the Order Book and the sell orders rest on the *sell side* of the Order Book.

Orders

All orders submitted to Turquoise Plato are executed at the midprice of the LOB on Turquoise Lit. Orders can still specify an optional limit price. For an order with a limit price to execute, the limit price has to be a better price than the current midprice on Turquoise Lit. For a buy order, this means that the limit price has to be greater than the current midprice on the lit pool. For a sell order, this means that the limit price has to be less than the current midprice on the lit pool. If no limit price is specified then the order is referred to as a market order.

Traders can specify a *Minimum Execution Size* (MES) on orders. The MES is the minimum quantity that can be executed in an order. An example will help to illustrate this. Say that a trader submits a buy order X with a quantity of 20 and an MES of 10. Say that another trader submits a sell order Y with quantity of 15 and an MES of 12. These orders can be matched for a trade since the MES of each order is larger than the other orders quantity. A quantity of 15 will be traded in this case. Say that instead of sell order Y, a sell order Z was submitted with a quantity of 9 and a MES of 5. In this case buy order X and sell order Z will not be matched for trading as the quantity of Z is less than the MES of X.

Traders can also specify the duration of an order. This specifies how long the order will remain on the order book, it at all. The duration can be set as Fill or Kill (FOK) which means that the order must be able to execute immediately and in full or it will not be executed at all and will not be added to the Order book. The duration can also be set as Fill and Kill (FAK), meaning that the order must be able to execute immediately either fully or partially. If it is only partially executed then the remaining quantity will not remain on the Order Book. The duration can instead specify an expiration time which will cause the order to rest on the Order Book until it is executed or the expiration time is reached. An order that lies on the Order Book until its execution or expiry is known as a *persistent* order. Orders that e execute immediately

Order Priority

Orders in the Turquoise Plato Order Book with higher quantity are given a higher priority when order matching is attempted. If two orders have the same quantity, then the time that the order arrived is used to determine the order with the higher priority. The order that arrived earlier is given the higher priority. An order maintains the priority of its initial quantity if it is only partially executed.

Continuous Matching Mode

When trying to match persistent orders, matching starts with the highest priority buy order. This means that the highest priority buy order is selected and then the sell orders are checked for a match in priority order from highest to lowest. If no match is found for this buy order then the next highest priority buy order is selected for matching with the sell orders. When a match is found between a buy order and a sell order then the order is executed and the highest priority buy order will be selected once again for matching. The whole process will be repeated until no more matches between order can be found.

2.6.2 Turquoise Plato Block Discovery

In this section, I will give a brief description of the Turquoise Plato Block Discovery service [4].

Overview

Turquoise Plato Block Discovery is used by traders that want to place block orders. Block Discovery allows a trader to first identify if there are any counterparties that can match their block order before they commit to submitting a firm order (a *firm order* is an order submitted into the Turquoise Plato Order Book). This is done by the trader first submitting a Block Indication (BI). A BI is an indication to the exchange that a trader wants to place a block order, but is not yet a firm order. Within the BI, the trader specifies the details of the block order that they would like to place. This includes the quantity, the limit price, the MES and the order duration. A trader's BI is attempted to match with another BI submitted by another trader. Once a match between two BIs is found, the traders that submitted the BIs are each sent an Order Submission Request (OSR) notifying them of the match. The OSRs ask the traders to convert their BIs into firm orders by sending a Qualifying Block Order (QBO). The QBO specifies the final details of the firm order that the trader is submitting. The details that a trader specifies in a QBO can be different than the original details specified in the BI. Each trader has a reputational score and if the details specified in the submitted QBO are different from the corresponding BI, then the trader's reputational score may be negatively affected. Once the QBOs are received from both traders, the firm orders specified in the QBOs are submitted to the Turquoise Plato Order Book. The matching of these orders and the execution of the trade will then take place. To summarise, the main steps that occur in the Block Discovery are:

1. A Trader submits a block indication.
2. If a match is found for that block indication, then the trader is sent an OSR to notify them.
3. The trader places a firm order by submitting a QBO.
4. The firm order is added to the Order Book and the trade takes place.
5. The trader's reputational score is updated.

A diagram depicting the entire process just described is shown in Figure 2.2.

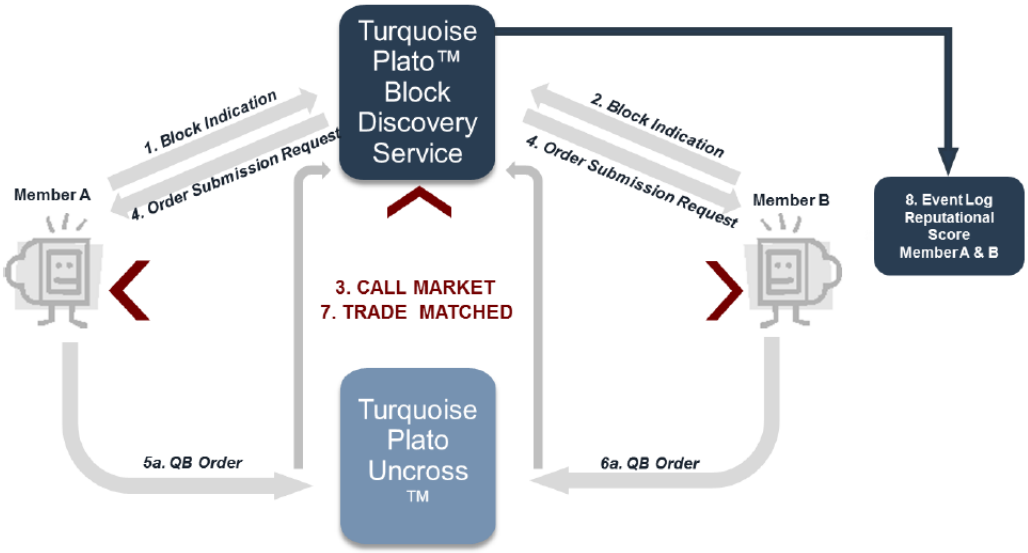

Figure 2.2: A diagram depicting the steps involved in the Block Discovery service when two BIs are matched together.

This diagram is taken from the Turquoise Plato Block Discovery document [4, p18]. A Turquoise Plato Uncross event is triggered when final the QBOs are submitted. This causes the trade to occur. The call market is a message sent to traders notifying them of an impending Turquoise Plato Uncross event.

Block Indications

A Block Indication is not a firm order, but an indication to the Block Discovery service that the trader would like to place a block order. The trader first wants to see if a counterparty can match their order.

The BI contains details about the firm order that the trader will place. The details include things like the quantity, the limit price, the MES and the order duration. The limit price and the quantity are optional and can be left out. The quantity specified in the BI has to be greater than the Minimum Indication Value (MIV) for that tradeable asset. If the quantity is not greater than the MIV, then the BI will be rejected.

Whenever a BI is added to the exchange, a check is performed to see if the BI can match with any of the other BIs already submitted to the service. If the BI specifies an MES and/or a limit price, then the check needs to make sure that these properties match between the orders and can produce a possible trade. Once a match is found, then each trader is sent an OSR to notify them.

Order Submission Requests

An Order Submission Request is sent to a trader to notify them when a match is found for a BI that they submitted. The OSR does not include any details about the counterparty or the counterparty's BI that was matched. The OSR is also used to notify the trader of their composite reputational score. Reputational scoring is described in detail in Section 2.6.2. Upon receiving a OSR, a trader is expected to respond with a QBO to confirm their BI.

Qualifying Block Orders

A Qualifying Block Order is sent by a trader to the exchange upon receipt of an OSR. The OSR tells the trader that their BI has been matched with another BI on the exchange. The QBO is a confirmation of the trader's BI: turning their indication into a firm order. The details specified in the QBO may be different from the details specified in the original BI. For example, the QBO could specify a smaller quantity or a higher limit price. The differing details between a QBO and the corresponding BI will affect the trader's reputational score. This is discussed in detail in Section 2.6.2. If a QBO's details are sufficiently different from the original BI, then this may mean that a trade is impossible. This will have a negative effect on the trader's reputational score. Once the QBOs are recieved from both traders, the firm orders are added to the Orderbook and the trade can occur.

Reputational Scoring

The Block Discovery service monitors the traders' conversion from BIs to QBOs with a reputational scoring mechanism. A trader is given an event reputational score every time they convert a BI into a QBO.

A QBO should to be *marketable* in comparison to its corresponding BI. In relation to price, A QBO is marketable if it meets any of the following:

- The BI does not specify a limit price and the QBO does not specify a limit price.
- The BI specifies a limit price and the QBO does not specify a limit price.
- The BI specifies a limit price and the QBO specifies a more marketable limit price (If we are dealing with a buy order, then the QBO limit price needs to be greater than the BI limit price. If we are dealing with a sell order, then the QBO limit price needs to be less than the BI limit price).

In relation to the MES, a QBO is marketable if it meets any of the following:

- The BI does not specify an MES and the QBO does not specify an MES.
- The BI specifies an MES and the QBO does not specify an MES.
- The BI specifies an MES and the QBO specifies an MES that is less than or equal to the BI's MES.

If a QBO is not marketable, then the event reputational score given to the trader is zero. If the QBO is marketable, then the event reputational score is calculated by taking into account the size of the BI and the size of the QBO. the resulting score will be between 50 and 100. The exact formula used to calculate the score in this case is not explicitly specified.

A trader's composite reputational score is calculated from their last 50 event reputational scores. This is done by using weighted factors for each event reputational score, with more recent event reputational scores having higher weights, i.e. 'The most recent event has a weighting of 50, the next most recent 49, and so on' [4, p23]. A trader is notified of their composite reputational score in every OSR that they

receive. If a trader's composite reputational score falls below the Reputational Score Threshold, then the trader will no longer be allowed to use the Block Discovery service. A trader's composite reputational score is persistent from one day to the next.

Block Discovery Notification

When a trader submits an order into the Turquoise Plato Orderbook, they can specify whether the order is a Block Discovery Notification (BDN). If an order is a BDN, then the order is eligible for participation in the Block Discovery Service. This means that this order may end up getting matched with a BI. An order can only be a BDN if the quantity of the order is greater than the Minimum Notification Value for the tradable asset. If the order is matched with a BI, then the trader that submitted the BI is notified with an OSR and they are asked to submit a firm order with a QBO. The trader that submitted the BI is not made aware that they have been matched with a BDN order.

Chapter 3

Implementation

BSE is open-source and publicly available on GitHub. The repository can be found at https://github.com/davecliff/BristolStockExchange. At the time I began my project, the latest version of BSE was the commit on July 22, 2018. I forked the repository and modified the original code in order to implement the dark pool. The GitHub repository for the dark pool is publicly available and be found online at https://github.com/gchurch/DarkPool. The source-code for the dark pool can be found in the `dark_pool.py` file.

I found two documents on the Turquoise website that were very useful for my implementation process. These documents are titled Turquoise Trading Service Description [5] and Turquoise Plato Block Discovery [5]. Although very descriptive and helpful, these documents did leave some gaps on how the Turquoise Plato service works which led to me struggling to understand some of the functionality. Nevertheless, I managed to produce a minimal implementation of the Turquoise Plato Order Book and the Turquoise Plato Block Discovery service.

3.1 Orders

In the original BSE code, there is a class called `Order` which is used to hold information about orders that are submitted to the exchange. This information includes the ID of the trader that submitted the order, the type of order (bid or ask), the limit price, the quantity and the time that the order was created.

A member variable called `MES` was added to the `Order` class so that orders can specify a minimum execution size. As explained in Section 2.6.1, the MES is the minimum quantity that can be executed for an order. The MES value should be less than or equal to the quantity value of the order.

In the original BSE code, orders could only have a quantity of one. The dark pool that was implemented allows orders to have a quantity of more than one. A member variable called `quantity_remaining` was added to the `Order` class since orders can partially execute. This member variable contains the quantity remaining on the order. The pre-existing `quantity` member variable is used to hold the original quantity of the order. The original quantity of the order is needed for prioritisation in order matching.

3.2 Order Book

The original BSE code contains the `Orderbook_half` class that is used to hold one half of the order book i.e. one instance of the class would act as the bid side of the order book and another instance of the class would act as the sell side of the order book. In the dark pool, orders to buy are referred to as *buy orders* and orders to sell are called *sell orders*. The half of the order book that contains the buy orders is referred to as the *buy side* of the order book. Similarly, the half of the order book that contains the sell orders is referred to as the *sell side* of the order book.

The `Orderbook_half` class was modified so that orders that are added to an instance of the class are stored in priority order. The orders are stored in a Python list data structure called `orders`. The order prioritization that is used is outlined in Section 2.6.1. Whenever a new order is added to the class using the `book_add()` method, the correct position to insert the order into the `orders` list is first calculated. This position is calculated by the `find_order_position()` method. This function returns the position in the `orders` list that this order should be added to so that order priority is maintained.

The `Orderbook` class holds each side of the order book in a `Orderbook_half` member variable. These are named `buy_side` and `sell_side`.

3.2.1 Matching Orders

The `Orderbook` class contains a method called `check_match()`. This method takes a buy order, a sell order and a price (which would normally be the midprice of the lit pool) as arguments, and determines whether the two orders can result in a trade. The method does this by making use of two other methods in the `Orderbook` class called `check_price_match()` and `check_size_match()`.

`check_price_match()` takes a buy order, a sell order and a price as arguments and determines whether or not the two orders can result in a transaction. The buy order's limit price needs to be greater than the sell order's limit price and the price needs to be between these two values. Orders can be placed without a limit price by specifying the limit price as `None`. When the limit price is set to `None`, the order will accept whatever value the price is. The code for the `check_price_match()` method is shown in Listing 3.1.

```
def check_price_match(self, buy_side, sell_side, price):

    if buy_side.limit_price == None and sell_side.limit_price == None:
        return True

    elif buy_side.limit_price != None and sell_side.limit_price == None:
        if buy_side.limit_price >= price:
            return True

    elif buy_side.limit_price == None and sell_side.limit_price != None:
        if sell_side.limit_price <= price:
            return True

    elif buy_side.limit_price != None and sell_side.limit_price != None:
        if buy_side.limit_price >= price and sell_side.limit_price <= price:
            return True

    return False
```

Listing 3.1: Code for the `check_price_match()` method in the `Orderbook` class.

Similarly, the `check_size_match()` method determines whether a buy order and a sell order can result in a transaction based on their quantities. This check is important because orders can optionally specify an MES. If both orders specify an MES, then both orders have to have a quantity remaining that is greater than the opposite order's MES in order for a transaction to occur. An order can optionally not specify an MES by setting the `MES` member variable to `None`. The code for the `check_price_match()` method is shown in Listing 3.2.

```
def check_size_match(self, buy_side, sell_side):

    if buy_side.MES == None and sell_side.MES == None:
        return True

    elif buy_side.MES != None and sell_side.MES == None:
        if sell_side.quantity_remaining >= buy_side.MES:
            return True

    elif buy_side.MES == None and sell_side.MES != None:
        if buy_side.quantity_remaining >= sell_side.MES:
            return True

    elif buy_side.MES != None and sell_side.MES != None:
        if buy_side.quantity_remaining >= sell_side.MES and \
        sell_side.quantity_remaining >= buy_side.MES:
            return True
```

```
18
19      return False
```

Listing 3.2: Code for the `check_size_match()` method in the `Orderbook` class.

The `check_match()` method checks that both the `check_price_match()` and `check_size_match()` functions return `True`, otherwise it will return `False`. The code for this method is shown in Listing 3.3.

```
# check that both the order size and the price match between the two given orders
def check_match(self, buy_side, sell_side, price):
    return self.check_price_match(buy_side, sell_side, price) and \
    self.check_size_match(buy_side, sell_side)
```

Listing 3.3: Code for the `check_match()` method in the `Orderbook` class.

The `find_matching_orders()` method attempts to find an order on the buy side of the order book and an order on the sell side of the order book that can result in a transaction. The method starts with the highest priority buy order and searches through all sell orders from the highest priority to the lowest priority. If no match is found then the next highest priority buy order is tried and so on. As described in Section 3.2, the orders in each side of the order book are already ordered in terms of their priority. The `find_matching_orders()` method uses the `check_match()` method in order to determine if two orders can result in a transaction. If a match is found, then a dictionary is returned containing information about the buy order, the sell order, the trade size that can be transacted and the price. If no match is found, then `None` is returned. The code for this method is shown in Listing 3.4.

```
# match two orders and perform the trade
def find_matching_orders(self, price):

    # get the buy orders and sell orders
    buy_orders = self.buy_side.get_orders()
    sell_orders = self.sell_side.get_orders()

    # matching is buy-side friendly, so start with buys first
    for buy_order in buy_orders:
        for sell_order in sell_orders:
            # find two matching orders in the order_book list
            if self.check_match(buy_order, sell_order, price):
                # calculate the trade size
                if buy_order.quantity_remaining >= sell_order.quantity_remaining:
                    trade_size = sell_order.quantity_remaining
                else:
                    trade_size = buy_order.quantity_remaining

                # return a dictionary containing the match info
                return {
                    "buy_order": buy_order,
                    "sell_order": sell_order,
                    "trade_size": trade_size,
                    "price": price
                }

    # if no match can be found, return None
    return None
```

Listing 3.4: Code for the `find_matching_orders()` method in the `Orderbook` class.

3.2.2 Executing Trades

The `execute_trade()` method executes a trade based on the information that is provided in the `trade_info` parameter. The argument will be a dictionary that is returned from a call of the `find_matching_`

`orders()` method. This method first removes the given orders from the order book. It then subtracts the trade size from each of the order's `quantity_remaining` member variable. If there is any leftover quantity remaining on either of the orders, then that order is re-added to the order book. If the MES of the order is now greater than the quantity remaining on the order, then the MES will be set to the quantity remaining. A record of the transaction will be created containing the time, the quantity, the price, the identity of the buyer and the identity of the seller. This record is added to the `Orderbook`'s `tape` member variable. This member variable is a list and contains a record of all transactions that have occurred. The code for the `execute_trade()` method is shown in Listing 3.5.

```
# Execute the trade specified in the trade_info parameter
def execute_trade(self, time, trade_info):

    # subtract the trade quantity from the orders' quantity remaining
    trade_info["buy_order"].quantity_remaining -= trade_info["trade_size"]
    trade_info["sell_order"].quantity_remaining -= trade_info["trade_size"]

    # remove orders from the order_book
    self.buy_side.book_del(trade_info["buy_order"].trader_id)
    self.sell_side.book_del(trade_info["sell_order"].trader_id)

    # re-add the residual
    if trade_info["buy_order"].quantity_remaining > 0:
        # update the MES if necessary
        if trade_info["buy_order"].MES > trade_info["buy_order"].quantity_remaining:
            trade_info["buy_order"].MES = trade_info["buy_order"].quantity_remaining
        # add the order to the order_book list
        self.buy_side.book_add(trade_info["buy_order"])

    # re-add the residual
    if trade_info["sell_order"].quantity_remaining > 0:
        # update the MES if necessary
        if trade_info["sell_order"].MES > trade_info["sell_order"].quantity_remaining:
            trade_info["sell_order"].MES = trade_info["sell_order"].quantity_remaining
        # add the order to the order_book list
        self.sell_side.book_add(trade_info["sell_order"])

    # create a record of the transaction to the tape
    transaction_record = {
        'type': 'Trade',
        'time':       time,
        'price': trade_info["price"],
        'quantity': trade_info["trade_size"],
        'buyer': trade_info["buy_order"].trader_id,
        'seller': trade_info["sell_order"].trader_id,
        'BDS': trade_info["buy_order"].BDS and trade_info["sell_order"].BDS
    }

    # add a record to the tape
    self.tape.append(transaction_record)

    # return the transaction
    return transaction_record
```

Listing 3.5: Code for the `execute_trade()` method in the `Orderbook` class.

The `execute_trades()` method keeps on finding matches between buy orders and sell orders (using the `find_matching_orders()` function) and executing transactions (using the `execute_trades()` function) until no more matches between orders can be found and so no more transactions are possible. The code for this method is shown in Listing 3.6.

```
# keep making trades out of matched orders until no more matches can be found
def execute_trades(self, time, price):

    # a list of all the trades made in this function call
    trades = []

    # find a match between a buy order a sell order
    match_info = self.find_matching_orders(price)
```

```
9
10      # keep on going until no more matches can be found
11      while match_info != None:
12
13          # execute the trade with the matched orders
14          trades.append(self.execute_trade(time, match_info))
15
16          # find another match
17          match_info = self.find_matching_orders(price)
18
19      # return the list of trades
20      return trades
```

Listing 3.6: Code for the `execute_trades()` method in the `Orderbook` class.

3.3 Block Discovery Service

In this section, I will be describing my implementation of a block discovery service based on the Turquoise Plato Block Discovery as described in Section 2.6.2. BIs to buy are referred to as *buy BIs* and BIs to sell are referred to as *sell BIs*.

A class called `Block_Indication` was created for block indications. This class is very similar to the `Order` class. Within this class are member variables for the time the BI was created, the trader id that created the BI, the type of the BI (buy or sell), the quantity, the limit price and the MES.

The `Block_Indication_Book` class was created to manage the block discovery service. This class has two member variables of the `Orderbook_half` class, one to store the buy BIs and the other to store the sell BIs. The priority of block indications for matching is the same as for orders in the order book. A trader can add BIs to the `Block_Indication_Book` with the `add_block_indication()` method, which takes a `Block_Indication` object as a parameter. The quantity stated in the BI has to greater than the Minimum Indication Value (MIV) in order to be accepted. The MIV is set in the `MIV` member variable of the `Block_Indication_Book` class. If the quantity of the BI is not greater than the MIV, then the BI is rejected. A trader must also have a composite reputational score greater than the Reputational Score Threshold (RST) in order for their BI to be accpepted. The RST is set in the `RST` member variable of the `Block_Indication_Book` class. A trader can only have one block indication or order placed in the exchange at any one point in time. This means that when a trader adds a new block indication, any order or block indication placed by that trader that is currently on the exchange will be removed. BIs can be deleted from the `Block_Indication_Book` with the `del_block_indication()` method.

3.3.1 Matching Block Indications

The `Block_Indication_Book` class contains a method called `find_matching_block_indications()`. This method tries to find matching block indications from either side of the `Block_Indication_Book`. The BIs on each side of the `Block_Indication_Book` are stored in priority order due to the use of the `Orderbook_half` class. The `find_matching_block_indications()` method starts with the highest priority buy BI and looks through all of the sell BIs from highest priority to lowest priority looking for a match. If no match is found for this buy BI then the next highest priority buy BI is used and the process is repeated until a match is found.

To check whether a buy BI and a sell BI match, the `check_match()` method is used. This method takes a buy BI, a sell BI and a price as arguments. For a buy BI and a sell BI to match, they need to match in terms of both their limit price and their size. To check whether the limit prices match, the `check_price_match()` method is used. The price argument is meant to be the current midprice on the lit exchange's LOB. The logic for this function is the same as for the `Orderbook` class's `check_price_match()` method as seen in 3.1. To check whether the BIs match in terms of their size, the `check_size_match()` method is used. The logic for this method is very similar to that of the `Orderbook` class's `check_size_match()` function as seen in Listing 3.2. The only difference being that in the `Block_Indication_Book`, the original quantity of each BI is checked instead of the quantities remaining on the orders.

If a match is found between a buy BI and a sell BI, then an item is added to the `matches` dictionary member variable of the `Block_Indication_Book` class. Each match is given a unique match ID which is used as the key in the dictionary. The dictionary value is a dictionary itself which contains the buy

BI and the sell BI. The buy QBO and sell QBO will be added to this dictionary when they are received. The code is for the `find_matching_block_indications()` method is shown in Listing 3.7.

```
# keep making trades out of matched orders until no more matches can be found
# attempt to find two matching block indications
def find_matching_block_indications(self, price):

    # get the buy side orders and sell side orders
    buy_side_BIs = self.buy_side.get_orders()
    sell_side_BIs = self.sell_side.get_orders()

    # starting with the buy side first
    for buy_side_BI in buy_side_BIs:
        for sell_side_BI in sell_side_BIs:
            # check if the two block indications match
            if self.check_match(buy_side_BI, sell_side_BI, price):

                # Add the matched BIs to the matches dictionary
                self.matches[self.match_id] = {
                    "buy_side_BI": buy_side_BI,
                    "sell_side_BI": sell_side_BI,
                    "buy_side_QBO": None,
                    "sell_side_QBO": None
                }
                # get the current match id
                match_id = self.match_id

                # increment the book's match_id counter
                self.match_id += 1

                # delete these block indications from the book
                self.del_block_indication(0, buy_side_BI, False)
                self.del_block_indication(0, sell_side_BI, False)

                # return the match id
                return match_id
    # if no match was found then return None
    return None
```

Listing 3.7: Code for the `find_matching_block_indications()` method in the `Block_Indication_Book` class.

The `Block_Indication_Book` class also has a method called `find_all_matching_block_indicaions()` which repeatedly calls the `find_matching_block_indications()` method until no more matches between BIs are found.

3.3.2 Notifying Traders

Once a match is found, an OSR needs to be sent to each of the traders. A class called `Order_Submission_Request` was created to simulate OSRs. This class contains the same member variables as specified in the `Block_Indication` class as well as a member variable called `match_id` to contain the unique match ID of the block indications and a member variable called `reputational_score` to hold the composite reputational score of the trader that the OSR is being sent to. The OSRs are created using the `create_order_submission_requests()` method of the `Block_Indication_Book` class. This method returns a dictionary containing both the OSR for the trader that is buying and the OSR for the trader that is selling. The code for this method is shown is Listing 3.8.

```
def create_order_submission_requests(self, match_id):
    # Get the block indications.
    buy_side_BI = self.matches[match_id]["buy_side_BI"]
    sell_side_BI = self.matches[match_id]["sell_side_BI"]

```

```
    # Get the composite reputational scores.
    buy_side_CRP = self.composite_reputational_scores[buy_side_BI.trader_id]
    sell_side_CRP = self.composite_reputational_scores[sell_side_BI.trader_id]

    # create the buy side OSR
    buy_side_OSR = Order_Submission_Request(self.OSR_id,
                                            buy_side_BI.time,
                                            buy_side_BI.trader_id,
                                            buy_side_BI.otype,
                                            buy_side_BI.quantity,
                                            buy_side_BI.limit_price,
                                            buy_side_BI.MES,
                                            match_id,
                                            buy_side_CRP)

    # increment the OSR id counter
    self.OSR_id += 1

    # create the sell side OSR
    sell_side_OSR = Order_Submission_Request(self.OSR_id,
                                            sell_side_BI.time,
                                            sell_side_BI.trader_id,
                                            sell_side_BI.otype,
                                            sell_side_BI.quantity,
                                            sell_side_BI.limit_price,
                                            sell_side_BI.MES,
                                            match_id,
                                            sell_side_CRP)

    # increment the OSR id counter
    self.OSR_id += 1

    # return both OSRs in a dicitionary
    return {"buy_side_OSR": buy_side_OSR, "sell_side_OSR": sell_side_OSR}
```

Listing 3.8: Code for the `create_order_submission_requests()` method in the `Block_Indication_Book` class.

3.4 Reputational Scoring

Reputational scoring is implemented as described in Section 2.6.2.

3.4.1 Event Reputational Scoring

Every time a trader submits a BI and a subsequent QBO after a match has been found, an event reputational score is calculated for that trader. The event reputational score is calculated by a method called `calculate_event_reputational_score()`. This method takes a `Block_Indication` object and a `Qualifying_Block_Order` object as parameters. Each trader's 50 most recent event reputational scores are stored in the `event_reputational_scores` dictionary member variable. Initially, each trader's 50 event reputational scores are set to a default value defined in the `initial_reputational_score` member variable.

The `calculate_event_reputational_score()` method checks whether the QBO is marketable. The concept of a QBO being marketable is outlined in Section 2.6.2. The `marketable()` method is used to determine whether or not a QBO is marketable. Firstly, a QBO needs to be marketable in relation to its price. This is checked by calling the `marketable_price()` method. The code for the `marketable_price()` method is shown in Listing 3.9.

```
def marketable_price(self, BI, QBO):

    if BI.limit_price == None and QBO.limit_price == None:
```

```
        return True

    elif BI.limit_price != None and QBO.limit_price == None:
        return True

    elif BI.limit_price != None and QBO.limit_price != None:
        if BI.otype == 'Buy':
            if QBO.limit_price >= BI.limit_price:
                return True
        if BI.otype == 'Sell':
            if QBO.limit_price <= BI.limit_price:
                return True
    else:
        return False
```

Listing 3.9: Code for the `marketable_price()` method in the `Block_Indication_Bookclass`.

Secondly, a QBO needs to be marketable in relation to its size. This is checked by calling the `marketable_size()` method. The code for the `marketable_size()` method is shown in Listing 3.10.

```
def marketable_size(self, BI, QBO):

    if BI.MES == None and QBO.MES == None:
        return True

    elif BI.MES != None and QBO.MES == None:
        return True

    elif BI.MES != None and QBO.MES != None:
        if QBO.MES <= BI.MES:
            return True

    return False
```

Listing 3.10: Code for the `marketable_size()` method in the `Block_Indication_Book` class.

Finally, The code for the `marketable()` method is shown in Listing 3.11.

```
def marketable(self, BI, QBO):
    return (self.marketable_price(BI, QBO) and self.marketable_size(BI, QBO))
```

Listing 3.11: Code for the `marketable()` method in the `Block_Indication_Book` class.

As outlined in Section 2.6.2, if a QBO is not marketable, then an event reputational score of 0 should be given to the trader. If a QBO is marketable, then an event reputational score of between 50 and 100 should be given to the trader. The algorithm used to calculate the event reputational score in the case where the QBO is marketable is not specified in the Turquoise Plato Block Discovery document [4]; therefore, I had to come up with my algorithm.

If the quantity specified in the QBO is greater than or equal to the quantity specified in the BI, then the trader should be given the maximum score of 100. If the quantity specified in the QBO is less than the quantity specified in the BI, then the trader should be given a score of 100 minus a value based on the percentage difference between the quantity specified in the QBO and the quantity specified in the BI. I wanted traders to get the minimum score of 50 if the quantity specified in the QBO is less than or equal to half of the quantity specified in the BI. I also wanted to the score to be exponentially worse as the percentage difference becomes larger. I found that using e as the growth rate for this was acceptable. So if the percentage difference is denoted by x, then the event reputational score ERS is given by:

$$ERS = 100 - 77.1 \times (e^x - 1)$$

The implementation of this algorithm is within the `calculate_event_reputational_score()` method and is shown in Listing 3.12.

```
# Calculate the reputational score of a trader for a single event. If the QBO is
# not marketable, then a score of 0 is given. If the QBO is marketable, then a
# score between 50 and 100 is given.
def calculate_event_reputational_score(self, BI, QBO):

    # check that the QBO is marketable
    if self.marketable(BI, QBO):

        # initial score of 100
        event_reputational_score = 100

        # calculate the difference between the QBO quantity and the BI quantity
        quantity_dec_diff = (BI.quantity - QBO.quantity) / BI.quantity

        # if the QBO quantity is less than the BI quantity then decrease score
        if quantity_dec_diff > 0:
            event_reputational_score -= round(77.1 * (math.exp(quantity_dec_diff) - 1))

        # make sure that the score is not less than 50
        if event_reputational_score < 50:
            event_reputational_score = 50

    # if the QBO is not marketable then the score is 0
    else:
        event_reputational_score = 0

   # add the event reputational score
    self.event_reputational_scores[BI.trader_id].insert(0,event_reputational_score)
    self.event_reputational_scores[BI.trader_id] = self.event_reputational_scores[BI.
    trader_id][:50]

    # return the score
    return event_reputational_score
```

Listing 3.12: Code for the `calculate_event_reputational_score()` method in the `Block_Indication_Book` class.

For example, say that a trader submits a buy BI with a quantity of 1000, an MES of 200, and a limit price of £2.50 per unit. This buy BI is then matched with a sell BI on the exchange. The exchange sends an OSR to the trader and the trader responds with a QBO. Inside the QBO, the trader specified a quantity of 900, an MES of 200 and a limit price of £2.50 per unit. The `caculate_event_reputational_score()` method is then used to calculate the event reputational score. The QBO is marketable since it has the same MES and limit price as specified in the original BI. The percentage difference between the quantity specified in QBO and the quantity specified in the BI will be calculated as $100 \times (1000-900)/1000 = 10\%$ or 0.1 in decimal form. Next, the event reputational score will be calculated as $100 - 77.1 \times (e^{0.1} - 1) = 92$ (rounded to the nearest integer). This score will be added to the list of the traders 50 most recent event reputational scores in the `event_reputational_scores()` dictionary in the `Block_Indication_Book` class.

3.4.2 Composite Reputational Score

A trader's 50 most recent event reputational scores are used to calculate their composite reputational score. The composite reputational score is calculated in the `calculate_composite_reputational_score()` method. This method takes a trader's unique ID as an argument. The algorithm that is used in this method is that which is outline in Section 2.6.2.

The code for the `calculate_composite_reputational_score()` method is shown in Listing 3.13. A weighting is applied to each of the trader's event reputational scores. The most recent event reputational score is given a weighting of 50, the next most recent is given a weighting of 49 and so on. The sum of all of the event reputational scores multiplied by their weighting is stored in the `total` variable. The final value in the `total` value is divided by 1275 (since $\sum_{n=1}^{50} n = 1275$) and then rounded to the nearest integer. This is the composite reputational score of the trader and is returned by the method. This value will be between 0 and 100.

```
# Calculate a trader's composite reputational score.
# The most recent event reputational score has a weighting of 50,
# the next most recent 49, and so on
```

```
def calculate_composite_reputational_score(self, tid):

    # The current weighting
    weighting = 50
    # The sum of the event reputational scores multiplied by their weighting
    total = 0.0

    for i in range(0, 50):
        total += weighting * self.event_reputational_scores[tid][i]
        weighting -= 1

    # Calculate the composite reputational score rounded to the nearest integer
    composite_reputational_score = int(round(total / 1275))

    # return the composite reputational score
    return composite_reputational_score
```

Listing 3.13: Code for the `calculate_composite_reputational_score()` method in the `Block_Indication_Book` class.

As example caculation of the composite reputational score, say that a trader has used the block discovery service a total of 5 times so far and has received event reputational scores of 100, 85, 0, 50 and 100 (ordered from most recent to least recent). The initial value given to their remaining event reputational scores is 70. The trader's composite reputational score will be calculated as

$$((100 \times 50) + (85 \times 49) + (0 \times 48) + (50 \times 47) + (100 \times 46) + \sum_{n=1}^{45} 70n)/1275 = 69.46...$$

This will then be rounded to 69.

Each trader's composite reputational score is stored in the `composite_reputational_scores` dictionary member variable in the `Block_Indication_Book` class. This dictionary uses the trader's unique ID as the key. If a trader's composite reputational score falls below the value defined in the `Block_Indication_Book`'s `RST` member variable, then the trader will no longer be allowed to use the block discovery service.

3.5 Exchange

3.5.1 Adding Orders and Block Indications

The `Exchange` class was created to bring together the order book and block discovery service. The `Exchange` class contains a member variable of the `Orderbook` class and of the `Block_Indication_Book` class. The Exchange class contains a method to add an order to the `Orderbook` called `add_order()` and a method to add a BI to the `Block_Indication_Book` called `add_block_indication()`. These methods make sure that the trader can only have one order or BI on the exchange at any one point in time. If a trader already has an order or BI on the exchange and either of these methods are used to add a new order or BI, then the order or BI already in the exchange will be removed when the new order or BI is added.

3.5.2 Creating Firm Orders out of QBOs

The exchange class contains a method called `add_firm_orders_to_order_book()`. This method takes the QBOs submitted by the traders and converts them into firm orders. These firm orders and are then added to the `Orderbook`. The code for this method is shown in Listing 3.14.

```
def add_firm_orders_to_order_book(self, match_id):

    # get the QBOs for this match
    full_match = self.get_block_indication_match(match_id)
    buy_side_QBO = full_match["buy_side_QBO"]
    sell_side_QBO = full_match["sell_side_QBO"]
```

```
7
8       # create the firm buy order
9       buy_side_order = Order(buy_side_QBO.time,
10                             buy_side_QBO.trader_id,
11                             buy_side_QBO.otype,
12                             buy_side_QBO.quantity,
13                             buy_side_QBO.limit_price,
14                             buy_side_QBO.MES)
15
16      # create the firm sell order
17      sell_side_order = Order(sell_side_QBO.time,
18                              sell_side_QBO.trader_id,
19                              sell_side_QBO.otype,
20                              sell_side_QBO.quantity,
21                              sell_side_QBO.limit_price,
22                              sell_side_QBO.MES)
23
24      # these orders were created using the block discovery service
25      buy_side_order.BDS = True
26      sell_side_order.BDS = True
27
28      # add the firm orders to the exchange
29      self.add_order(buy_side_order, False)
30      self.add_order(sell_side_order, False)
```

Listing 3.14: Code for the `add_firm_orders_to_order_book()` method in the `Exchange` class.

3.5.3 Matching Block Indications and Getting Firm Orders

The `Exchange` class has a method called `match_block_indications_and_get_firm_orders()`. This method first tries to find matching BIs by calling the `Block_Indication_Book` class's `find_all_matching_block_indications()` method. If matches are found, then this method obtains the OSRs that need to be sent to the traders by calling the `Block_Indication_Book` class's `create_order_submission_requests()` method. The OSRs are sent to the traders and the subsequent QBOs are received from the traders by calling each trader's `get_qualifying_block_order()` method (the `Trader` class is described in Section 3.6). The QBOs received from the traders are then added to the `Block_Indication_Book` using the `add_qualifying_block_order()` method. This method adds the QBO to the corresponding item in the `Block_Indication_Book`'s `matches` dictionary. Subsequently, the composite reputational score of each of the traders is updated using the `Block_Indication_Book`'s `update_composite_reputational_scores()` method. The submitted QBOs are then converted into firm orders and added to the order book by use of the `add_firm_orders_to_order_book()` method. The item in the `Block_Indication_Book`'s `matches` dictionary for this match is then deleted. The code for the `match_block_indications_and_get_firm_orders()` method is shown in Listing 3.15.

```
1  # match block indications that are resting in the exchange and then convert the
2  # matched block indications into firm orders
3  def match_block_indications_and_get_firm_orders(self, time, traders, price):
4
5      # match block indications
6      self.block_indication_book.find_all_matching_block_indications(price)
7
8      # go through all matches
9      for match_id in self.block_indication_book.matches.keys():
10
11         # get the block indications that were matched
12         full_match = self.block_indication_book.get_block_indication_match(match_id)
13         buy_BI = full_match["buy_side_BI"]
14         sell_BI = full_match["sell_side_BI"]
15
16         # send an OSR to each traders and get back a QBO
17         OSRs = self.block_indication_book.create_order_submission_requests(match_id)
18         buy_QBO = traders[buy_BI.trader_id].get_qualifying_block_order(time, OSRs["
       buy_side_OSR"])
```

```
19          sell_QBO = traders[sell_side_BI.trader_id].get_qualifying_block_order(time, OSRs[
    "sell_OSR"])
20
21          # add the QBOs to the exchange
22          self.block_indication_book.add_qualifying_block_order(buy_QBO, False)
23          self.block_indication_book.add_qualifying_block_order(sell_QBO, False)
24
25          # update the reputational scores of the traders in the match
26          self.block_indication_book.update_composite_reputational_scores(time, match_id)
27
28          # add the firm orders to the order book.
29          self.add_firm_orders_to_order_book(match_id)
30
31          # delete the block indication match from the matches dictionary
32          self.block_indication_book.delete_match(match_id)
```

Listing 3.15: Code for the `match_block_indications_and_get_firm_orders()` method in the `Exchange` class.

3.6 Traders

I made some modifications to the pre-existing `Trader` class. Since I implemented multiple order sizes, I added a member variable called `quantity_remaining`. This member variable keeps track of how much of the quantity given to the trader in the customer order has been traded so far. I also added a member variable called `reputational_score` to the `Trader` class so that the trader can keep a note of their reputational score.

Each trader that wishes to use the block discovery service has to define a method called `get_qualifying_block_order()`. This function takes an `Order_Submission_Request` object as a parameter and returns a `Qualifying_Block_Order` object. The `get_qualifying_block_order()` method specifies how a trader converts their BIs into QBOs. For example, a trader may send a qualifying block order with exactly the same details as given in the original block indication every time. An example implementation is shown in Listing 3.16.

```
1  # the trader recieves an Order Submission Request (OSR) and needs to respond with
2  # a Qualifying Block Order (QBO)
3  def get_qualifying_block_order(self, time, OSR):
4
5      # update the traders reputationa score
6      self.reputational_score = OSR.reputational_score
7
8      # create a QBO from the received OSR, with exactly the same details
9      QBO = Qualifying_Block_Order(
10         time,
11         OSR.trader_id,
12         OSR.otype,
13         OSR.quantity,
14         OSR.limit_price,
15         OSR.MES,
16         OSR.match_id
17     )
18
19     # return the created QBO
20     return QBO
```

Listing 3.16: Example definition of the `get_qualifying_block_order()` method for a trader.

The only trader that remains in the `dark_pool.py` file is the Giveaway trader. This is because all of the other traders require LOB information in order to function. In the dark pool, no such LOB information is supplied.

3.7 Unit Tests

I performed unit testing on the dark pool using *unittest*, Python's unit testing framework. The source-code for the unit tests that I implemented can be found in the GitHub repository in the `tests/unit/test_dark_pool.py`. I created 75 unit tests in total, testing various functions and methods. Unit testing was performed in order to verify that the behaviour of functions was as expected. The use of unit testing helped with the development process by helping to quickly identify bugs created by making changes to the code. A unit test for the `book_add()` method of the `Orderbok_half` class is shown in Listing 3.17.

```
# Testing that when orders are added to the Orderbook_half class, that they are
# inserted into the correct position based on their priority
    def test_book_add_function_ordering(self):

        # Create an instance of the Orderbook_half class
        booktype = "Buy"
        orderbook_half = dark_pool.Orderbook_half(booktype)

        # Create some orders
        orders = []
        orders.append(dark_pool.Order(25.0, 'B00', 'Buy', 5, 100, 3))
        orders.append(dark_pool.Order(35.0, 'B01', 'Buy', 10, 100, 4))
        orders.append(dark_pool.Order(45.0, 'B02', 'Buy', 10, 110, 4))

        # Add the orders to the Orderbook_half object
        for order in orders:
            orderbook_half.book_add(order)

        # testing that the orders were added successfully and the orders are
        # ordered by priority (Size, Time)
        self.assertEqual(orderbook_half.traders.keys(), ['B01', 'B00', 'B02'])
        self.assertEqual(orderbook_half.traders['B00'], 1)
        self.assertEqual(orderbook_half.traders['B01'], 1)
        self.assertEqual(orderbook_half.traders['B02'], 1)
        self.assertEqual(orderbook_half.orders[0].__str__(), "Order: [ID=-1 T
    =35.00 B01 Buy Q=10 QR=10 P=100 MES=4]")
        self.assertEqual(orderbook_half.orders[1].__str__(), "Order: [ID=-1 T
    =45.00 B02 Buy Q=10 QR=10 P=110 MES=4]")
        self.assertEqual(orderbook_half.orders[2].__str__(), "Order: [ID=-1 T
    =25.00 B00 Buy Q=5 QR=5 P=100 MES=3]")
```

Listing 3.17: Unit test for the `orderbook_half` class's `book_add()` method.

I also performed some integration testing. To perform the integration tests, I used *unittest* module once again. The integration tests that I implemented can be found in the `tests/integration/test_dark_pool.py` file.

Chapter 4

Evaluation

4.1 Functional Testing

4.1.1 The BDS Giveaway Trader

I created a modified version of the Giveaway trading algorithm and named it the *BDS Giveaway* trading algorithm. This trader behaves much like the original Giveaway trader (described in Section 2.5.3) but can now also interact with the block discovery service. This simple trading algorithm was created in order to demonstrate that traders can submit orders and BIs and that trades can successfully execute. The BDS Giveaway trader has a member variable called `BI_threshold`. In the `getorder()` method for this trader, if the remaining quantity left from the trader's customer order is greater than or equal to the value specified in `BI_threshold`, then a block indication will be returned. If the quantity remaining is less than the `BI_threshold`, then a normal order will be returned. The trader also defines the `get_qualifying_block_order()` method. The code for BDS Giveaway trader is shown in Listing 4.1.

```
# BDS Giveaway Trader
class Trader_BDS_Giveaway(Trader):

# define what orders a trader submits
def getorder(self, time):
    # if the trader has no customer order then return None
    if self.customer_order == None:
        order = None

    elif self.quantity_remaining > 0:

        # if the quantity remaining is greater than the BI threshold then issue a
        # block indication
        if self.quantity_remaining >= self.BI_threshold:

            # the minimum execution size
            MES = 100

            # create the block indication
            block_indication = Block_Indication(time,
                                                self.trader_id,
                                                self.customer_order.otype,
                                                self.quantity_remaining,
                                                self.customer_order.price,
                                                MES)

            # update the lastquote member variable
            self.lastquote = block_indication

            # return the block indication
            return block_indication

```

```
33          # otherwise issue a normal order
34          else:
35
36              # the minimum exeuction size
37              MES = None
38
39              # create the order
40              order = Order(  time,
41                              self.trader_id,
42                              self.customer_order.otype,
43                              self.quantity_remaining,
44                              self.customer_order.price,
45                             MES)
46
47              # update the last quote member variable
48              self.lastquote=order
49
50              # return the order
51              return order
52
53  # define how a trader converts their BIs into QBOs
54  def get_qualifying_block_order(self, time, OSR):
55
56      # Update the traders reputational score
57      self.reputational_score = OSR.reputational_score
58
59      # modify these values in order to test the reputational scoring
60      quantity = OSR.quantity
61      limit_price = OSR.limit_price
62      MES = OSR.MES
63
64      # create a QBO from the received OSR
65      QBO = Qualifying_Block_Order(    time,
66                                       OSR.trader_id,
67                                       OSR.otype,
68                                       quantity,
69                                       limit_price,
70                                       MES,
71                                       OSR.match_id)
72      # return the created QBO
73      return QBO
```

Listing 4.1: Code for the `Trader_BDS_Giveaway` class.

4.1.2 Market Sessions

The original `market_session()` function from BSE was modified so that it can work with the dark pool. The `customer_orders()` function was also modified so that it now allows you to specify a range for the quantities that customer orders will have. At the end of a market session, the trades that occurred are written to a CSV file called `transactions.csv` using the `Orderbook` class's `tape_dump()` method. This file displays the time that each trade took place, the buyer and seller in each trade, the quantity of each trade, the price of each trade and finally whether or not each trade was created by use of the block discovery service. This output file demonstrates that trades are actually taking place and that the dark pool is working as expected. An example of the `transactions.csv` output file from a market session is displayed in Figure 4.1.

4.1.3 Reputational Scoring

In order to verify that the reputational scoring mechanism of the block discovery service is operating correctly, I ran a multitude of market simulations. A method called `CRS_history_dump()` was added to the `Block_Indication_Book` class which writes the entire history of each trader's composite reputational

time	buyer	seller	quantity	price	BDS
1.97	B19	S03	246	50	
3.22	B19	S10	45	50	
3.95	B10	S10	418	50	
3.95	B10	S07	81	50	
6.65	B04	S05	380	50	
7.13	B02	S05	2	50	
9.73	B14	S02	523	50	
11.48	B05	S16	333	50	
12.75	B07	S16	262	50	
12.75	B07	S05	170	50	
13.38	B11	S06	822	50	Yes
13.38	B11	S01	138	50	
14.65	B15	S17	353	50	
19.95	B08	S19	267	50	
20.15	B18	S15	855	50	Yes
20.77	B12	S19	167	50	
21.32	B06	S19	212	50	
21.32	B06	S05	186	50	
21.32	B06	S09	63	50	
26.02	B16	S13	89	50	

time	buyer	seller	quantity	price	BDS
30.02	B00	S12	519	50	
30.02	B00	S13	26	50	
30.5	B16	S04	620	50	
30.5	B16	S13	132	50	
31.65	B17	S19	198	50	
32.85	B03	S10	929	50	Yes
32.85	B03	S19	46	50	
37	B05	S08	574	50	
37	B05	S19	34	50	
38.5	B02	S19	474	50	
38.5	B02	S18	284	50	
43.2	B12	S11	246	50	
44.47	B18	S04	414	50	
44.47	B18	S00	182	50	
46.52	B14	S02	954	50	Yes
46.52	B14	S14	9	50	
46.82	B08	S14	262	50	
46.82	B08	S11	268	50	
46.82	B08	S09	219	50	
52.42	B19	S01	52	50	

Figure 4.1: An example `transactions.csv` file produced from a market session.

The market session ran for a period of 600s and the first 40 transactions are shown. In this simulation, there were 20 buyers and 20 sellers. All of the traders were BDS Giveaway traders. The buy side traders were supplied with customer orders with prices from a uniform random distribution within the range of 25 to 45. The sell traders were supplied with customer orders with prices from a uniform random distribution within the range of 55 to 75. The quantity of each customer order was drawn from a uniform random distribution with range 1 to 1000. The midpoint of the lit market was set at £50 for the entire simulation. The MIV for the Block Discovery Service is set to 800. The `BI_threshold` member variable of each trader was set to 800. The MES of each order was set to `None`. The MES of each block indication was set to 100. The QBO details that the traders send are the same as the original BI details. Transactions where the BDS column has a value of `Yes` made use of the block discovery service. As can be seen from the figure, all transactions with a quantity over 800 made use of the block discovery service.

score to a CSV file called `CRS_history.csv`. The BDS Giveaway Trader's `get_qualifying_block_order()` method was modified in different ways and the effect that this had on a trader's composite reputational score was observed.

Figure 4.2 shows how an individual trader's composite reputational score changes over time when the `get_qualifying_block_order()` method returns a QBO with the exact same details as the originally submitted BI. The trader receives an event reputational score of 100 each time they convert a BI into a QBO which results in the trader's composite reputational score rising.

Figure 4.3 shows how an individual trader's composite reputational score changes over time when the `get_qualifying_block_order()` method returns a QBO with a quantity half that of the originally submitted BI. This results in the trader receiving an event reputational score of 50 every time a BI is converted into a QBO. This causes the trader's composite reputational score to fall from its initial value over time. Eventually, the trader's composite reputational score falls below the Reputational Score Threshold. At this point, the trader is no longer allowed to participate in the block discovery service.

Figure 4.4 shows how an individual trader's composite reputational score changes over time when the `get_qualifying_block_order()` method returns a QBO where the MES of the QBO is one greater than the MES specified on the originally submitted BI. This means that the QBO is unmarketable. This results in the trader receiving an event reputational score of zero every time a BI is converted into a QBO. The trader's composite reputational score falls below the Reputational Score Threshold faster than in the previous example.

These three examples all demonstrate the intended behaviour of the reputational scoring mechanism of the block discovery service.

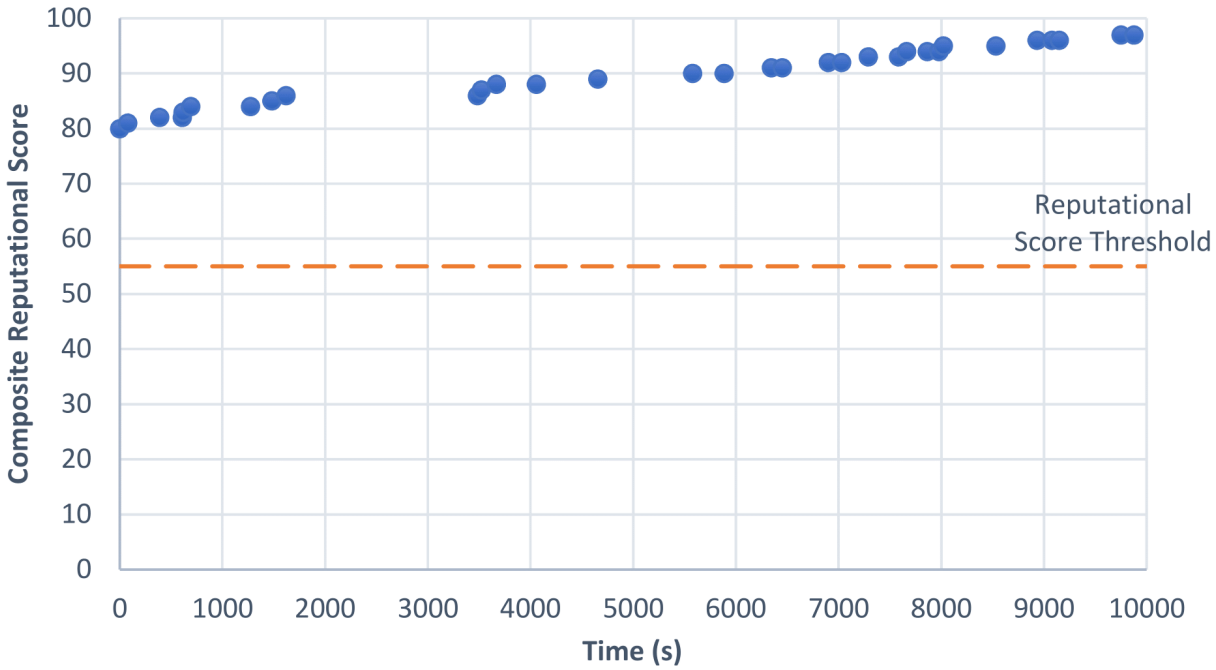

Figure 4.2: The composite reputational score of a trader over time when QBOs are identical to BIs.

This figure shows how the composite reputational score of a single trader is affected over time in a market session. The initial reputational score of the trader is set to 80 and is shown on the graph at time $t = 0$. All other data points represent the trader's composite reputational score after using the block discovery service to make a trade. The Reputational Score Threshold is set to 55 and is shown by a dashed horizontal line. The trader specifies the same details in each QBO as specified in the original BI. As a result, the trader is given an event reputational score of 100 every time they use the block discovery service. This causes the trader's composite reputational score to rise. The trader's composite reputational score is 97 at the end of the market session.

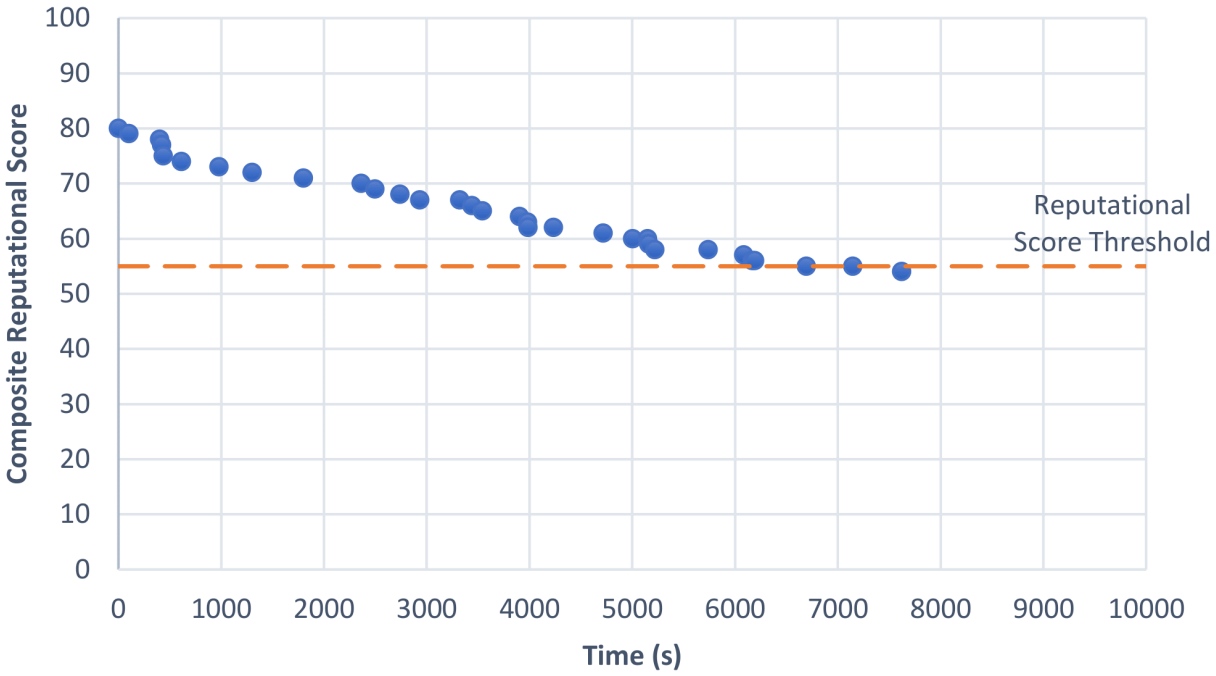

Figure 4.3: The composite reputational score of a trader over time when QBOs have half the quantity of BIs.

This figure shows how the composite reputational score of a single trader is affected over time in a market session. The initial reputational score of the trader is set to 80 and is shown on the graph at time $t = 0$. All other data points represent the trader's composite reputational score after using the block discovery service to make a trade. The Reputational Score Threshold is set to 55 and is shown by a dashed horizontal line. The trader specifies the same details in each QBO as specified in the corresponding BI but the quantity is halfed. As a result, the trader is given an event reputational score of 50 every time. This causes the trader's composite reputational score to decrease after every use of the block discovery service. At time $t = 7621s$ the trader's composite reputational score is 54, which is below the Reputational Score Threshold of 55. At this point, the trader is no longer allowed to use the block discovery service. This results in their being no more data points in the graph.

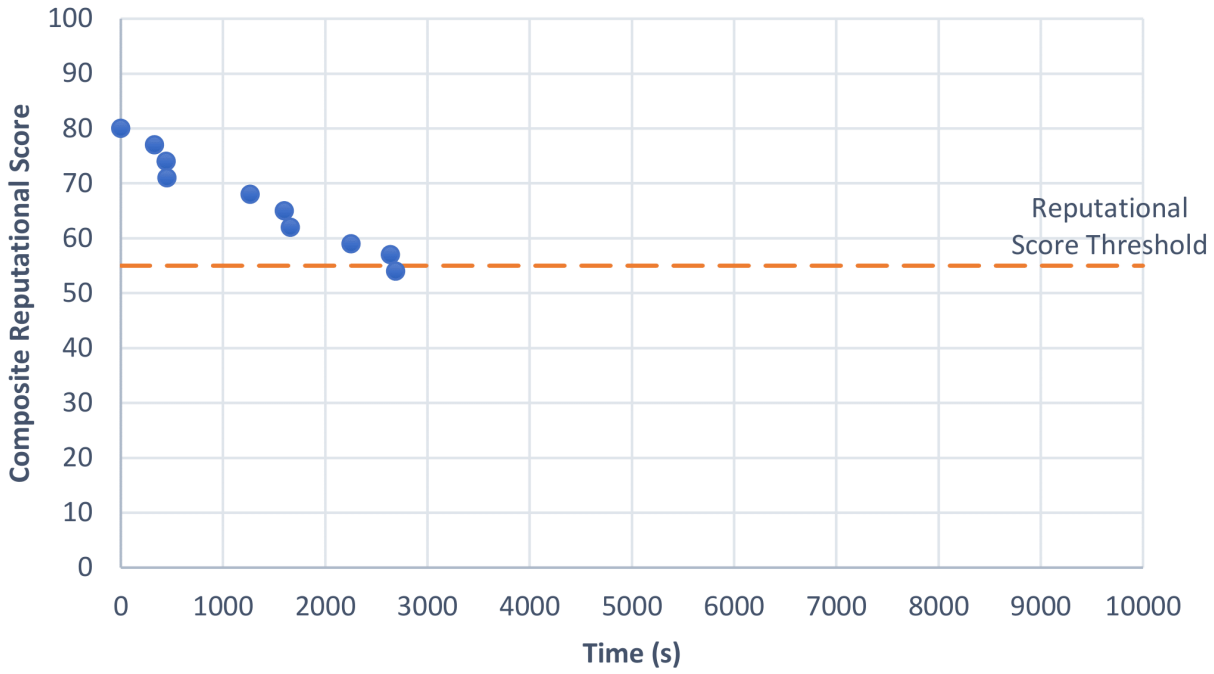

Figure 4.4: The composite reputational score of a trader over time when QBOs are unmarketable.

This figure shows how the composite reputational score of a single trader is affected over time in a market session. The initial reputational score of the trader is set to 80 and is shown on the graph at time $t = 0$. All other data points represent the trader's composite reputational score after using the block discovery service to make a trade. The Reputational Score Threshold is set to 55 and is shown by a dashed horizontal line. The trader specifies the same details in each QBO as specified in the corresponding BI apart from the MES which is set to one greater than in the BI. As a result, the QBO is not marketable. This means that the trader is given an event reputational score of 0 every time they use the block discovery service. This causes the trader's composite reputational score decrease after every use of the block discovery service. At time $t = 2691s$ the trader's composite reputational score is 54, which is below the Reputational Score Threshold of 55. After this point, the trader is no longer allowed to use the block discovery service. This is why after this point there are no more data points.

Chapter 5

Conclusion

5.1 Achievements

I created a working simulator of a dark pool exchange that is based on the Turquoise Plato trading service [5]. Traders can submit buy orders and sell orders into the exchange. Orders are matched for trading in priority order based on their quantity. Trades are executed between matched orders.

The dark pool includes an implementation of the Block Discovery service of Turquoise Plato [4]. This service allows a trader to first determine if their are any counterparties that can match a block order before the trader actually submits a block order. Traders first submit a block indication which indicates the block order that they would like to submit. Once a match between block indications is found, each trader is notified with an order submission request. Traders then submit firm orders via qualifying block orders. Traders are given a reputational score based on their conversion of block indications to qualifying block orders.

5.1.1 Aims and Objectives

- I successfully achieved my aim of creating a minimal simulator of a dark pool that is based on Turquoise Plato. The implementation of the dark pool is described within Chapter 3.

- I didn't manage to meet my aim of integrating the dark pool with BSE. This was due to running out of time. This would have been fairly simple to achieve given an extra couple of days for development.

- I successfully achieved my aim of running market simulations with the dark pool simulator that I implemented. The evidence for this is within Chapter 4.

5.2 Project Status

- The dark pool simulator lacks some of the features that are available on Turquoise Plato which could have been implemented given more time for development. I only implemented a subset of the order types that are available in Turquoise Plato. More specifically, I only implemented persistent order types which rest in the order book until they are fully executed (or are cancelled by the trader). These orders can specify a quantity, a limit price and an MES. I could have also implemented non-persistent order types such as FOK and FAK orders which are described in Section 2.6.1. I did not manage to implement the Block Discovery Notification feature of the Turquoise Block Discovery service. This feature is described in Section 2.6.2. I also didn't manage to implement the Turquoise Plato Uncross events. This feature is used in the trading process of block orders. Implementing all of these features would make the dark pool a more realistic simulator of the Turquoise Plato exchange.

- Traders are currently only able to have one order or one block indication in the exchange at any one point in time. This design decision was taken purely to reduce complexity. The dark pool could be extended to allow traders to have multiple orders and block indications in the exchange at any one point in time.

- During a call of the `market_session()` function, the trader's update their total profit whenever they make a trade. This is done by use of the `Trader` class's `bookkeep()` method. The correctness of this method was not thoroughly evaluated.
- When the QBOs are received from the traders whose block indications have been matched together, the QBOs are converted into orders and are added to the order book. There is no guarantee that these orders will be matched with eachother in the matching logic found in the `find_matching_orders()` function of the `Orderbook` class. It is unclear in the documentation [5, 4] whether this is the correct behaviour or not as it doesn't go into much detail about the Turquoise Plato Uncross matching logic. This did not cause any problems in my select few experiments although it may present issues in future use. A related issue which did cause a few issues for me was that since QBOs do not have to be identical to BIs, a pair of BIs may be matched together but the subsequent firm orders that are created and added to the order book may not possibly to be matched together. For instance, the quantity of one order may be less than the MES of the counterparty's order, therefore the orders cannot be matched together for trading. Again, it is not clear whether or not this is the correct behaviour in the Turquoise Plato Block Discovery service. Further investigation and development is required for these issues.

5.3 Future work

- The dark pool simulator was created so that trading strategies could be investigated in a coupled lit and dark pool. The simulator could be used for this purpose in future work.
- The implementation was not built with speed or efficiency in mind. Research experiments may require running many thousands of market simulations in the dark pool which would require a lot of compute time and resources. If the implementation was redesigned with speed and efficiency more in mind then these experiments would be able to complete in a much shorter time. This is not such a problem, however, with readily available high performance cloud computing resources that could run many thousands of market simulations in a relatively short period of time.
- The implementation assumes that there is zero latency between traders and the exchange. Clearly this is not realistic. Further development could involve implementing simulated latency and performing experiments under this model.
- I ran out of time before I was able to integrate the dark pool with BSE. If I was able to do this, then I could have been able demonstrated that the use of the dark pool would eliminate/reduce market impact effects. This could have been achieved by populating the lit pool with traders that exhibit market impact effects. If only the lit pool is used, then market impact effects will be present when block orders are placed. However, if block orders are instead routed to the dark pool, then the market impact effects should be removed/reduced. Demonstrating this would have been quite novel and it was unfortunate that I ran out time before being able to demonstrate this.

Bibliography

[1] D. Cliff. *Minimal-Intelligence Agents for Bargaining Behaviours in Market-Based Environments*. Hewlett-Packard Labs Technical Report HPL-97-91, 1997. [Online] Available: https://www.hpl.hp.com/techreports/97/HPL-97-91.pdf.

[2] D. Cliff. BSE: The Bristol Stock Exchange, 2012. [Online] Available: https://github.com/davecliff/BristolStockExchange/blob/master/BSEguide1.2e.pdf.

[3] D. Cliff. *An Open-Source Limit-Order-Book Exchange for Teaching and Research*. 2018 IEEE Symposium Series on Computational Intelligence (SSCI 2018), 2018.

[4] London Stock Exchange Group. Turquoise Plato Block Discovery^{TM} (version 2.26.2), 2019. [Online] Available: https://bit.ly/2ZPJAi0.

[5] London Stock Exchange Group. Turquoise Trading Service Description (version 3.34.9j), 2019. [Online] Available: https://bit.ly/2Y1jWFi.

[6] R. Palmer J. Rust, J. Miller. *Behavior of Trading Automata in a Computerized Double Auction Market*. 1992. [Online] Available: https://www.researchgate.net/publication/242530425_Behavior_of_trading_automata_in_a_computerized_double_auction_market.

[7] A. le Calvez and D. Cliff. *Deep Learning can Replicate Adaptive Traders in a Limit-Order-Book Financial Market*. 2018. [Online] Available: https://arxiv.org/ftp/arxiv/papers/1811/1811.02880.pdf.

[8] London Stock Exchange. LSEG - Electronic Order Book Trading, April 2019. [Online] Available: https://www.londonstockexchange.com/statistics/monthly-market-report/lseg-monthly-market-april-2019.pdf.

[9] P. Vytelingum. *The Structure and Behaviour of the Continuous Double Auction*. PhD thesis, School of Electronics and Computer Science, University of Southampton, 2006. [Online] Available https://eprints.soton.ac.uk/263234/1/THESIS.pdf.