Linköping University | Department of Management and Engineering (IEI)
Master's thesis, 30 ECTS | Industrial Engineering and Management, Finance
2023 | LIU-IEI/TEK-A--2023/04695—-SE

Order Matching Optimization

– Developing and Evaluating Algorithms for Efficient Order Matching and Transaction Minimization

Victor Jonsson
Adam Steen

Supervisor : Mathias Henningsson
Examiner : Jörgen Blomvall

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

This report aimed to develop algorithms for solving the optimization problem of matching buy and sell orders in call auctions while minimizing the number of transactions. The developed algorithms were evaluated based on their execution time and solution accuracy. The study found that the problem was more difficult to solve than initially anticipated, and commercial solvers were inadequate for the task. The data's characteristics were critical to the algorithms' performance, and the lack of specifications for instruments and exchange posed a challenge. The algorithms were tested on a broad range of datasets with different characteristics, as well as real trades of stocks from the Stockholm Stock Exchange.

Evaluating the best-performing algorithm became a trade-off between time and accuracy, where the quickest algorithm did not have the highest solution accuracy. Therefore, the importance of these factors should be considered before deciding which algorithm to implement. Eight algorithms were evaluated: four greedy algorithms and four cluster algorithms capable of identifying 2-1 and 3-1 matches. If execution time is the single most crucial factor, the Unsorted Greedy Algorithm should be considered. However, if accuracy is a priority, the Cluster 3-1 & 1-3 Algorithm should be considered, even though it takes longer to find a solution.

Ultimately, the report concluded that while no single algorithm can be definitively labeled as the best, the Cluster 2-1 Algorithm strikes the most effective balance between execution time and solution accuracy, while also remaining relatively stable in performance for all test cases. The recommendation was based on the fact that the Cluster 2-1 Algorithm proved to be the quickest of the developed cluster algorithms, and that cluster algorithms were able to find the best solutions for all tested data sets. This study successfully addressed its purpose by developing eight algorithms that solved the given problem and suggested an appropriate algorithm that strikes a balance between execution time and solution quality.

Acknowledgments

We would like to express our heartfelt gratitude to everyone who has contributed to the successful completion of this project. Our journey would not have been possible without the guidance, support, and encouragement of those around us.

First, we would like to extend our sincere appreciation to our internal supervisor, Mathias Henningsson, for his invaluable counsel and unwavering commitment to our progress throughout the project. His expertise and insights have greatly enriched our work.

We would also like to express our thanks to Severin Nilsson, as well as the entire team at Vermiculus Financial Technologies, for their generous help and support throughout the project. Their collaboration has played a crucial role in our research and allowed us to explore new perspectives.

Abbreviations

CPU	Central Processing Unit 3
CSD	Central Security Deposit 3, 5
IP	Integer Programming 18, 19
LP	Linear Programming 16–20, 36
MCIP	Minimum Common Integer Partitioning 14, 15, 26, 29, 73
MILP	Mixed Integer Linear Programming 16, 17, 19, 20
MIP	Mixed Integer Programming vi, 8, 15, 16, 18–21, 26, 29, 36, 37
MP	Mathematical Programming 19
NP	Non-Deterministic Polynomial 13–15
NP-Complete	Non-Deterministic Polynomial Time Complete 13, 14, 26
OMXS30	OMXS30 is a market index consisting of the 30 most actively traded stocks on the Stockholm Stock Exchange 7, 27, 39, 55, 69, 70
PILP	Pure Integer Linear Program 16

Contents

List of Figures

List of Tables

1 Introduction

The operation of capitalist economies relies heavily on financial markets, which allocate resources and provide liquidity to businesses in need of capital [59]. These markets facilitate the trading of financial holdings between buyers and sellers [36]. Furthermore, financial markets generate securities, products that afford individuals who possess surplus funds the opportunity to achieve a return on investment greater than what they would earn through savings at a financial institution [35].

According to The World Bank, over 60 trillion dollars was traded in 2019 on global stock markets alone [60]. Beyond stock markets there exist other markets that trade on electronic exchanges such as the bond market and the futures market [16]. Given the critical role that financial markets play in the operation of the global economy, it is imperative that these markets operate with strict regulation and efficiency [22].

Optimization is a widely utilized technique within the financial system, with applications in various areas, including the clearing market, to develop efficient systems [40]. However, the majority of optimization research in finance has focused on portfolio optimization [5]. Despite the extensive attention given to portfolio optimization in the literature, there remains a significant gap in exploring other potential applications for optimization within finance. Notably, optimizing trading activities within financial markets presents a particularly intriguing area for investigation.

In the financial markets, exchanges have two major alternatives for trade mechanisms: continuous trading and call auction trading [4]. Continuous trading is the most common mechanism, wherein a trade is executed when the bid and ask price of the instrument is crossed [41]. Call auction trading differs in that buy and sell orders are instead accumulated, a single price is determined which maximizes the amount of traded volume, and a single multilateral batched trade is executed [41]. When exchanges execute trades in a cleared market, they are sent to be cleared and settled by a central clearing house [7]. In cases where the exchange does not clear its own trades, the exchange might have to pay a clearing fee to a clearing house to clear and settle the trades in the multilateral batched trade [47].

There is an incentive then for the exchange to match orders in the batched trade in a manner

that minimizes the total number of transactions to be cleared. This can be formulated as a mixed integer optimization problem wherein buy and sell orders are matched by quantity in order to minimize the number of transactions. Multiple approaches will be developed and evaluated with a focus on finding an algorithm that provides good solutions in a reasonable amount of time for the order-matching problem that will be evaluated in this thesis.

Given the importance of financial markets in the operation of capitalist economies and the increasing use of optimization within the financial system, it is essential to understand the different trade mechanisms and the optimization models that can be implemented within the financial markets architecture. This study aims to contribute to understanding possible solutions by evaluating multiple algorithms for the optimization of order matching in financial markets. By comparing the results of these algorithms, we hope to determine which algorithm provides the best solutions in a reasonable amount of time for this important problem in financial markets. The research is particularly important given the increasing amount of money traded in global financial markets and the need for efficient and regulated operation of these markets for the benefit of all stakeholders.

1.1 Purpose

The purpose of this thesis is to investigate the optimization problem of matching buy and sell orders in a call auction, in order to minimize the number of transactions. Algorithms to address the problem will be developed, evaluated, and tested in order to be able to suggest which approach finds an appropriate balance between execution time and solution quality.

1.2 Delimitations

This thesis will solely focus on the pairing of buy and sell orders in a call auction trading mechanism after the price has been established. Therefore, no consideration will be given to how the price is determined, the price spread, or other pricing factors that could impact the suitability of the algorithms used in this study.

The developed algorithms will be evaluated quantitatively based on execution time and solution accuracy. Although the authors acknowledge that other quantitative factors, such as algorithm robustness and space complexity, may also affect the suitability of the algorithms for implementation, these factors will not be examined in this report. The main focus of the report is illustrated in figure 1.1.

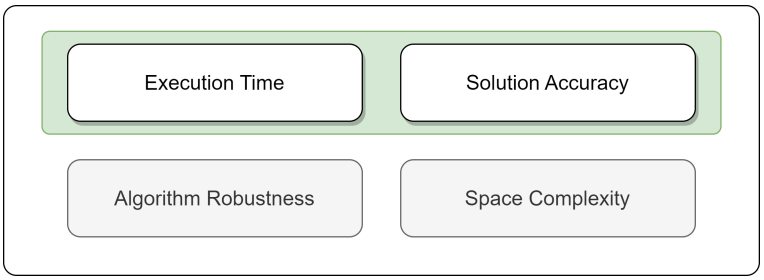

Figure 1.1: Focus Factors

Since pre-trade data is not publicly available, the study will be limited to testing the algorithms on data generated using predefined distributions and publicly available order data. All tested algorithms will be evaluated on the same hardware with identical system settings, namely a Dell Latitude 7300 with an Intel Core i7-8665U CPU.

It should also be noted that in this report, the problem at hand will be referred to as a order matching problem and should not be confused with the established optimization problem category known as matching problems.

1.3 Theoretical Background

Trading in financial markets has historically required face-to-face communications [33]. Although today all large financial markets rely on an electronic foundation to enable their activities [45]. There are several actors in the financial market that are a part of the financial system which enables trading in instruments all around the world. The system is constructed to be stable and secure for all stakeholders [22]. The exchange is a marketplace for buyers and sellers that handles orders and enables trades between actors [33]. Once the match for the trade has been identified by the exchange the clearing house handles the contracts and is a secure third party for the trade to mitigate the risks of default by a counterpart [56]. Once the trade has been finalized the clearing house sends information to a central security deposit (CSD) regarding the transfer of the instrument including, volume, price, and between which parties the deal where made [56]. An illustration of the system can be found in figure 1.2 below.

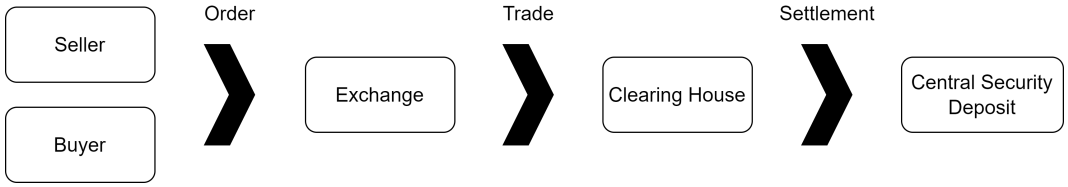

Figure 1.2: Order process

1.3.1 Trading

A trading system structure includes a set of rules governing its trade execution and mechanism as well as data of all trades including ID, volume, time, and price [33]. ID refers to the identification of the actor that sent an instruction for a trade. The volume is equal to the amount of a specific instrument that an actor wants to include in the trade [33]. The time of the trade is when the contract has been registered by the exchange. Price is either what the actor is willing to pay or sell depending on which side of the deal the actor is for the given instrument at a specific time [33].

Trades can be made in three different time periods, pre-market which is before the market actually opens, during market opening hours, and post-market [12]. Different trade mechanism occurs during these various market conditions which are explained below [4] [9] [52].

Continuous Trading

Continuous trading occurs during opening hours of the trade exchange when both buy and sell orders are settled in real-time once there is a match [4]. In a continuous market, a trade

is made whenever the bid and ask prices cross [9]. The orders are prioritized based first on price and secondly, on time as a standard in continuous trading [9].

Call Auction Pricing

In a call auction, the buy and sell orders are accumulated at all price points for each stock for simultaneous execution in a multilateral, batched trade, at a single price, at a predetermined time [52]. The single price mechanism enables a trade matching that only takes volume into consideration in the matching process [15]. A single trading price is determined which maximizes the amount of traded volume and a single multilateral batched trade is executed [52]. In figure 1.1 below an example of how the price is determined in a call auction is presented.

Table 1.1: Call auction pricing example

Price	Buy	Sell	Buy cumulative	Sell cumulative	Tradeable
101.5		300	0	1100	0
101.25		100	0	800	0
101.00	300	500	300	700	300
100.75		100	300	200	200
100.50		100	300	100	100
100.25	100		400	0	0
100.00	100		500	0	0
99.75	100		600	0	0
99.50	500		1100	0	0
99.25	100		1200	0	0

In columns two and three in figure 1.1 the aggregated volume of trades for each price point is presented. In columns four and five respectively the cumulative volume that would be traded at a given price is presented. The logic builds on that all buyers are willing to buy at the price of their order and below and the sell side will sell at all prices at least as big as the asked price of the order. The tradeable volume is the minimum of the cumulative buy and cumulative sell volume at each price point. The price at which the trades will be executed is the maximum value of the tradeable volume in column six in figure 1.1. For the presented example the price of the call auction will be 101.00 and the volume that will be traded is 300.

The auction pricing generally occurs in the pre-trade market and sets the initial trading price for the continuous trading [52]. To determine the closing price in markets around the world different trade mechanisms can be applied to determine the closing price [52]. Many markets use call auction pricing for the last minutes of the day to get an accurate represen-

tation of the closing price [52]. Using call auction pricing for the closing price reduces the volatility and the risk of manipulation [15]. Another way of determining the closing price is to do a volume-weighted price average of the last 15 minutes of continuous trading [52]. Call auction pricing is accordingly used both to determine starting and closing prices for a variety of instruments on markets all around the world.

1.3.2 Clearing

Clearing houses are influential repositories of information for stabilizing and mitigating risks in the financial markets [22]. The role of the clearing house is to store enough capital so they can cover up if one or multiple trading actors would default [22]. To cope with the risk of a defaulting actor, all trades on cleared markets go through a clearing house [22]. When a buy and sell order is matched the clearing house writes contracts with both parties separately. Writing a contract with clearing houses instead of the counterpart in the transaction transfers the risk to the clearing house if either the seller or buyer defaults [7]. The clearing mitigates the risks of defaults by enforcing that all actors on the exchange transfer an amount as a deposit [7]. The amount that each actor has to deposit varies depending on traded volume and changes in different market conditions and is re-balanced daily [7].

One actor can be both an exchange and a clearing house represented in figure 1.2, an actor who operates both the trading and clearing operation is called a silo organization [47]. A silo organization is a vertically integrated organization within an industry [47]. The same regulations apply to both the non-silo organizations and silo organizations [47]. Each transaction to the clearing house is associated with a cost and in a non-silo organization it is the exchange that pays the clearing house for each transaction [7].

1.3.3 Central Security Deposit

Once a trade has been settled it will be registered at the CSD. The main purpose of the CSD is to provide security in the market and prevent market manipulation of any sort [56]. The CSD has a ledger of all transactions for each instrument on a specified market. The most important aspect of a high-performing CSD is security, reliability, and speed [56]. These three factors are not unique for CSD:s but the security aspect is extra important to have a secure financial market where trades can be executed reliably [56].

1.4 Client of Scientific Report

Vermiculus Financial Technologies is a company that provides digital solutions for exchanges, clearing houses, and CSD's around the world. Vermiculus was founded in 2020 and has its office in Stockholm Sweden. Vermiculus works together with its clients to build digital financial infrastructure.

Vermiculus has ordered this report and the purpose of this report was developed together with supervisors from the company. The goal of this report from Vermiculus was to develop an algorithm that they can implement in call auction order matching procedures. The supervisors have been briefed about the progress of the report throughout the process of algorithm development and the writing of the report.

2 Methodology

This chapter presents a scientific approach to achieve the purpose stated in section 1.1. The model, consisting of five parts, namely problem definition, order generation, a survey of potential methods, implementation of order matching algorithms, and model evaluation, is derived from previous literature. These five parts are presented in figure 2.1 and further explained in the following section. An extended model of figure 2.1 will also be presented in figure 2.6, summarizing the topics in the following chapter.

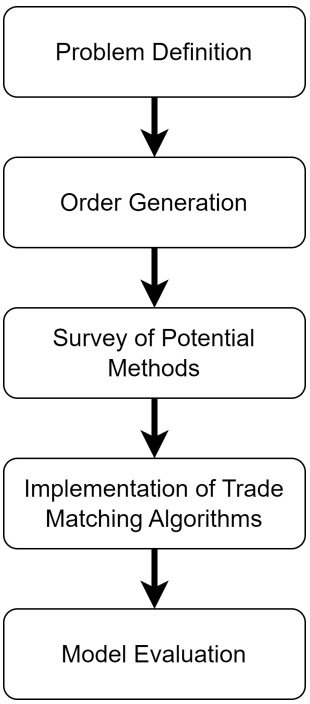

Figure 2.1: Diagram of the method

2.1 Problem Definition

The initial step focuses on understanding the problem and narrowing the scope down to be able to create a clear problem definition. The problem definition involves analyzing the trading market to understand all requirements that need to be fulfilled to enable trades in a call auction. Thereafter analysis of the trading market will be made to understand the operational requirements of a potential algorithm.

The purpose of the report is to solve the optimization problem of minimizing transaction costs. It is therefore important to investigate when transaction costs occur, how big the cost is, and which different types of transaction costs exist, whether they are fixed, variable, or both. However, since it's not specified which type of exchange, this is done in general terms.

When the situation has been fully analyzed a mathematical model of the problem was formulated. The purpose of formulating the problem mathematically is to clearly define the objective function as well as the constraints in a way that can be optimized. All parameters and variables used are defined unambiguously.

2.2 Order Generation

To be able to properly evaluate the algorithms it was fundamental to obtain quality data. The data should be an accurate representation of reality, it was also important to obtain data with different characteristics to make sure that the algorithms were tested against multiple scenarios to assess their capabilities of coping with varying data sets.

The investigation of order sizes from both the buy and sell side was hindered by the lack of publicly available data. To circumvent this issue, an analysis of ten Swedish stocks listed on the OMXS30 was conducted. The objective was to elucidate the distribution of trades and gain insight into the underlying order sizes and their respective distributions. While the size of trades does not perfectly reflect the size of orders, at least one of the two order volumes is typically equivalent to the volume of the trade. This enabled to draw inferences regarding the order sizes. Specifically, the frequency with which different order volumes occurred in the data set. The characteristics of the order sizes were investigated, and assessed whether certain sizes were more common than others

In many financial markets, there are both institutional investors as well as private investors. The order size that an institutional investor trades with is significantly larger than a private investor [65]. Therefore it is important to generate both large and small order sizes to accurately reflect the actual market. Depending on the market and which instrument is traded the value for one instrument can differ pronounced and that directly affects the number of instruments traded in each order [65].

A scenario-generating model was constructed to generate data sets with different distributions, with the possibility to manually determine the total traded volume and the distribution of the order volume for each issuer. The approach also enables to create multiple data sets with different characteristics to evaluate the algorithm on a variety of data sets.

2.3 Survey of Potential Methods

There exist multiple methods to solve the given problem that differs in execution time and accuracy of the solutions. A survey of potential methods was conducted considering exe-

cution time and solution accuracy primarily. S.A. Curtis (2003) suggests greedy algorithms as an approach to solving optimization problems quickly, however with varying solution accuracy [18]. There exist optimization solvers which can produce optimal or near-optimal solutions, but depending on the problem instance may not be sufficiently efficient, Meindl and Templ (2012) compare a number of different open-source and commercial solvers and came to the conclusion that for most problems instances, commercial solvers outperform open-source solvers [46]. According to Achterberg (2007) a primal heuristic may produce sufficiently accurate solutions in a reasonable amount of time [1].

Numerous optimization solvers exist for finding solutions to optimization problems. There are two main categories of solvers, commercial and open-source [6]. Commercial optimization solvers are typically developed and sold by software companies specializing in mathematical optimization. CPLEX, GUROBI, and FICO Xpress are three of the most widely used commercial optimization solvers [46]. Commercial solvers are designed to handle large-scale optimization problems and are superior to open-source alternatives with regard to performance, accuracy, and robustness [46]. To deploy them commercially licenses need to be purchased. These solvers often contain advanced proprietary algorithms, and since they are closed-source there's limited insight into the underlying algorithms [6]. Benchmarks show that commercial solvers are generally better at solving large and complex problems [6].

Open-source optimization solvers are free, publicly available software programs for solving mathematical optimization problems. Some of the most widely used open-source optimization solvers include CLC (for Linear Problems) and CBC (for mixed integer) from the Coin-Or organization, as well as SCIP [46]. Being open-source means the algorithms are often free to use commercially, with the exception of some variants of SCIP [46]. The source code is also available for insight into the underlying algorithms [6]. For some medium-sized problems, open-source solvers may be sufficient [46].

2.4 Implementation of Order Matching Algorithms

Based on the conclusions of the survey of potential methods a number of methods was implemented. First, a simple algorithm was implemented with a low execution time. The simple heuristic was thereafter built on with more complexity with the goal to create an algorithm that reaches a solution as close to optimal as possible.

Secondly, multiple integer optimization solvers were implemented, the goal of implementing a solver was to obtain an optimal solution to use as a benchmark for the other algorithms. Multiple integer solvers were implemented to be able to compare the execution time between available solvers both open-source and commercial.

The implementation of all algorithms was done in the programming language Python and OR Tools, as well as GUROBI's Python interface, were used to implement the MIP-models.

2.5 Model Evaluation

In order to validate the results, a clearly defined evaluation process is introduced. The model evaluation consists of first clearly defining evaluation metrics, in the case of this study execution time and solution accuracy. The algorithms were tested on multiple diverse data sets with different distributions and sizes to evaluate performance in a wide range of use cases. To compare the performance of the algorithms, statistical tests were used to determine whether the differences in the evaluation metrics are statistically significant.

Execution time can be measured in multiple ways. One method is the stopwatch method, where the execution time is measured for the actual execution time of the algorithm. Another method is to calculate the theoretical execution time[58]. For each of these two categories, there exist sub-categories that use a unique approach how to measuring or calculating the execution time [58]. There is no single best technique to use, instead, each technique has its benefits [58].

Stewart (2006) highlights four crucial factors to consider when selecting a method for measuring time: resolution, accuracy, granularity, and difficulty [58]. Stopwatch methods generally have lower accuracy and resolution compared to theoretically calculated methods. However, stopwatch methods are more straightforward to use and provide an actual representation of the execution time [58].

When comparing execution times, it is essential to ensure that the comparison accounts for the same granularity and recognizes the distinction between Wall time and CPU time [31]. Wall time, akin to a wall clock, measures the total actual time elapsed during a given interval [31]. In contrast, CPU time corresponds to the duration the CPU spends processing instructions from the program [31]. This distinction becomes particularly relevant in cases where a program must wait for garbage collection or utilizes multiple threads, potentially leading to inaccuracies in time measurements [31]. There are advantages and disadvantages for each metric, it is most important that the same metric is used to enable comparisons between tests.

In order to measure the solution accuracy of different solution methods, the solutions were compared to the optimal solution derived with a mixed integer optimization solver. All solutions were evaluated to investigate whether it was feasible solutions or not. The validation test was analyzed if all orders are satisfied and checked that no order was sent or received a larger quantity than the order size. The total traded volume for one solution was always summarized to establish that the solution meets the requirements for an order matching algorithm.

2.6 Model Summary

At the beginning of the chapter, a holistic overview of the model was presented that illustrated the five parts that set the foundation of the scientific method and the approach for the thesis. The key topics and approaches presented in the sections above are summarised into bullets and added to the previously presented model to clarify each step even further. The extended model is presented in 4.1.

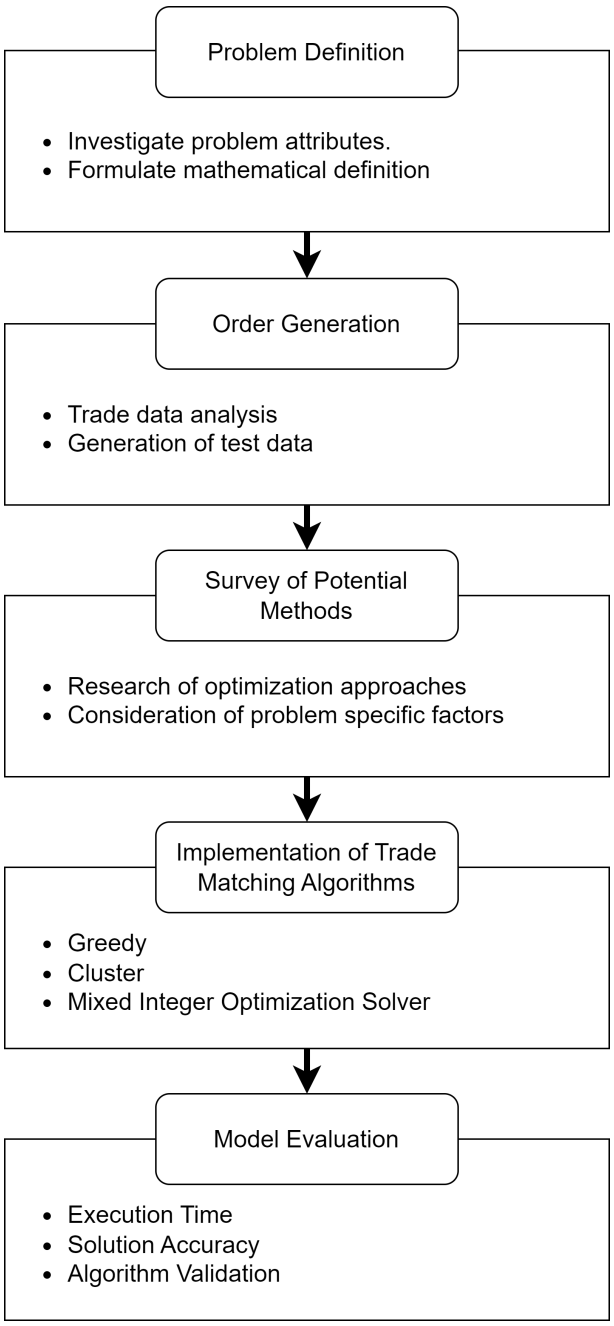

Figure 2.2: Extended diagram of the method

3 Theory

The presented theory in the following chapter serves as the mathematical foundation for calculations and optimization purposes. Aspects of scenario generation techniques are presented followed by optimization problem characteristics and different methods available to obtain a solution.

3.1 Scenario Generation

Scenario generation is widely used to solve problems that contain elements of uncertainty [37]. To implement a successful scenario generation method there are at least two requirements for the generation technique. The first is stability, meaning that the different trees generated should reach the same optimal value when the optimization problem is being solved [37]. The second requirement is that the scenario should not introduce any biases, therefore it is important to find a probability distribution that represents the true probability distribution correctly [37].

When evaluating and developing algorithms that build on big data sets the most common problem is the lack of available data [50], therefore a common approach is to generate data to evaluate the algorithms. To obtain quality data when generating data it is important to accurately reflect the empirical data [44]. It is important to capture the distribution of the data as well as the quantity, deviation, and mean to be able to recreate the properties of the empirical data [55].

3.1.1 Probability Distribution

A probability distribution is a formula that describes the likelihood of different outcomes for a random variable [49]. The random variable is a variable that can take on different outcomes depending on the outcome of the system [49]. The probability distribution assigns a probability value ta all possible outcomes such that the sum of the probability sums to 1 [49].

There exist multiple different probability distributions with different characteristics [49]. One widely applicable with a lot of real-world applications is the normal distribution [20]. The normal distribution has two input parameters, mean μ and the standard deviation σ

[20]. Another distribution is the uniform distribution where all outcomes are equally likely to occur [23]. It is also common in the financial market to construct a distribution with other characteristics such as fat-tail, which is when extreme outcomes are more common compared to the normal distribution[44].

3.2 Statistical Test

Statistical tests are a set of mathematical methods used to determine whether the results of an experiment or study are statistically significant. These tests help researchers assess the probability that their findings occurred by chance, rather than as a result of the treatment or intervention being studied.

3.2.1 Student t-test

The student t-test is a widely used statistical technique used to test if the mean difference between two groups is statistically significant [48]. It is based on the assumption that the data is normally distributed and that the variances of the two groups are equal [39]. The t-test calculates a t-value, which is the difference between the means of the two groups divided by the standard error of the difference [48]. The t-value is then compared to a critical value from a t-distribution with degrees of freedom equal to the sample size minus two [48].

Suppose we have two independent samples, $X_1, X_2, \ldots, X_n$ and $Y_1, Y_2, \ldots, Y_m$, with sample means $\bar{X}$ and $\bar{Y}$ and sample standard deviations s_X and s_Y, respectively. The null hypothesis is that the means of the two populations from which the samples were drawn are equal, i.e., $H_0 : \mu_X = \mu_Y$. The alternative hypothesis is that the means are not equal, i.e., $H_A : \mu_X \neq \mu_Y$[39]. Under the assumption of normality and equal variances of the populations, the test statistic

$$t = \frac{\bar{X} - \bar{Y}}{\sqrt{\frac{s_X^2}{n} + \frac{s_Y^2}{m}}} \tag{3.1}$$

follows a t-distribution with degrees of freedom given by

$$df = n + m - 2 \tag{3.2}$$

where n and m are the sample sizes of the two independent samples The p-value can then be calculated based on the distribution of the t-test statistic and the chosen level of significance α. If the p-value is less than α, we reject the null hypothesis and conclude that the means of the populations are different [48].

The t-test is a powerful tool for comparing means, but it has some limitations. It assumes that the data is normally distributed and that the variances of the two groups are equal, which may not always be the case [48]. Additionally, the t-test is sensitive to outliers, so it is important to check for outliers and consider their impact on the results. Overall, the t-test is a valuable tool for comparing means and can provide useful insights in a variety of research contexts [48].

3.3 Time Complexity

Time complexity is a measure of the amount of time an algorithm takes to run as a function of the size of the input. It's used to analyze the efficiency of an algorithm and compare the performance of different algorithms. Time complexity is usually expressed using big O notation, which gives an upper bound on the number of operations the algorithm performs.[38]

For instance, take an unsorted array of n numbers. A linear search algorithm searching for one number in the array would in the worst case have to iterate through every element in the array, in the best case the algorithm would find the number on the first iteration. Therefore, the best time complexity for this algorithm would be expressed as $O(1)$ using the big O notation, while the worst time complexity would be expressed as $O(n)$. Now consider a two-dimensional array of numbers with n columns and n rows, a linear search algorithm in the worst case has to search all n^2 elements, such an algorithm would therefore have the notation $O(n^2)$. In figure 3.1 time complexity is visualized in a graph showing how the number of operations of different time complexities is affected correspond to the number of elements.

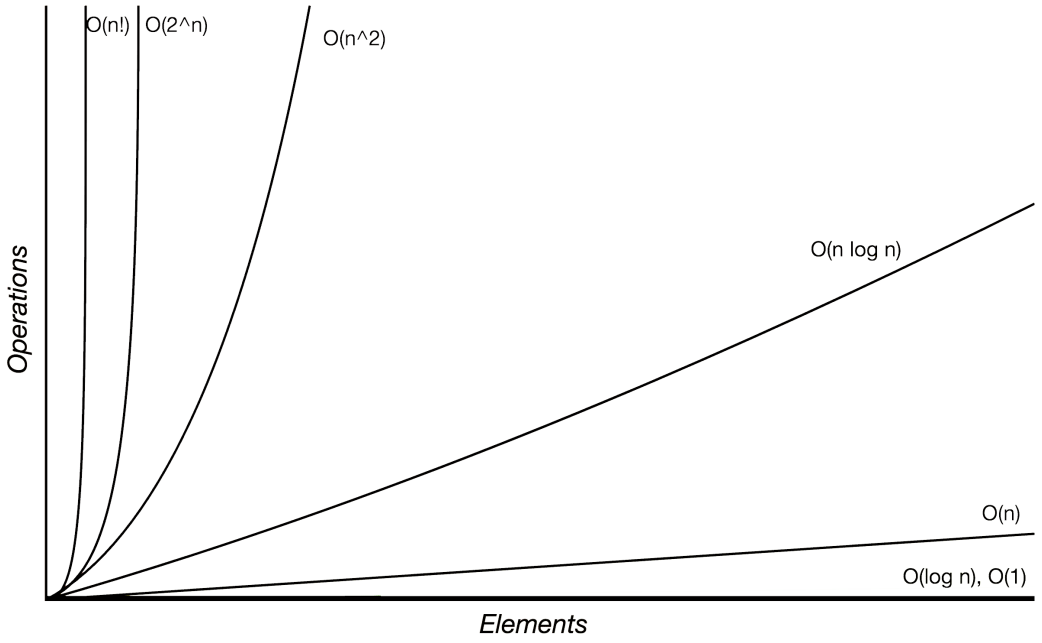

Figure 3.1: Time complexity visualized

Whether an algorithm can be considered efficient or inefficient depends on the context in which the algorithm is applied[25]. However, there is a general distinction between algorithms with a polynomial time complexity and algorithms with an exponential time complexity[25]. When considering large problems, algorithms with exponential time complexity can not be executed in a feasible amount of time. In computer science, polynomial time refers to the running time of an algorithm that can solve a problem within a time frame that is proportional to a polynomial function of the input size [34]. The distinction between polynomial and exponential time is an important one in computer science, as there are classes of problems that can not be solved in polynomial time[25].

3.3.1 NP-Complete Problems

A NP-Complete (non-deterministic polynomial time complete) problem is a mathematical problem for which there is no known algorithm that can solve the problem in polynomial time independent of the size of the problem [25]. NP is a set of problems that can not be solved by a normal computer in polynomial time but could theoretically be solved by a non-deterministic computer in polynomial time [25]. A problem is an NP-Complete problem if it is in NP, and every other problem in NP is reducible to it in polynomial time [25]. A

reduction in computational complexity theory refers to mapping an instance of one problem to another problem, if possible then an algorithm that solves the first problem can also be converted to solve the second problem [25]. If problem A is reducible to problem B, it can not be harder to solve problem A than to solve problem B [25]. Therefore, if there exists an algorithm that can solve one NP-Complete problem in polynomial time, any NP-complete problem can be solved in the same time[25]. No such algorithm exists currently [34]. This has consequences on the appropriate approach to a problem. If a problem is proven to be NP-complete then it can be concluded that no algorithm can efficiently find the optimal solution in polynomial time, but an algorithm could find an approximate solution in a feasible amount of time.

The theory of NP-completeness applies only to the class of problems called *decision problems* [25]. Decision problems have only two possible solutions, "yes" or "no"[25]. The theory of NP-completeness is however also relevant to optimization problems, as a decision problem can be derived from an optimization problem [25]. In "Computers and Intractability: A Guide to the Theory of NP-completeness", authors Michael R. Garey and David S. Johnson provide a relevant example of how a generic instance of the Traveling Salesman Problem can be described as a decision problem [25]:

> INSTANCE: A finite set of $C = c_1, c_2, ..., c_m$ of "cities," a "distance" $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, and a bound $B \in Z^+$ (where Z^+ denotes the positive integers).
>
> QUESTION: Is there a "tour" of all the cities in C having a total length no more than B, that is, an ordering $< c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(m)} >$ of C such that
>
> $$\left[\sum_{i=1}^{m-1} d(c_{\pi(j)}, c_{\pi(i+1)})\right] + d(c_{\pi(m)}, c_{\pi(1)}) \leqslant B \ ?$$

So long as the cost function of the optimization problem is easy to evaluate, the corresponding decision problem can not be harder than the optimization problem [25]. Therefore, if the corresponding decision problem can be proven to be NP-Complete, then it can be concluded that the optimization is at least as hard [25]. The common approach to proving that a problem Π is NP-Complete is reducing a known NP-Complete problem to it. To be considered NP-Complete, a problem must satisfy two requirements [25]:

1. $\Pi \in$ NP, and
2. a known NP-complete problem Π' transforms to Π.

3.4 Minimum Common Integer Programming

In 2006 the MCIP problem was first introduced by Chen et al. [13]. Chen et al. defines the MCIP problem as a partition of a positive integer n is a multiset of positive integers that adds up to exactly n. An integer partition of a multiset S of integers is defined as the multiset union of partitions of integers in S. In this study, a sequence of multisets $S_1, \ldots, S_k$ of integers, where $k \geqslant 2$ was considered. A multiset is said to be a common integer partition if it is an integer partition of every multiset $S_i, 1 \leqslant i \leqslant k$. The minimum common integer programmingn (MCIP) problem is defined as finding a common integer partition of $S_1, \ldots, S_k$ with the minimum cardinality. When k is equal to 2 there exist two lists of integer that needs to be matched with the minimum number of transactions.

The MCIP-problem is a NP-hard problem [14], and is therefore not solvable in polynomial time. To solve the problem Chen et al. implemented a greedy algorithm that sorted all elements in the array by order before doing the match. The greedy approach was chosen to implement an algorithm that can solve the complex problem in a short time compared to other MIP-solvers [14]. The implemented greedy algorithm was proven to be an approximation that solved the problem within 100 % of the optimal solution [14].

The study by Chen et al. provides a proof of the upper bound for the generated 2-MCIP algorithm which was produced in the study [13]. By reducing it to a maximum set packing problem, where the collection of sets is formed by all pairs of basic related submultisets of sizes three and four between two input multisets S and B. To solve this problem, a heuristic algorithm was designed, which first finds a maximal set packing and then recursively replaces a multiset of size four in the packing by a multiset of size three or adds a multiset into the packing until no more replacements or additions can be made. The algorithm can be implemented in $O(|U| \cdot |C|^2)$ time, where U is the universe of elements in the set packing and C is the collection of multisets formed by pairs of submultisets from S and B.

Two variables, q_3 and q_4, are introduced to denote the numbers of pairs of basic related multisets of sizes three and four in the packing found by the heuristic algorithm and let q_3^* and q_4^* the numbers of pairs of basic related multisets of sizes three and four in an optimal weighted set packing, respectively. It is then shown that $2q_3 + q_4 \leqslant 2q_3^* + q_4^*$ and $2q_3^* + q_4^* \leqslant 4(q_3 + q_4)$ based on properties of the multisets and the heuristic algorithm. By combining the two inequalities, the proof establishes that the heuristic algorithm provides a $5/4$-approximation to the optimal solution since q_3 and q_4 can be computed in polynomial time. Specifically, the approximation ratio is $5/4$ because $4(q_3 + q_4)$ is an upper bound on the optimal solution, and the heuristic algorithm finds a solution with weight at least $(2q_3 + q_4)/2$, which is at least $5/4$ times the optimal weight. Therefore, the proof concludes that the $5/4$-approximation algorithm for the 2-MCIP problem is validated. Meaning that the developed algorithm produces approximations that are at most 20 % from the optimum. The $5/4$-approximation algorithm runs in $O((m+n)^9)$

Further improvements in the algorithms have been made that guarantee a solution closer to the optimal amount of trades [63]. Different improvements have been tested but finding common occurring elements from the list and removing these elements before proceeding with other matching techniques has proven to improve the efficiency of the algorithm [42]. Although improvements have been made the best solution published is an $6/5$-approximation algorithm that runs in a time complexity of $O((m+n)^{186})$ where m is the number of elements in the first list and n is the number of elements in the second list [42].

3.5 Optimization Problems

A wide variety of problems have been shown to be polynomially solvable via dynamic programming recursive formulations [10]. Linear programming is also shown to be an efficient method to reach an optimal solution for problems with smaller amounts of data [10]. Linear programming also has the advantage of a high execution speed compared to other optimization methods [10].

3.5.1 Linear Programming

Linear programming (LP), also referred to as linear optimization, is a method for solving a class of optimization problems where the target function and constraints are linear [57]. The decision variables make up an objective function to be minimized or maximized based on constraints [57]. In linear programming, all variables are allowed to be continuous, which means that problems can often be solved accurately and quickly [57]. For this model x_j is the decision variables. With linear constraints and $x_j \in \mathbb{R}$ the problem can be defined as

$$\begin{aligned} min \quad T = \quad & c^t x \\ s.t. \quad & Ax = b \\ & Dx \geqslant d \\ & x_j \in \mathbb{R}, j \in J \end{aligned} \tag{3.3}$$

for some vectors $c \in \mathbb{R}$, $b \in \mathbb{R}$, $d \in \mathbb{R}$ matrices $A \in \mathbb{R}^{m \times n}$, $D \in \mathbb{R}^{p \times n}$ and subset $J \in [1, \cdots, n]$ of the variables [17]. All variables are continuous and can be defined in an arbitrary set of real numbers.

3.5.2 Mixed Integer Programming

When constructing optimization problems a common constraint is that some variables need to be integers to reflect the problem accurately [17]. Common interpretations are for decision variables if a variable should be used or not as well as constraints that state that a variable can not be divided into fractions [17]. If a problem has variables with integer constraints as well as variables that are allowed to be fractions of an integer then the problem is classified as a mixed integer programming (MIP) [17]. With linear constraints, the problem can be defined as

$$\begin{aligned} min \quad T = \quad & c^t x \\ s.t. \quad & Ax = b \\ & Dx \geqslant d \\ & x_j \in \mathbb{Z}, j \in J \end{aligned} \tag{3.4}$$

for some vectors $c \in \mathbb{R}$, $b \in \mathbb{R}$, $d \in \mathbb{R}$ matrices $A \in \mathbb{R}^{m \times n}$, $D \in \mathbb{R}^{p \times n}$ and subset $J \in [1, \cdots, n]$ of the variables [17]. If all variables are restricted to integers then the problem is classified as a pure integer linear program (PILP). Additional constraints such as binary constraints where a variable only can take the value $0, 1$ can replace the integer constraint or be combined with integer constraint variables.

Linear and integer optimization are among the most successful and widely used tools for making decisions in the quantitative analysis of practical problems [57]. There exist numerous open-source solvers that can solve MIP and MILP problems in a sufficient way [54]. The available solvers each have their pros and cons and there is important to implement the right solver for the given problem [54].

3.6 Upper and Lower Bounds

Upper and lower bounds in optimization refer to boundaries on the feasible solutions of an optimization problem [57]. These bounds set limits on the range of values that the objective function in the problem can take [57]. Upper bounds define the maximum value a variable can take, while lower bounds define the minimum value a variable can take [57]. Upper

and lower bounds are used to define the feasible region for an optimization problem and to limit the search space of the optimization algorithm [21]. By setting bounds, the optimization algorithm can be guided towards solutions that satisfy the constraints of the problem, making it more efficient and effective [21].

3.6.1 Deriving Upper and Lower Bounds

The methods for deriving upper and lower bounds vary depending on the type of optimization problem being solved [32]. In linear programming, bounds can be derived from the constraints of the problem, while in nonlinear programming, bounds may be derived through derivative information, such as the gradient of the objective function [32].

There are several methods for finding upper and lower bounds, including brute-force search, linear programming, convex optimization, the Lagrange multiplier method, and duality theory [67]. Greedy heuristic are one common method to derive the upper bound of a minimization problem and LP-relaxation is equally as common to derive the lower bound of a MILP- minimization problem. The choice of method depends on the specifics of the optimization problem being solved and the computational efficiency of the method [67].

3.7 Perfect Match Trading

In call auction pricing there can occur situations when a buy order has the exact same quantity as a sell order, this is referred to as a perfect match. According to Nilsson (2023) [51] a perfect match is always optimal when minimizing the number of transactions between the buy and the sell side. The proof of the statement from Nilsson (2023) follows below.

Let $(\mathcal{B}, \mathcal{S})$ be an order log. Given a subset $B \subset \mathcal{B}$ with a matching $x[i,j]$, if $x[i,j]$ does not perfectly match, then there exists $b \in B$ and $s \in \mathcal{S}$ (which appears in $x[i,j]$) such that the quantity $q_s = q_b$. Let $S = \{s_1, \ldots, s_k\}$ and $B = \{b_1, \ldots, b_m\}$ Using the notation it follows that $S = b_1 + \ldots + b_m$, and per definition $q_{s_1} + \ldots + q_{s_k} = q_{x_1} + \ldots + q_{x_m}$, where m is the number of buy orders and k is the number of sell orders. In particular, we may further divide the s_j's so that they align with the b_i's by making strategic splits at carefully chosen $s_{j_1}, \ldots, s_{j_m}$. Specifically, there exists an index j_1 such that $q_{b_1} \leqslant q_{s_1} + \ldots + q_{s_{j_1}}$, thus subdivide s_{j_1} in two parts $s_{j_1}^{(1)}$ and $s_{j_1}^{(2)}$ so that $q_{b_1} = q_{s_1} + \ldots + q_{s_{j_1}}^{(1)}$ and $q_{s_{j_1}} = q_{s_{j_1}}^{(1)} + q_{s_{j_1}}^{(2)}$. Inductively, continue this process for $b_2, \ldots, b_m$, and for each subdivision S_i replace b_i with the elements $s_{j_{i-1}}^{(2)}, \ldots, s_{j_i}^{(1)}$. Calls these new subdivisions S_i'. Then, notice

$$|S_1'| = |S_1| + j_1 - 1 \qquad \text{and} \qquad |S_i'| = |S_i| + (j_i - j_{i-1}) \quad \forall\, 2 \leqslant i \leqslant m$$

and moreover replace S via $S' = \{s = b_1 + \cdots + b_m\}$, so that $|S'| = |S| - (k-1)$. Putting it all together we have that

$$\begin{aligned} |S'| + |S_1'| + \ldots + |S_m'| &= |S'| + (|S_1| + j_1 - 1) + \ldots + (|S_m| + j_m - j_{m-1}) \\ &= |S'| + |S_1| + \ldots + |S_m| + (j_1 - 1) + \sum_{l=1}^{m} j_l - j_{l-1} \\ &= |S'| + |S_1| + \ldots + |S_m| + j_m - 1 \\ &= |S| - (k-1) + |S_1| + \ldots + |S_m| + j_m - 1 \\ &= |S| + |S_1| + \ldots + |S_m| \end{aligned} \tag{3.5}$$

since $j_m = k$. Through the newly created subdivisions S', $S_1', \ldots, S_m'$ we have perfectly matched b with s and not increased the transaction cost. This clearly illustrates that perfectly matching preserves or lowers transaction costs and is thus optimal.

3.8 Approaches for Solving MIP-models

Introducing integrality constraints to LP models can increase the complexity of the problems significantly[57]. In a linear program, the optimal solutions exist at one of the vertices of the feasible region[57]. When introducing integrality constraints this may no longer hold true, as the feasible integer solutions are not necessarily on one of the vertices of the feasible region of the original LP-problem[57]. The methods for solving IP-problems can be divided into three categories: Enumeration methods, relaxation methods, and heuristic methods.

There exist several approaches for solving MIP-models, the most commonly used are the same approaches as for solving IP-problems. One commonly used method is the Branch-and-Bound first proposed by Land and Doig (1960) which is an enumeration method [19]. The cutting plane algorithm developed by Ralph Gomory 1958 [30] is another method used. Both methods have been further developed and applied to solvers over the past 60 years and are effective methods for solving mixed integer optimization problems [54]. The third most used approach is dynamic programming which is specially adapted for complex problems that can be divided into sub-problems [54]. All three methods are still in use today and the branch and bound as well as cutting plane approach is described in the section below. The dynamic programming approach will not be relevant to the presented problem structure and therefore not be further discussed in the report.

3.8.1 Enumeration Techniques

Enumeration techniques in optimization are methods that systematically examine all possible solutions to a problem in order to find the optimal solution [26]. The basic idea is to list all possible combinations of variables and evaluate the objective function for each combination to determine the best solution [26]. There exist several types of enumeration techniques but the most used method is branch-and-bound [57].

The algorithm used to solve a model with branch-and-bound involves a process of dividing the feasible region into smaller and smaller sub-regions (branching), calculating estimates of the objective value for each sub-model (bounding), and using these estimates to eliminate sub-models that do not contain a better solution than what has already been found [57]. This is achieved by replacing the current sub-model with a simpler model (relaxation) whose solution provides an estimate of the original sub-model [57]. It is also known as implicit enumeration, divide-and-conquer, backtracking, or decomposition [1]. The process terminates when every sub-model has either been shown to contain no better solution or has produced an infeasible solution [57]. The optimal solution is then the best solution found throughout the procedure [57].

To gain a detailed description of branch and bound, a pseudo-code example in the appendix section A.2 algorithm 8 can be found. The example is written in a general context to illustrate the logic of a branch and bound algorithm for a general optimization problem.

3.8.2 Relaxation Techniques

Another way of solving IP-problems is relaxing the optimization problem to an LP problem by removing the integrality constraint [57]. In general, the optimal solution of the LP relaxation is not the optimal solution, or even a feasible solution, for the IP problem [57]. There are two essential techniques for finding a feasible integer solution from an LP-relaxation: rounding and the cutting-plane technique [57].

The process of rounding in mixed-integer optimization refers to the conversion of con-

tinuous variables obtained as the solution to an LP problem into integer values [57]. This approach can result in feasible solutions for IP problems, but may also result in sub-optimal solutions that are far from the true optimal integer solution [57]. Rounding algorithms can produce improved solutions by employing smarter methods for converting the continuous variables of the LP solution into integers [62]. This can result in solutions that are closer to the optimal integer solution, thus increasing the quality of the overall solution to the IP problem [62].

The cutting-plane technique involves iteratively adding linear constraints to the problem, which separates the current solution from infeasible solutions. This results in a sequence of relaxed linear programs, which are solved to improve the objective value until a feasible integer solution is found [29]. An example of the cutting plane algorithm can be found in appendix section A.2 in algorithm 9.

Domain Propagation is another relaxation technique used although not as frequently [1]. The term "Domain Propagation" refers to the process of refining the domains of variables through the examination of constraints and the current domains of other variables within a localized sub-problem in the search tree [1]. Within the Mathematical Programming (MP) community, this process is commonly referred to as "Node Preprocessing" [1]. The primary constraint for operations performed on local nodes is to maintain the integrity of the constraints [1]. Specifically, deletion of variables is prohibited, as this would result in a significant increase in bookkeeping and LP management overhead [1]. Instead, the focus is on tightening the domains of the variables, which can be achieved without substantial impact on the management overhead [1].

Lagrange relaxation

Lagrange relaxation is an optimization technique that is commonly employed to solve complex optimization problems that are constrained [11]. By incorporating Lagrange multipliers into the original objective function, the constrained optimization problem is transformed into an unconstrained, or just a simpler optimization problem [11]. Lagrange relaxation is widely utilized in solving diverse optimization problems such as integer programming, combinatorial optimization, and nonlinear programming [11].

The Lagrange relaxation technique involves relaxing the initial constraints of the problem and appending them as penalty terms to the objective function [11]. The penalty term is subsequently minimized concerning the Lagrange multipliers to acquire a lower bound on the objective function [11]. This lower bound is beneficial since it is a reasonable approximation of the optimal solution, and the unconstrained optimization problem is more straightforward to solve [11].

3.8.3 MIP-Solver Methods

A MIP-solver uses a variety of methods to approach different optimization problems. The already introduced methods, cutting plane and branch and bound lay the foundation of the solvers. Although multiple heuristics are also implemented to improve the solving process. Three heuristics that are implemented in MIP-solvers used in the report are presented below.

Rounding Heuristics

Rounding Heuristics begin with a fractional vector $\hat{x} \sim \in R^n$ that satisfies the linear constraints $Ax \leqslant b$ and $x_j \in \mathbb{R}$ of the mixed integer linear programming (MILP) problem, but violates some of the integrality restrictions [2]. Typically, the optimal solution of the linear programming (LP) relaxation of the current sub-problem is used as the starting point [2]. The

goal of a classical rounding heuristic is to round the fractional values $\hat{x}_j$, $\{ j \in I \mid \hat{x}_j \notin \mathbb{Z} \} \subseteq I$, of the integer variables up or down such that the final integral vector $x' \in I$, $\forall j$ still satisfies all linear constraints [2]. More advanced heuristics may also modify the values of continuous variables or integer variables with integral values to restore linear feasibility that was lost due to the rounding of fractional variables [2].

Diving Heuristics

In addition to rounding heuristics, diving heuristics for mixed integer programming are another type of heuristic approach [1]. Diving heuristics select a variable and explore different branches of the search tree, diving deeper into the sub-problems that appear promising [1]. The objective is to guide the search towards an optimal solution in a more focused and efficient manner compared to a general branch-and-bound approach [1].

Improvement Heuristics

The objective of improvement heuristics is to find feasible solutions with improved objective values, starting from one or more initial feasible solutions [1]. An improvement heuristic can be very similar to an enumeration technique that can search for a better solution with methods like branch-and-bound [1]. But with these types of heuristics, it always starts with a feasible solution and searches in multiple directions with the aim to find a better solution even though it does not have to be optimal like branch-and bound [1].

3.8.4 Presolving

Presolving is a technique used to simplify a given MILP problem instance by transforming it into an equivalent, easier-to-solve instance [25]. Many MILP instances encountered in practice contain a large amount of unnecessary data that can slow down the solving process [25]. To address this issue, all leading MIP solvers employ some form of presolving. The core concepts of presolving in MILPs have been described in detail by Linderoth and Savelsbergh (1999) [43].

The purpose of presolving is three-fold: (1) reducing the size of the model by eliminating redundant constraints or fixed variables, (2) strengthening the linear programming, the LP-relaxation of the model by utilizing integrality information to improve bounds on variables or constraint coefficients, and (3) extracting information such as implications or cliques from the model which can be used for branching or cutting plane separation [43].

3.8.5 Heuristic Methods

The Branch-and-Bound algorithm and cutting plane techniques, which are used to solve mixed integer programming problems, are considered a complete procedure [3]. This means that, excluding numerical inaccuracies, they will always find the optimal solution within a finite amount of time for any given problem instance [3]. However, it is a computationally intensive method and its worst-case run time grows exponentially with the size of the problem instance [3]. On the other hand, Primal Heuristics are incomplete methods, striving to find feasible solutions of acceptable quality within a reasonable time frame, but there is no guarantee of success in finding a solution, let alone an optimal one [1]. Nevertheless, heuristics can have significant advantages and be able to find a solution in a quick and effective way [1].

Greedy Heuristics

A greedy algorithm makes a sequence of choices, following the problem-solving heuristic of making the best available choice in each step [18]. Greedy algorithms are straightforward and efficient, but depending on the problem they may not produce optimal solutions [18]. They are characterized as short-sighted as they don't exhaustively consider all data [18].

Figure 3.2 visualizes a greedy algorithm for finding the largest sum in a binary tree. In this example, the greedy algorithm looks at the next available subsequent nodes and chooses the greatest one. The short-sightedness of greedy algorithms is exemplified here, by making choices based only on what the next best step is, the algorithm disregards the most valuable overall route.

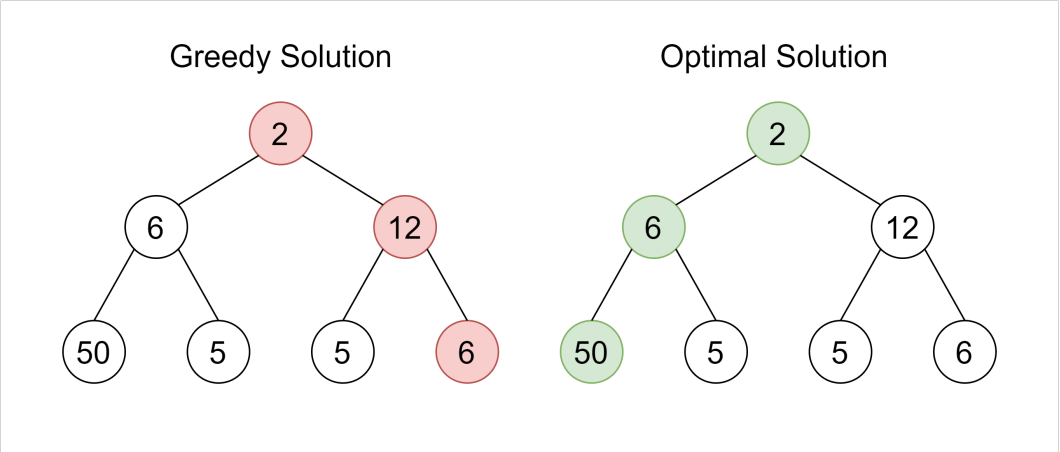

Figure 3.2: Example of a greedy algorithm

3.8.6 Metaheuristics

The term metaheuristics were first introduced in the 1980s by Glover (1986) [27]. Metaheuristics form a group of algorithms that act like guiding mechanisms for the underlying heuristics available to solve the problem [28]. Optimization techniques can be broadly classified into exact and approximate methods[61]. Exact methods always obtain the optimal solution, this includes methods such as branch and bound, dynamic programming [28]. On the other hand, approximate methods can be split into sub-groups, approximation algorithms, and heuristics. The former provides a provable solution in a reasonable execution time [61]. The latter subgroup involves finding an acceptable solution in a reasonable amount of time [28]. The heuristics algorithms are generally very problem specific while the metaheuristics are broader and can be applied to various categories of optimization problems.

Metaheuristics typically employ a set of high-level strategies to guide the underlying heuristics in the search for optimal or near-optimal solutions [28]. These strategies may include neighborhood search, stochastic search, or evolutionary search [61]. Metaheuristics have been successfully applied in various fields, including engineering, finance, and operations research, to solve complex problems that are difficult or impossible to solve with exact methods [61].

3.9 Sorting Algorithms

Sorting algorithms rearrange a list of elements in a certain order [24] [38]. Sorting algorithms are among the most utilized algorithms in computer science, as handling sorted and structured data is more efficient than handling randomized data [38]. Attributes to consider in sorting algorithms are time complexity, stability, and space complexity [38]. The question of what sorting algorithm is most efficient depends on the context in which the algorithm will be utilized [38]. In this section, different algorithms are presented, namely quick sort, merge sort and insertion sort. An important consideration is the initial order of the elements, a nearly sorted initial order, and a random initial order calls for different sorting algorithms [38][24].

3.9.1 Quicksort

Quicksort is a divide-and-conquer algorithm that sorts an array or list of items [68]. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot [66]. The pivot element is then in its final position in the sorted array. The two sub-arrays are then recursively sorted using the same process [68].

The steps of the Quicksort algorithm are:

1. Select a pivot element from the array.
2. Partition the array into two sub-arrays, one with elements less than the pivot, and the other with elements greater than the pivot
3. Recursively sort the sub-array of elements less than the pivot.
4. Recursively sorts the sub-array of elements greater than the pivot.
5. Combine the sorted sub-arrays and the pivot element to obtain the final sorted array.

The choice of the pivot element can affect the performance of the algorithm [66]. A common strategy is to choose the pivot as the last element of the array, but other strategies such as choosing the first, middle, or random element can also be implemented [66].

The worst-case time complexity of Quicksort is $\mathcal{O}(n^2)$, but when implemented well with good pivot choice, it has an average time complexity of $\mathcal{O}(n * log(n))$ which makes it a fast sorting algorithm [68]. Quicksort's fast average time complexity makes it well-suited for sorting large, randomly-ordered data sets [68].

3.9.2 Merge sort

Merge sort is a sorting algorithm that works by dividing a list of elements into smaller sub-lists, sorting those sub-lists, and then merging them back together to obtain a sorted list [66]. The steps in merge sort are as follows:

1. Divide: The algorithm begins by dividing the list into two equal halves. This process is repeated recursively until there are only one or zero elements in each half.
2. Sort: Once the list has been divided into its smallest sub-lists, the sorting process begins. Each sub-list is sorted independently using any sorting algorithm.
3. Merge: The sorted sub-lists are merged back together in pairs. The elements in each pair are compared, and the smaller element is moved to a temporary list. This process is repeated until all the elements have been merged into the temporary list.

4. Final merge: The temporary list is merged back into the original list, with the elements being placed in sorted order. The algorithm continues recursively until the entire list has been merged and sorted.

The key benefit of merge sort is its efficiency. It has a time complexity of $\mathcal{O}(n * log(n))$, which means that it can sort a list of n elements in an average of log n steps [68]. This makes it faster than many other popular sorting algorithms, such as quick sort and insertion sort, which have a time complexity of $\mathcal{O}(n^2)$ [68]. Another benefit is that merge sort is a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the original list [68].

Despite its efficiency and stability, merge sort does have some limitations. One of the main limitations is its space complexity. Because merge sort requires the creation of temporary lists during the merge step, it can require a large amount of additional memory compared to other sorting algorithms [68]. There is also important to consider that the time complexity is given for the worst case scenario, for the best case scenario the merge sort is close to the worst case because of the built in robustness in the algorithm that preforms the same operations for each data set [68].

3.9.3 Insertion sort

Insertion sort is a simple sorting algorithm that works by maintaining a sorted sublist in the lower positions of the list [68]. It repeatedly iterates through the list by removing one element at a time and inserting it into the correct position in the sorted sublist [18].

The steps of the Insertion sort algorithm are:

1. Initialize an empty "left" subarray and an unsorted "right" subarray with the input array.
2. Starting from the second element, iterate through the right subarray.
3. For each element of the right subarray, compare it with each element of the left subarray, starting from the rightmost element.
4. Move the elements of the left subarray from one position to the right until the correct position of the current element is found.
5. Insert the current element in the correct position within the left subarray.
6. Repeat steps 2 through 5 for each element of the right subarray.
7. Once all elements have been processed, the left subarray will be the fully sorted version of the input array.

Insertion sort has a time complexity of $\mathcal{O}(n)$ in the best case when the algorithm is already sorted, and $\mathcal{O}(n^2)$ in the worst case[8]. It operates in place, meaning it does not require additional memory space[68]. Insertion sort is not efficient for large unsorted data sets, but it performs well for sorted or nearly-sorted data sets[68].

4 Method

In this chapter, we describe the method used to address the purpose of this paper, which is to solve the problem of matching buy and sell orders in a call auction. Building on the method model presented in chapter 2, a detailed overview of the steps involved in the method and the relevant concepts presented in the frame of reference for each step is provided. The aim is to provide a clear and concise description of the method used for the study. To help guide the reader, figure 4.1 is included, which provides a visual representation of the method model.

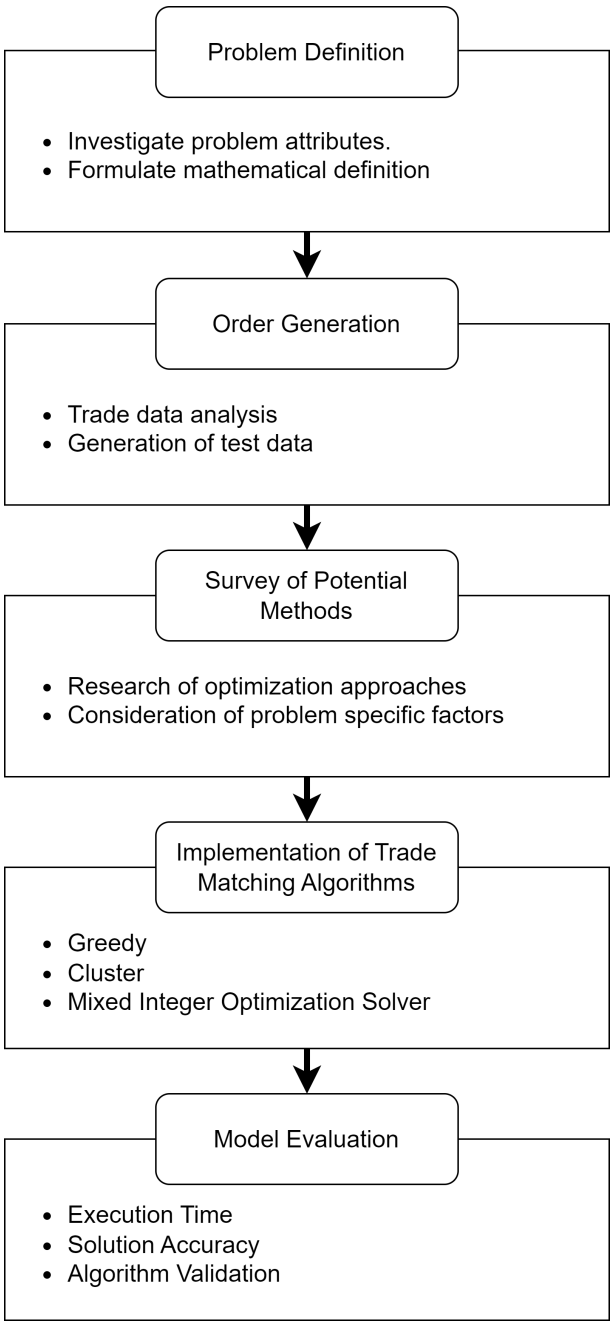

Figure 4.1: Diagram of the method

4.1 Problem Definition

The situation described in the introduction chapter 1 and the theory presented in chapter 3 has led to a mathematical representation of the problem. The problem is to minimize transaction costs of trades in a call auction. Each buy and sell order is represented by a tuple of the form (id, q) where id is a unique identifier for each order and q the quantity represented by B_i for buy orders and S_j for sell orders. I is a set of buy orders and J was a set of sell orders. B_i and S_j were defined as integers $\forall i, j$. Parameter t denotes the cost of one transaction, which for this problem is constant. x_{ij} denotes the value transferred from a buy order B_i to a sell order S_j. y_{ij} is a binary variable which denotes whether a transaction has taken place between buy order B_i and sell order S_j, takes value 0 if $x_{ij} = 0$ and value 1 if $x_{ij} > 0$. The optimization problem was defined as

$$min \quad T = \sum_{i \in I}^{I} \sum_{j \in J}^{J} t y_{ij} \tag{4.1a}$$

$$s.t. \quad \sum_{i \in I}^{I} x_{ij} = B_i \; \forall i \in I \tag{4.1b}$$

$$\sum_{j \in J}^{J} x_{ij} = S_j \forall j \in J \tag{4.1c}$$

$$x_{ij} \leqslant y_{ij} * min(\, B_i \,, \;\; S_j \,) \; \forall \; i \in I, \; \forall \; j \in J \tag{4.1d}$$

$$y_{ij} \in [0, 1] \; \forall \; i \in I, \; \forall \; j \in J \tag{4.1e}$$

$$x_{ij} \in \mathbb{R} \; \forall \; i \in I, \; \forall \; j \in J \tag{4.1f}$$

where the goal of the optimization was to reach a solution as close to optimal as possible in a reasonable amount of time. One should observe that all constraints as well as the target function are linear and the problem can therefore be defined as a MIP-problem. The mathematical formulation presented above was used for the implementation of existing solvers, both commercial and open source. The presented optimization problem is an order matching problem.

The theoretical upper and lower bound of the general optimization problem can be derived with ease, the lower bound and the best theoretical solution would be if each buy order is matched with exactly one sell order. That would mean that the lower bound is the maximum of the amount of buy and sell orders. The worst case theoretically would be if all orders are divided into the smallest possible constituents, meaning that each transaction would only be with one instrument and the upper bound is theoretically equal to the total traded volume. For each generated data set specific upper and lower bounds can be derived to further tighten the ambit of the optimal solution.

One possible interpretation of the problem of minimizing transactions between two lists with the same total volume is that it can be defined as a k-MCIP. In fact, since there is only one k value for all cases, it can be further narrowed down to a 2-MCIP. This type of problem presents a significant challenge in optimization, and while heuristic approaches have been developed, they come with high time complexity [42].

2-MCIP are known to be NP-Complete [13]. The problem presented in 4.1 was defined as a mixed integer optimization problem and was, therefore, NP-Complete. The implications of NP-Complete problems is that in the worst case scenario, the problem can not be solved in polynomial time with any known algorithm [34].

4.2 Order Generation

To accurately test the optimization algorithms, high-quality data is crucial. In this study, the data required was the quantity and a unique identifier for each order in the form (id, q). Since data from call auction pricing was not publicly available, the closest available data was trade data, the trade data was analyzed and then data were to reflect reality.

To comprehend the salient features of the required data, an analysis of closed trades from ten companies included in the OMXS30 index was conducted. The trade data under consideration was procured from continuous trading over three days for ten distinct stocks. It should be emphasized that trade data differs from order data, as trades reflect matches that have already been identified and an order may have been segmented into smaller sizes. Nonetheless, it can be assumed that the trade size corresponds to the size of at least one of the sell or buy orders.

To analyze the data, the average volume traded for each stock was calculated. In addition, the frequency of all order volumes was ascertained to determine whether any order size was more common than any other. All orders were also ranked based on their occurrence frequency, following which the most common order volumes were compared for all ten stocks. Furthermore, an analysis of all trades was conducted to investigate the distribution of numbers that denote trade volumes, specifically to examine the prevalence of trades that end with zero or other digits. Finally, the frequency of trades that end with two or three zeros was evaluated with the aim of comprehending the likelihood of placing orders with these particular values.

For each stock, the trades were randomly separated into two lists of equal length. To make sure that the lists had the same volume random elements were removed from the list with the larger total volume until both lists had the same total volume. If the last order removed was larger than the gap between the two lists, the order was cut to ensure that only a necessary amount of orders were removed. This process was repeated for all ten stocks, resulting in the creation of ten buy order lists and ten corresponding sell order lists. The resulting data sets will be utilized to assess the performance of the algorithms with real-world data, albeit with some limitations.

The obtained data is a representation of real-life data, however, it may contain certain flaws that do not capture all the different data characteristics such as size and distribution. In order to mitigate this issue, additional data sets were generated and tested to identify potential flaws in the algorithms. The generated data sets are described below, and modifications were made after the analysis to better capture some key characteristics in the data from the stock trades.

In addition to the data from the trade data list that's been separated a uniform probability distribution was applied to generate the order quantities for additional data sets. The algorithm used ensured that the total quantity on the buy and sell sides was equal.

To generate the data, Python [64] was used as the programming language, and the Python library NumPy was used to generate data in the desired uniform distribution. Each scenario was stored in a .csv file to enable multiple runs on the same data.

To examine how the optimization algorithms cope with increasing data size and complexity, four different data sizes was generated for each distribution: 100, 1000, 10,000, and 100,000 orders. The size was the sum of the amount of buy and sell orders. Therefore the number of buy and sell orders does not have to be equal to only the total quantity in the buy

order and the total quantity of sell orders needs to be equal.

The first distribution applied was a uniform distribution called Uniform Distribution 1 with a mean from the analyzed data, divided by the amount of buy and sell orders respectively, and a range from 1 to $mean * 2 - 1$. The second distribution called Uniform Distribution 2 was also a uniform distribution with a mean of double the size of the first distribution.

All data scenarios were re-sampled 10 times for 10 unique data sets for each scenario. A representation of all data sets that were generated is presented in table 4.1 below.

Table 4.1: Generated data sets

	Uniform Distribution 1	**Uniform Distribution 2**
Size	**Number of data sets generated**	
100	10	10
1,000	10	10
10,000	10	10
100,000	10	10
Total for each distribution	40	40
Total amount of data sets	80	

Data were generated with different means to test algorithms in different scenarios. For each data size and distribution 10 data sets were sampled with varying ratios between buy and sell orders. The total traded volume from the buy and sell side was still equal but the number of orders was changed according to table 4.2. The number of buy orders for each data set is listed in the table and the number of sell orders is equal to the size subtracted by the number of buy orders.

Table 4.2: Buy order ratio

	Buy order ratio									
	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%
Size	**Number of buy orders**									
100	5	10	15	20	25	30	35	40	45	50
1,000	50	100	150	200	250	300	350	400	450	500
10,000	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
100,000	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000

Overall, the generated data will provide a suitable range of orders with various quantities and distribution types that have been developed to test the optimization algorithm's effectiveness in different sizes and distributed to test the algorithms in a wide range of conditions to be able to find potential strength and weaknesses for each algorithm.

4.3 Survey of Potential Methods

A comprehensive literature survey was conducted to identify potential methods for solving the order matching optimization problem outlined in section 4.1. The survey focused on order matching optimization, mixed integer programming, linear programming, and heuristics for solving mixed integer programming. To ensure a thorough review of the available literature, Google Scholar, Scopus, and Science Direct databases were searched without any limitations on the publication year or specific journals. However, preference was given to

recent articles with regard to optimization solvers due to significant progress made in the past decade.

The literature survey involved an exploration of available literature on order matching optimization, mixed integer programming, linear programming, and heuristics for solving mixed integer programming. In addition, the survey sought to identify the pros and cons of available algorithms, programming languages, and solvers for mixed integer optimization.

During the literature search, an article by Chen et al. [13] was found, which explained the concept of minimum common integer partition. The problem originally arose in the field of computational molecular biology, where it was used to optimize algorithms for fingerprint matching. While the problem described in the article was similar to the one addressed in this paper, the data sizes in this study are significantly larger. Additionally, this paper focuses solely on matching two lists, whereas Chen et al. also addressed the problem with multiple lists.

Additional research has been conducted to enhance the solution for the MCIP-problem. The most recent published study, by Guohui and Weitian [42], developed an algorithm that leveraged commonly occurring elements and combinations of elements, identifying matches for these elements first before employing a greedy algorithm. Despite ongoing research efforts to improve the MCIP-problem, the field still has a relatively low number of published articles compared to other well-known optimization problems.

The literature survey revealed several methods for solving mixed integer optimization problems, including the greedy algorithm, and mixed integer programming using cutting-plane, branch-bound, or rounding. Although these methods have been well-researched and applied to a variety of applications, there is a risk that the problem definition may have flaws or that there may be other relevant models to solve the problem. Therefore, each model was triangulated with multiple sources to obtain a deep understanding of its possibilities, as well as its pros and cons. However, since no article describing a model implemented for the proposed optimization problem was found, the presented methods were deemed relevant according to the literature studied and the theory presented in chapter 3.

The result of the literature study was that the available MIP-solvers are well suited for solving general optimization. In each MIP-solver a number of heuristics were already implemented and therefore will not be implemented separately in the following method. The literature suggested that a simple approach with problem-specific characteristics can be effective for solving an optimization problem. Together with the information about the simplicity of the greedy algorithm and the possibility of using metaheuristics, six different heuristics were built on a greedy foundation, and a MIP-solver was evaluated.

4.4 Implementation of Order Matching Algorithms

This section outlines the implementation of order matching algorithms used in this study. Python is the chosen programming language for all tests, and they are run on the same machine. Google's programming toolbox, OR Tools [53], is used to implement the SCIP solver in the Python environment, while GUROBI's Python interface was used to implement the GUROBI solver. Pseudo-code for each order matching algorithm is presented in the following section, which represents the logic of the implemented algorithms. The algorithms' selection process originates from the study's aim to find a solution to the order matching problem with low transaction costs and execution time.

In order to prevent an algorithm or solver from running indefinitely, a runtime stop criterion was applied to limit each solution. Unless otherwise specified, a ten-minute stop criterion was set for all algorithms and solvers. The authors determined that an execution time exceeding ten minutes would not be feasible to implement. Algorithms that fail to identify a feasible solution within ten minutes will be considered unable to find a solution within a reasonable amount of time.

4.4.1 Heuristics

Greedy algorithms are one type of heuristic approach to solving a decision problem where something needs to be distributed amongst multiple buckets. In this report, four different versions of greedy algorithms and four cluster algorithms will be implemented and evaluated. The eight methods are all based in a greedy approach and are described in the section below. The eight algorithms have an ascending order of complexity where the first algorithm is the simplest one.

The eight greedy algorithms investigated in this report are: Unsorted, Sorted, Repeated Sort, Repeated Sort, and Match, Cluster 2-1, Cluster 2-1 & 1-2, Cluster 3-1, and Cluster 3-1 & 1-3. The differences between the algorithms are illustrated in table 4.3 below. All algorithms are further explained in the following section with an explanation of how each algorithm has been implemented.

Table 4.3: Comparison of six greedy algorithms

	Ordered list	**Repeated sort**	**Perfect match**	**Cluster matching**
Unsorted	No	No	No	No
Sorted	Yes	No	No	No
Repeated Sort	Yes	Yes	No	No
Repeated Sort and Match	Yes	Yes	Yes	No
Cluster 2-1	Yes	Yes	Yes	Yes
Cluster 2-1 & 1-2	Yes	Yes	Yes	Yes
Cluster 3-1	Yes	Yes	Yes	Yes
Cluster 3-1 & 1-3	Yes	Yes	Yes	Yes

Unsorted Algorithm

The unsorted algorithm starts with two unsorted lists of buy and sell orders. For each iteration, the first element in the buy list will be matched with the first element in the sell list. If the buy (sell) order is smaller than the sell (buy) order, the buy (sell) order is subtracted from the sell (buy) order. A transaction is saved in the format $(buy\_id, sell\_id, quantity)$ The buy (sell) order is then removed from the buy (sell) list. The algorithm is complete once every order has been matched up, i.e. when the buy and sell lists are empty. In figure 4.2 the sorted version of the greedy algorithm is visualized. In algorithm 1 a pseudo code example of the Unsorted algorithm can be found that generally explain hoew the algorithm can be implemented in an arbitrary code language.

Sorted Algorithm

The sorted algorithm shares the procedure with the unsorted algorithm, with the exception that the buy and sell order lists are sorted before matching. The Quicksort algorithm is used to sort the buy and sell sides in descending order. Figure 4.2 illustrates the sorted algorithm's procedure.

Algorithm 1 Unsorted Greedy Algorithm

Require: *buy_orders, sell_orders*

while there are sell orders or buy orders remaining **do**
    if buy order volume > sell order volume **then**
        add transaction with sell order to list
        update buy order volume
    else if buy order volume < sell order volume **then**
        add transaction with buy order to list
        update sell order volume
    else
        add transaction with buy order to list
    end if
end while
create numpy array from list of transactions
return transactions array

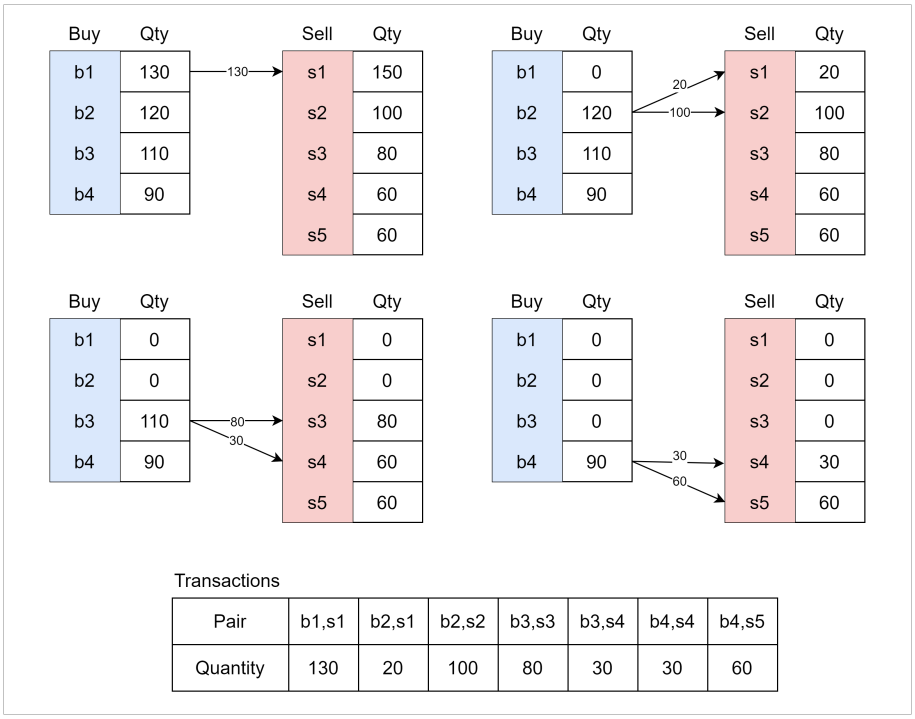

Figure 4.2: Greedy visualized

A representation of the logic used when implementing the greedy algorithm illustrated above can be found in the pseudo code presented in algorithm 2. The greedy algorithm pseudo code, shows an example of how the greedy algorithm can be implemented.

Algorithm 2 Sorted Greedy Algorithm

Require: *buy_orders*, *sell_orders*
1: Sort buy orders in descending order of volume and store in *sorted_buy_orders*.
2: Sort sell orders in descending order of volume and store in *sorted_sell_orders*.
3: **while** there are sell orders or buy orders remaining **do**
4: **if** buy order volume > sell order volume **then**
5: add transaction with sell order to list
6: update buy order volume
7: **else if** buy order volume < sell order volume **then**
8: add transaction with buy order to list
9: update sell order volume
10: **else**
11: add transaction with buy order to list
12: **end if**
13: **end while**
14: **return** transactions array

Repeated Sort Algorithm

In contrast to the algorithms previously introduced, the repeated sort algorithm ensures that the buy and sell order lists are always sorted by inserting the leftover value of an incomplete match of a buy and sell order. The largest orders are thus always positioned at the top of the list. Once the lists are sorted, the biggest buy order is once again matched with the biggest sell order in the same way as the first iteration. This process is repeated until both lists are empty, and all orders have been matched. The logic of the algorithm is represented in pseudo code algorithm 3 below.

Algorithm 3 Repeated Sort Greedy Algorithm

Require: *buy_orders*, *sell_orders*
1: *sorted_buy_orders* ← sort *buy_orders* by descending *Volume*
2: *sorted_sell_orders* ← sort *sell_orders* by descending *Volume*
3: **while** there are sell orders or buy orders remaining **do**
4: **if** buy order volume > sell order volume **then**
5: add transaction to *transactions*
6: update buy order volume
7: insert buy order into sorted list
8: **else if** buy order volume < sell order volume **then**
9: add transaction to *transactions*
10: update sell order
11: insert sell order into sorted list
12: **else**
13: add transaction to *transactions*
14: **end if**
15: **end while**
16: **return** transactions array

Repeated Sort and Match Algorithm

The repeated sort and match algorithm follows the same procedure as the repeated sort algorithm, however, it aims to find perfect matches between buy and sell orders for each iteration. To achieve this, the algorithm uses binary search to search through both lists to identify any matches between the two in each iteration. If a match is found, it is reported as a transaction and added to the transaction list, and the corresponding buy and sell orders are removed from their respective lists. An example of how the algorithm could be implemented is presented in pseudo code algorithm 4.

Algorithm 4 Repeated Sort & Match Algorithm

```
Require: buy_orders, sell_orders
sorted_buy_orders ← sort buy_orders by descending Volume
sorted_sell_orders ← sort sell_orders by descending Volume
while there are sell orders or buy orders remaining do
    if buy order volume > sell order volume then
        add transaction
        update buy order volume
        foundmatch ← PerfectMatchFast()
        if foundmatch then
            add transaction
        else
            insert buy order into sorted list
        end if
    else if buy order volume < sell order volume then
        foundmatch ← PerfectMatchFast()
        if foundmatch then
            add transaction
        else
            insert sell order into sorted list
        end if
    else
        add transaction
    end if
end while
return transactions array
```

Cluster Match Algorithm

The cluster match algorithms intend to find one-to-one, two-to-one and three-to-one combinations between the buy and sell orders, i.e. matching one buy order to one sell order of the same size, two sell orders that amount to one buy order, and three sell orders that amount to one buy order. In figure 4.3 below the one-to-one, two-to-one and three-to-one matches are visualized.

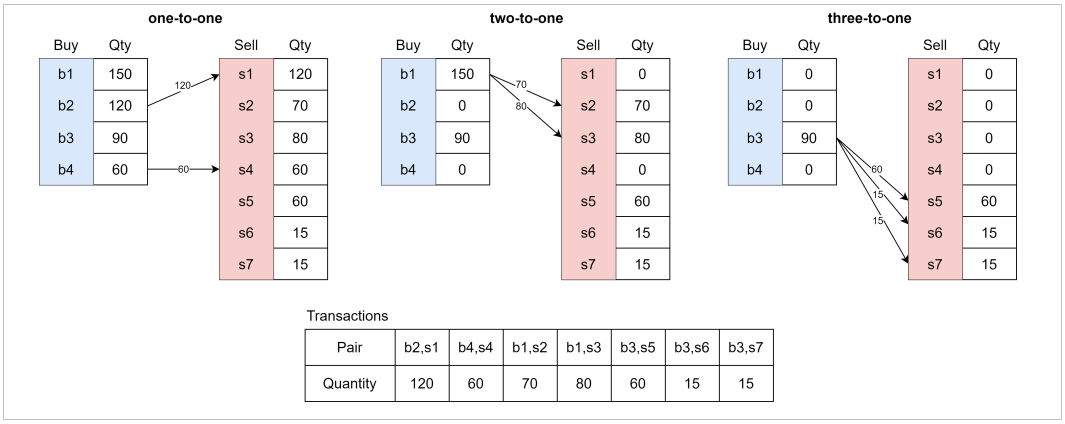

Figure 4.3: Cluster Algorithm visualized

There are four variations of the cluster algorithm:

1. Cluster 2-1: This version identifies pairs of orders in the longer list that, when combined, equal one order in the shorter list.
2. Cluster 2-1 & 1-2: In addition to the Cluster 2-1 procedure, the Cluster 2-1 & 1-2 reverses the process by also finding pairs of orders in the shorter list that add up to orders in the larger list.
3. Cluster 3-1: Adding onto the cluster 2-1 procedure, this version also identifies all three-to-one matches in one direction, where three orders from the longer list combine to match one order in the shorter list.
4. Cluster 3-1 & 1-3: In addition to the cluster 2-1 & 1-2 procedure, this version identifies three-to-one matches in both directions.

The cluster algorithms create an index of the buy and sell orders, utilizing a Python dictionary where the keys are order volumes, and the value is a list of orders of that volume. Using an index ensures that orders can be found in $O(1)$ time. Orders from the list of greater size are searched through in order to find combinations of orders that amount to the same volume as an order from the smaller list. Once all combinations have been identified and transactions between orders have been recorded, the remaining unmatched orders are matched using the Repeated Sort and Match algorithm.

Algorithm 5 below displays in pseudo code the function which is utilized in the cluster algorithms to find two-to-one matches by finding two orders in a dictionary that sum to a specified target volume.

Algorithm 5 Find Two Numbers That Sum to Target

```
1:  procedure FIND_TWOPLETS(nums_dict, target)
2:      nums ← keys of nums_dict as list
3:      for each num in nums do
4:          complement ← target − num
5:          if complement is in nums_dict then
6:              if complement ≠ num or length of nums_dict[complement] > 1 then
7:                  return [num, complement]
8:              end if
9:          end if
10:     end for
11:     return None
12: end procedure
```

Algorithm 6 presented in pseudo code below outlines the function used in the cluster algorithms to find three-to-one matches by finding three orders in a dictionary that together sum to a specified target volume.

Algorithm 6 Find Three Numbers That Sum to Target

```
1:  procedure FIND_TRIPLETS(nums_dict, target)
2:      nums ← keys of nums_dict as list
3:      for each i in the range of length of nums do
4:          for each j in the range from i to length of nums do
5:              complement ← target − nums[i] − nums[j]
6:              if complement is in nums_dict then
7:                  if orders exist in the dictionary then
8:                      return [nums[i], nums[j], complement]
9:                  end if
10:             end if
11:         end for
12:     end for
13:     return None
14: end procedure
```

In algorithm 7 a pseudo code representation over the general logic of the main function is represented. The algorithm 7 illustrates which path the algorithms go through depending on the version of the cluster algorithm.

Algorithm 7 Cluster Algorithms

Require: *buy_orders*, *sell_orders*, *version*
where: version 1: 2-1, version 2: 3-1, version 3: 2-1 & 1-2, version 4: 3-1 & 1-3
buy_order_index, *sell_order_index* ← index(*buy_orders*, *sell_orders*)
transactions ← one_to_one(*buy_orders*, *sell_orders*)
if |*buy_orders*| < |*sell_orders*| **then**
    transactions ← two_to_one(*buy_order_index*, *sell_order_index*)
    if *version* == 3 or *version* == 4 **then**
        transactions ← one_to_two(*sell_order_index*, *buy_order_index*)
    end if
    if *version* == 2 or *version* == 4 **then**
        transactions ← three_to_one(*buy_order_index*, *sell_order_index*)
    end if
    if *version* == 4 **then**
        transactions ← one_to_three(*sell_order_index*, *buy_order_index*)
    end if
else
    transactions ← one_to_two(*buy_orders*, *sell_orders*)
    if *version* == 3 or *version* == 4 **then**
        transactions ← two_to_one(*sell_order_index*, *buy_order_index*)
    end if
    if *version* == 2 or *version* == 4 **then**
        transactions ← one_to_three(*buy_orders*, *sell_order_index*)
    end if
    if *version* == 4 **then**
        transactions ← three_to_one(*sell_order_index*, *buy_order_index*)
    end if
end if
unmatched_buy, *unmatched_sell* ← remaining unmatched orders
transactions ← greedy(*unmatched_buy*, *unmatched_sell*)
return *transactions*

4.4.2 Mixed Integer Programming

In order to implement the optimization model presented in Section 4.1, the optimization library from OR Tools in Python was utilized, as well as GUROBI's Python interface. OR Tools supports various solvers, including both open source and commercial solvers, as well as MIP-solvers and LP-solvers. Multiple solvers were implemented, including the commercial solver GUROBI and the open-source solver SCIP. These solvers were chosen based on their availability and compatibility with OR Tools.

The optimization model that was implemented corresponds to the one presented in equation 4.1, which was derived from the order matching problem description. The implementation involved translating the mathematical model into code, defining decision variables and constraints, and invoking the solver to solve the optimization problem.

To ensure the correctness of the implementation, several tests were conducted to verify the results of the model. These tests involved checking that the output matched the expected values for a given input. Additionally, the implementation was tested with different input data to evaluate its robustness and performance under varying conditions. Overall, the implementation of theMIP-solver was a crucial step in solving the order matching problem, and the choice of solver played a significant role in the efficiency and accuracy of the results.

4.5 Model Evaluation

The performance evaluation of the optimization models will focus on the quantitative performance. The quantitative aspect involves measuring the execution time of the algorithms and assessing how close they are to the optimal solution. This evaluation provides insight into the algorithms' performance in terms of efficiency and accuracy.

4.5.1 Algorithm Performance Evaluation

To get an accurate representation of the results each model was tested on the same data set 10 times to avoid discrepancies in a single run affecting the results. 10 samples were chosen to ensure a statistically significant number of tests while minimizing the impact of random variation on the results. The purpose of multiple tests on the same data set is to investigate if the algorithms find different solutions for the order matching problem or if the time to obtain the solution differs. The solutions should be consistent between runs, but the execution time will differ slightly. For each distribution and size, including the 10 real data sets, the tests are run 10 times to attain more certainty in the results. The total number of runs is therefore 100 for each algorithm for a given distribution and size, for a total of 900 unique tests for each algorithm.

The collected data for each test was, execution time, number of transactions, a transaction list, and for the MIP-solvers number of branches. The time and number of transactions will be plotted on a graph and the algorithms will be mainly compared on these parameters. Although it was important to collect the transaction list that contains how many instruments that are sent and between which actors. This list was collected for validation purposes to determine that the solution was feasible. This was done by summarizing all transactions and comparing the sum with the total quantity on the buy and sell side. The number of branches was obtained with the purpose of comparing the number of branches and execution time.

To measure the execution time for each algorithm at each test the stopwatch method was applied using the Python function perf_counter() for high granularity. The stopwatch method was used for all tests in a manner that measured the actual elapsed execution time of the algorithms. The time used to compare the algorithms is the average time for each data size and distribution. Since one data set was tested 10 times and for each data size 10 randomly generated data sets for each distribution were generated. This resulted in 100 samples of execution time for each data size and distribution. The average execution time for each data set was used for comparison between the different algorithms.

The solutions generated are benchmarked against the lower bound for the specific data set. The lower bound for a data set is the number of orders in the largest of either the buy or sell order lists, this is generally lower than the optimum, but provides a solid benchmark to test against.

After the tests for all algorithms were completed, student t-tests were carried out to compare how much each algorithm differed in time for each data set for all algorithms. The t-tests were carried out at a significance level of 95%. The test is made to investigate if one algorithm's execution time is significantly lower than another algorithm in a specific data set.

4.6 Model Validation

In order to assess the effectiveness of the developed algorithms, it is crucial to verify that the resulting solutions are feasible solutions that satisfy all constraints outlined in equation 4.1. To ensure the validity of the algorithms, all solutions will undergo a thorough verification

process. This process ensures that no order sends more than the smallest size of the buy or sell order included in the trade and that each order does not exceed its specified size for all trades the order is included in. Moreover, checks are performed to ensure that the total volume sent from the sell side is equivalent to the total sell volume, and the same holds for the buy side. Any algorithm found to violate a constraint will be flagged, and further investigation will be conducted to determine the root cause of the violation. Performance evaluation will only be performed on tests that satisfy all constraints. Any tests that violate a constraint will be presented in the validation section (5.7) of the results chapter (5).

To ensure the stability of the results and mitigate the potential impact of outliers, each algorithm will be tested ten times for each data set. This will enable a comprehensive performance evaluation that considers the average execution time for each algorithm and data set. This approach is aimed at increasing the reliability and robustness of the results.

5 Results and Analysis

The goal of this chapter is to present the results and analysis of the study, with a focus on data analysis, algorithm performance, and algorithm validation. In the first section, the data analysis is presented, including a detailed examination of the data characteristics and distribution. The second section is dedicated to the performance of the algorithms, including a discussion of their efficiency and accuracy. Finally, the third section provides an overview of the algorithm validation process, including the testing and evaluation of the algorithms against a set of predetermined criteria. The combination of these three sections will provide a comprehensive picture of the results and analysis of the study and will enable a deeper understanding of the problem and the proposed solutions.

5.1 Data Analysis

In the following section the results from the data analysis conducted on the obtained trade data from ten stocks included in the OMXS30 index. The included stocks can be found in the table 5.1 below. The obtained data is the volume from completed trades. The volume from the trade data is the best available data since it provides the volume from at least one of the buy and sell sides. The RIC codes that were used to retrieve the data from Refintiv Eikon as well as the time frame from which the trade data are from. All data were downloaded on the same day 2023-04-04 at 13:00.

Table 5.1: Stock trade data

	ABB	ERIC	INVE	HEXA	VOLV	SBB	NIBE	SHB	EVOG	AZN
RIC	ABB.ST	ERICb.ST	INVEb.ST	HEXAb.ST	VOLVb.ST	SBBb.ST	NIBEb.ST	SHBb.ST	EVOG.ST	AZN.ST
Retrieved Data	2023-03-28 09:00 – 2023-03-28 17:30									

All trades for all ten stocks were analyzed. The results from the analysis were used to modify the generated data to better reflect real-life data and to gain a better understanding of the characteristics of the order volume in trade data. The analysis included all trades over three days for ten stocks with a wide range in how many trades were made and the price at which the trades were traded. The results of the analysis include the mean volume for each stock, the most common volumes traded for each stock, how common it is for one volume to occur more than once, and an investigation of the last digits of the traded volume.

The first analysis of the trade data was to investigate the average traded volume and the average stock price the stock has been traded on. In table 5.2 below all ten analyzed stocks are presented with the number of trades, average volume, and price for each stock.

Table 5.2: Average traded volume and average price for each stock

Stock	**ABB**	**ERIC**	**INVE**	**HEXA**	**VOLV**	**SBB**	**NIBE**	**SHB**	**EVOG**	**AZN**
No. of Trades	2,601	5,998	10,686	3,333	6,071	23,066	5,074	1,915	5,520	3,443
Volume	213	1,033	243	1,378	335	1,368	482	97	52	66
Price (SEK)	334	57	199	113	203	13	113	106	1,312	1,426

Significant differences were found in the average traded volume among the analyzed stocks in table 5.2. A negative correlation was observed between the stock price and the average traded volume, which can be attributed to the total value of the trade. The total value of a trade is calculated by multiplying the stock price with the traded volume. For instance, if a trader has a fixed amount of money to invest, they will need to buy a larger volume of a lower-priced stock to reach the same total traded value as a higher-priced stock. Hence, the total traded value plays a crucial role in explaining the average traded volume.

The subsequent phase of data analysis aimed at determining the most frequently occurring volume sizes for each stock and investigating whether there were any common volumes across the ten stocks. The table 5.3 presents the ten most common volume sizes for each stock. Although all trades were analyzed to identify the most common volume sizes, the top ten for each stock are presented as a representative sample.

Table 5.3: Most common traded volumes for each stock

	ABB	**ERIC**	**INVE**	**HEXA**	**VOLV**	**SBB**	**NIBE**	**SHB**	**EVOG**	**AZN**
#1	300	1500	2	700	120	1000	700	5	60	75
#2	1	333	1	130	400	47	162	10	1	1
#3	3	100	5	487	2	100	1	15	54	2
#4	5	1	3	1	1	46	75	1	2	36
#5	2	50	93	351	5	1	2	2	20	50
#6	100	300	400	500	4	10	5	3	28	37
#7	4	215	10	226	3	50	3	700	4	10
#8	896	500	4	79	10	39	4	4	3	3
#9	19	350	366	863	364	200	100	9	10	100
#10	448	10	25	2	145	500	200	6	50	70

The analysis revealed that the most common trade volume is one stock, with other singular digit order sizes also occurring frequently across all stocks, but more so for those with relatively high stock prices. In an attempt to explain the variation in order volumes across stocks, the common order sizes were multiplied by the prices at which they were traded. The results showed that in many cases, the value of the trade was close to a large even number, such as multiples of 10^x, where x is a number between zero and six. However, despite these findings suggesting a correlation between trade volumes and total trade value, the results were inconclusive and no definitive conclusions can be drawn.

In order to gain a better understanding of the frequency of order volumes, an analysis was conducted to investigate how often each order volume appeared in the data set. The results are presented in Figure 5.1, which includes the percentage of order volumes that occur only once, twice, and so forth.

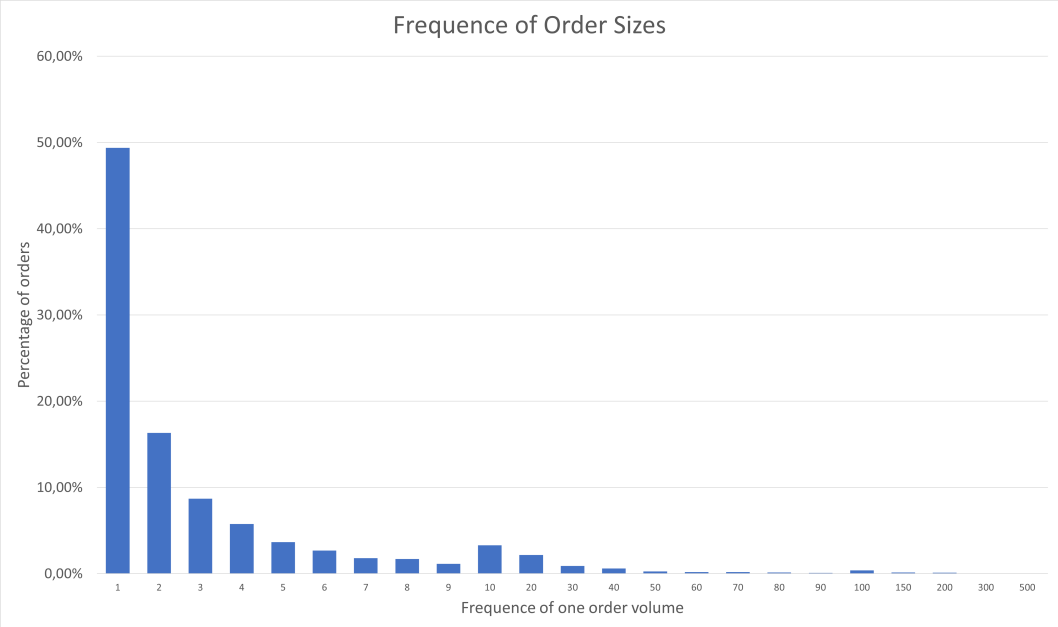

Figure 5.1: Frequency of order sizes. On the x-axis the number of times an order volume has occurred and on the Y-axis the percentage of all orders

The analysis revealed that nearly half of the orders in the data set occurred only once, and the number of occurrences decreased exponentially as the frequency increased. This decrease was likely due to rounding errors in the data, as demonstrated in Figure 5.1. Specifically, results above ten were rounded to the nearest ten, and results above one hundred were rounded to the nearest fifty. Furthermore, less than 10% of all orders occurred more than ten times for each of the ten stocks analyzed.

The distribution of the last digit in all trades was analyzed to investigate if any particular singular digit was more common. Table 5.4 presents the frequency and distribution of the last digits. The analysis considered both singular digits as well as numbers ending in two or three zeros.

Table 5.4: Frequency distribution of last digits for trade data

Last Digits	**Frequency**	**Percentage**
0	16619	25 %
1	6258	9 %
2	5744	9 %
3	5553	8 %
4	5259	8 %
5	7601	11 %
6	4846	7 %
7	4980	8 %
8	4861	7 %
9	4545	7 %
00	7932	12 %
000	2449	4 %

The result shows that the total percentage of the trades that end with zero is 25 % which is the sum of the percentage of the numbers that end with one, two, or three zeros. The result also indicates that the other singulars are fairly evenly distributed, although 5 is the most frequently occurring singular except for 0. The distribution of the last digits was applied to the generated data to reflect the real-life data more accurately.

The distribution of the data was also investigated. No direct distribution could be identified to accurately reflect the real distribution. A plot of the order volume can be found in figure 5.2, where all ten stocks have been combined into one plot.

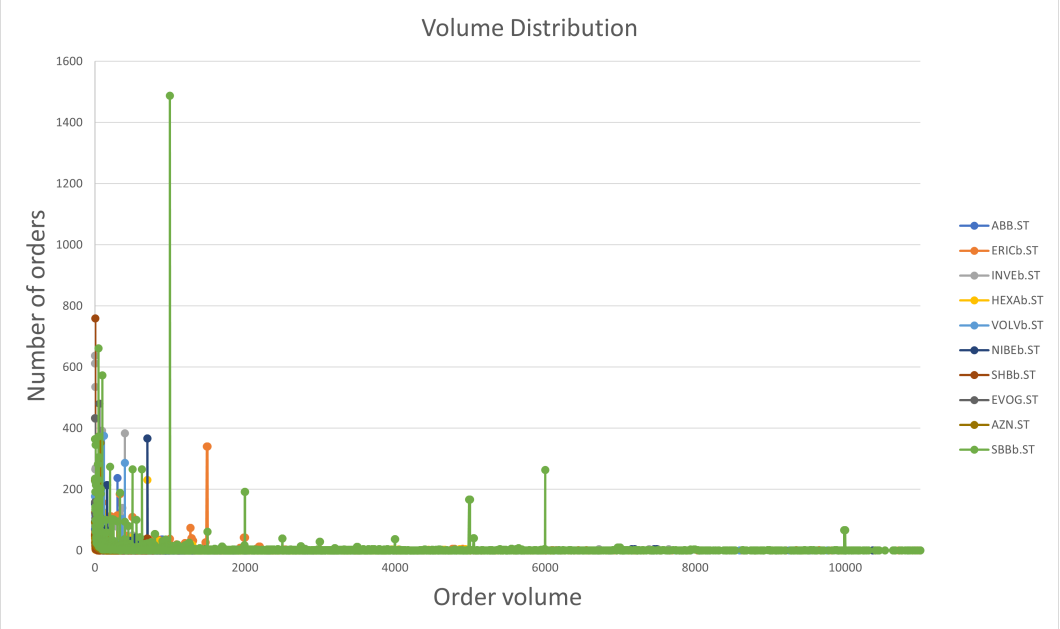

Figure 5.2: Distribution of order volumes for all ten stocks

Based on the information presented in figure 5.2, it is evident that the green line stands out prominently among the other colors. This can be attributed to the fact that it represents the SBB stock, which had a much higher number of trades compared to the other stocks during the data collection period. Although after studying all stocks individually as well the distributions are similar for all stocks.

The results shown in figure 5.2 indicate a higher frequency of smaller order sizes, with some noticeable outliers occurring more frequently. However, beyond a volume of ten, the data appears to be relatively uniformly distributed. These findings are limited in their scope, and it is difficult to draw definitive conclusions based solely on this figure. To gain a more comprehensive understanding of the distribution, it would be necessary to analyze a larger data set and differentiate the distribution for various instruments, in addition to stocks. Furthermore, it would be valuable to compare the distribution based on the price of the traded instrument.

The conducted data analysis has been utilized to modify the generated data sets, in order to better replicate real-life data. As no clear probability distribution was identified in the analysis, a uniform distribution was used to generate volume data. The most major characteristic has been deemed the distribution of ending digits, which was replicated in the generated data. Given the limited trade data available, various ratios between the buy and sell volumes were implemented to explore different potential scenarios.

5.2 Mixed Integer Optimization Solver Performance

Both the open-source SCIP solver and the commercial GUROBI solver faced challenges in finding solutions within a reasonable time frame. SCIP struggled to find feasible solutions even for smaller data sets, leading to the exclusion of its results from the evaluation. On the other hand, GUROBI managed to find feasible solutions for smaller data sets of sizes 100 and 1,000 but encountered difficulties in identifying optimal solutions within a reasonable time frame.

GUROBI experienced challenges in creating models for larger data sets. Initially, the solver struggled to build models with 1,000 orders. By replacing a big-M constraint with an indicator constraint, the model-building process became significantly faster. An indicator constraint is a feature present in the GUROBI solver, and is simply another method used to specify that the value of a binary variable is dependent on the state of a continuous variable. However, even with this improved implementation, GUROBI was still unable to build models for data sets of size 10,000 and 100,000 within a reasonable time frame.

A single run of GUROBI on a data set with 1,000 orders used the best heuristic solution as a starting solution. After running for 55 hours, GUROBI failed to find a better solution than the initial one. However, the solver was able to raise the lower bound, resulting in a 2.14% gap to the lower bound. This indicates that the starting solution is at most 2.14% away from the optimum.

Table 5.5: Gurobi run results

Metric	**Value**
Running Time (h)	55.3
Best Objective Value	608
MIP Gap (%)	2.14

GUROBI was tested further on 40 uniform data sets: 20 with a mean of 500 and 20 with a mean of 1,000, and data sizes of 100 and 1,000 orders. To avoid excessively long run times, a stopping criterion was implemented to terminate the optimization after 10 minutes. The solver recorded the upper and lower bounds, with the upper bound representing the best feasible objective value found, and the best lower bound is the best possible solution for the data set, however may not feasible. Table 5.6 displays the results from 10 data sets with a mean of 500 and a data size of 1,000, differentiated by their buy/sell ratios.

Table 5.6: Results for mean 500 and data size 1000 with different Buy/Sell Ratios

Buy/Sell Ratio	**Best Objective Value**	**Best Lower Bound**	**MIP Gap (%)**
5/95	987	945	4
10/90	983	904	8
15/85	978	850	13
20/80	962	798	17
25/75	965	755	22
30/70	965	707	27
35/65	945	652	31
40/60	947	597	36
45/55	892	562	37
50/50	869	521	40

Since the optimization solver could not identify optimal solutions within a reasonable time frame, it does not provide a reliable reference point for evaluating the heuristics. The best

heuristic solution obtained a lower objective value than GUROBI and therefore outperformed GUROBI in all but one case.

5.3 Theoretical Time Complexity of Algorithms

In this section, we present an analysis of the theoretical time complexity of the eight heuristics developed in this study. We break down the algorithms into their core functions and provide the big O-notation for each.

Unsorted

The Unsorted Greedy algorithm matches orders by iterating through the lists of buy and sell orders. The time complexity is $O(n)$, where n is the total number of orders.

Sorted

The Sorted Greedy algorithm first sorts the two lists of orders before running the matching function. The sorting algorithm used has a time complexity of $O(n \log(n))$. Along with the matching process $O(n)$, the overall time complexity is $O(n \log(n))$, where n is the total number of orders.

Repeated Sort

The Repeated Sort algorithm uses a function to insert the leftovers after an incomplete match. The insertion function employs binary search ($O(\log(n))$) to find the index where the leftover should be inserted. Since inserting into a Python list takes $O(n)$, the resulting time complexity is $O(n \log(n) + n * (n + \log(n))$. The dominating term is $O(n^2)$, which is also the overall time complexity for this algorithm where n is the total number of orders.

Repeated Sort and Match

The Repeated Sort and Match algorithm first sorts the two lists and finds one-to-one matches. The initial sorting has a time complexity of $O(n \log(n))$, and the perfect match function also runs in $O(n \log(n))$. In each iteration, if there is a leftover from an incomplete match, a one-to-one algorithm searches for orders in the opposite list with the same volume as the leftover, running in $O(\log(n))$. If no one-to-one match is found, the insertion function runs in $O(n)$. The overall time complexity is $O(n^2)$.

Cluster 2-1 and Cluster 2-1 & 1-2

The Cluster two-to-one algorithms use Python dictionaries to index buy and sell orders by volume. Creating dictionaries from a list of orders takes linear time (n). It then identifies one-to-one matches between the two lists, with a time complexity of $O(m)$, as well as the two-to-one algorithm, which runs in $O(m^2)$, where m is the total number of indexes in the dictionary. The unmatched orders are matched using the Repeated Sort and Match algorithm, which has a time complexity of $O(n^2)$. Thus, the overall time complexity of the Cluster two-to-one algorithms is $O(n^2 + m^2)$.

Cluster 3-1 and Cluster 3-1 & 1-3

The Cluster three-to-one algorithms run the one-to-one function in $O(m)$ and the two-to-one function in $O(m^2)$. It then runs the three-to-one matching function, which has a time complexity of $O(m^3)$. The unmatched orders are matched using the Repeated Sort and Match

function $O(n^2)$. The overall time complexity of the Cluster three-to-one algorithms is, therefore, $O(m^3)$. Where m is the total number of indexes in the dictionaries. The time complexity for each algorithm is presented in table 5.7.

Table 5.7: Summary of the time complexity of the algorithms

Algorithm	**Time Complexity**
Unsorted Greedy	$O(n)$
Sorted Greedy	$O(n \log(n))$
Repeated Sort Greedy	$O(n^2)$
Repeated Sort and Match Greedy	$O(n^2)$
Cluster two-to-one	$O(n^2 + m^2)$
Cluster three-to-one	$O(n^2 + m^3)$

5.4 Algorithm Performance on Generated Data

This section presents the test results of the developed algorithms. As outlined in the limitations section 1.2, the performance evaluation of the algorithms focuses on their solution accuracy and running time. The reported running time is the average duration of each algorithm's execution per data set, given that each algorithm was tested ten times. The accuracy evaluation is based on the number of transactions required between the buy and sell sides in a call auction, as the primary objective of the study is to design algorithms that minimize the number of transactions.

5.4.1 Execution Time

The execution time of the algorithms to find the optimal solution varied across all evaluated algorithms and data sets. The subsequent section presents the time performance of each algorithm on the generated data sets. However, as the Cluster 3-1 and the Cluster 3-1 & 1-3 algorithms have higher time complexities, they took significantly longer time for complex data sets. Consequently, the values for these two algorithms are excluded from figure 5.5 but are included in table 5.10.

To investigate the effect the size of the data set has on the execution time for the eight developed algorithms four different sizes have been investigated, 100, 1,000, 10,000, and 100,000. The size is the number of sell orders and buy orders combined. In figure 5.3 the four data sizes are presented, the mean for the data sets is 500 and the buy order ratio is 50 %.

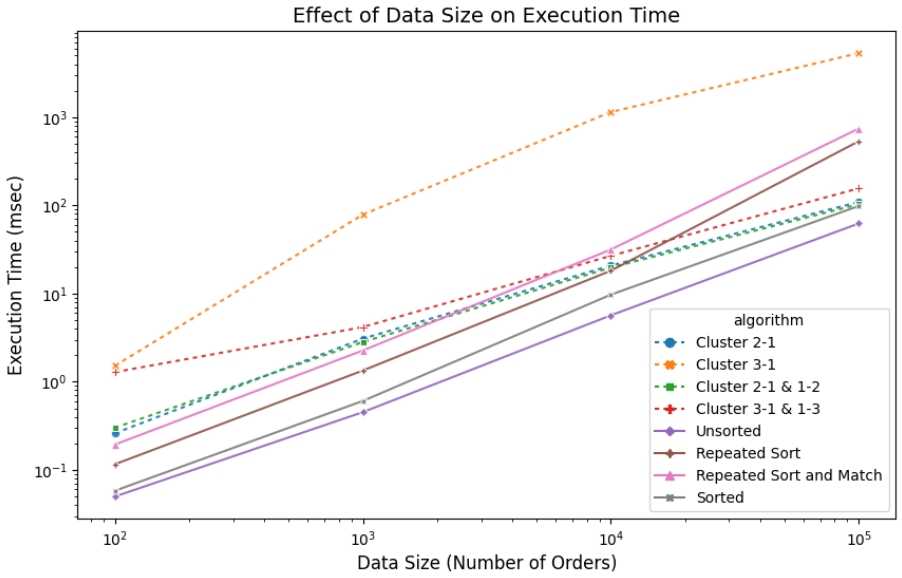

Figure 5.3: Execution time for different data sizes, mean = 500, buy order ratio 50 %.

The figure above shows that the time required to solve the problem appears similar for most algorithms until the size of the data set reaches 10,000. However, for larger data sets, there is a significant difference in the time required. It is important to note that both the x-axis and y-axis of the graph have a logarithmic scale in order to visualize the differences in results. Additionally, the theoretical time complexity of each algorithm, presented in Table 5.7, also plays a role in understanding the results. The values for each algorithm for the same data sets are included in table 5.8.

Table 5.8: Algorithm execution times (msec) for different data sizes, buy order ratio 50 %, mean = 500.

Algorithm	**Size 100**	**Size 1,000**	**Size 10,000**	**Size 100,000**
Cluster 2-1	0.26	3.08	20.76	109.22
Cluster 3-1	1.52	78.62	1144.19	5328.45
Cluster 2-1 & 1-2	0.30	2.82	19.70	103.07
Cluster 3-1 & 1-3	1.28	4.17	26.77	155.89
Unsorted	0.05	0.45	5.66	62.33
Repeated Sort	0.12	1.34	18.14	533.16
Repeated Sort & Match	0.19	2.26	31.55	743.07
Sorted	0.06	0.61	9.70	98.36

In the table above the time data for the Cluster 3-1 algorithms are also included for when the algorithms could find a feasible solution in a reasonable time. Noteworthy is that Cluster 3-1 takes longer time than Cluster 3-1 & 1-3. This is because Cluster 3-1 & 1-3 algorithm runs the full 2-1 & 1-2 logic before the 3-1 is started while the Cluster 3-1 only runs the Cluster 1-2 logic before the 3-1 match. Since the 2-1 matching process is faster than the 3-1 process it is beneficial to find as many 2-1 matches as possible before entering the 3-1 process since that process requires more time.

The data size and the ratio between buy and sell orders significantly affect the execution time of all developed algorithms. Figure 5.4 presents the execution time for all developed algorithms with a data set size of 10,000 and a mean of 500. Since the developed algorithms

do not differentiate between handling buy and sell orders, flipping the buy order ratios to test with more sell orders than buy orders is unnecessary. The key factor is how the algorithms handle cases where one list is longer than the other, and how this differs when the lists are of equal length.

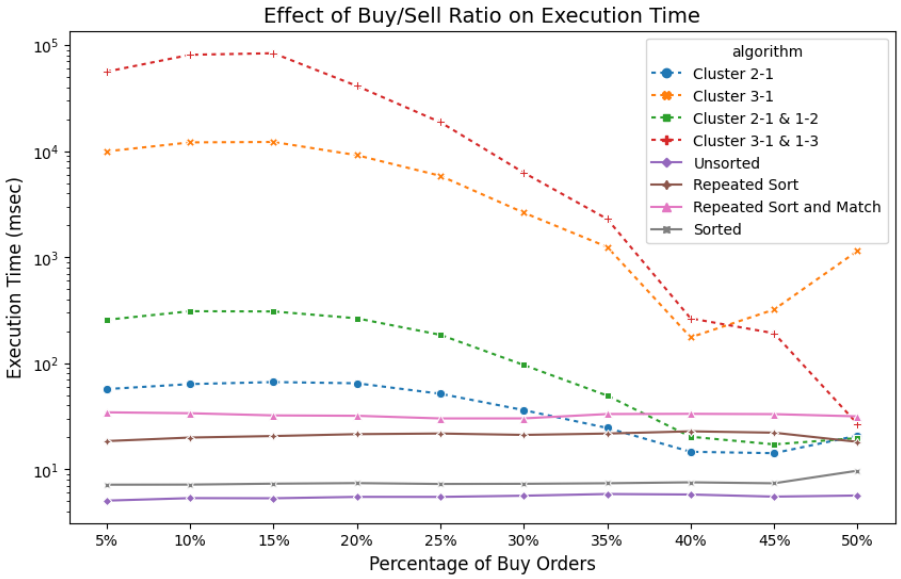

Figure 5.4: Buy order ratio for the size of 10 000 and mean = 500.

Figure 5.4 illustrates the significant increase in execution time for the Cluster 2-1 & 1-2 algorithm when the buy order ratio is longer, which is also observed for Cluster 3-1 and Cluster 3-1 & 1-3 algorithm. The longer execution time for these double-sided cluster algorithms can be attributed to the need to look through cluster matches for a longer list when performing the second cluster match, which increases the execution time compared to using the shorter list as the values for identifying cluster matches.

Moreover, the Cluster 2-1 algorithm also shows a longer execution time when the buy order ratio is larger and decreases when the ratio is closer to an even split. Notably, the Cluster 2-1 algorithm is slower than the simple greedy algorithms for smaller buy order ratios and becomes one of the quickest algorithms as the ratio increases. In contrast, the other algorithms remain stable when the ratio changes. Table 5.9 below also includes the Cluster 3-1 and Cluster 3-1 & 1-3, which displays further that the cluster algorithms, especially the 3-1 and double-sided are heavily adversely affected by a significant difference in the number of orders between the buy and sell lists.

Table 5.9: Algorithm execution times (msec) for different buy order ratios for the size of 10,000 and mean = 500

Algorithm	**10 % Buy**	**20 % Buy**	**30 % Buy**	**40 % Buy**	**50 % Buy**
Cluster 2-1	63.56	64.68	36.12	14.62	20.76
Cluster 3-1	12072.76	9138.83	2623.78	175.43	1144.19
Cluster 2-1 & 1-2	309.12	264.95	96.17	20.27	19.70
Cluster 3-1 & 1-3	80838.74	41066.71	6253.89	264.54	26.77
Unsorted	5.35	5.49	5.64	5.78	5.66
Repeated Sort	19.92	21.44	21.10	22.80	18.14
Repeated Sort & Match	33.76	31.94	30.19	33.33	31.55
Sorted	7.18	7.41	7.30	7.54	9.70

The key takeaway from figure 5.4 and table 5.9 is that the cluster algorithms, particularly the double-sided cluster algorithms and the 3-1 algorithms, are sensitive to a significant difference in the number of orders between the buy and sell lists. For this data set the Cluster 3-1 & 1-3 require longer execution time than the Cluster 3-1 which was the opposite in table 5.8

The third variation on the data that has been made is different means on the data sets. The mean of 500 comes from the data analysis on the real data where the mean of the trades for all ten stocks was close to 500. To investigate how a larger mean would affect the results a mean of 1,000 was applied to investigate the differences.

In figure 5.5 one can see how the algorithms have performed for the two different means. The data set that is illustrated in the figure is for the size of 10,000 and a buy order ratio of 50 %. The Cluster 3-1 algorithms have been excluded due to infeasible execution time, those results are present in table 5.10.

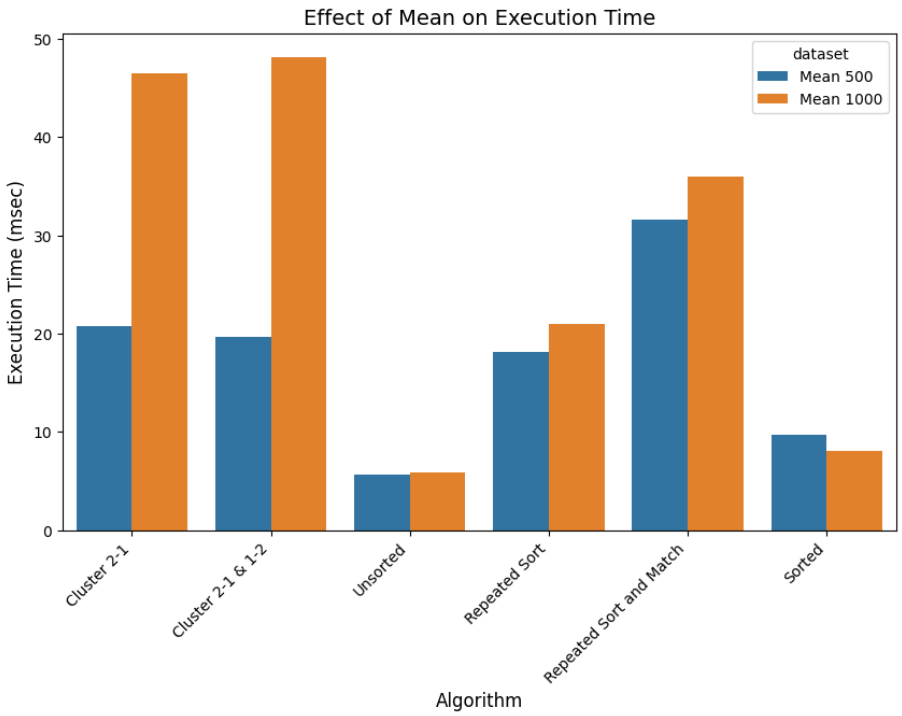

Figure 5.5: Execution time for different means for the data size 10,000 and 50 % buy order ratio

It is evident that the cluster algorithms are highly impacted by changes in the mean. This trend is also especially the case for the Cluster 3-1 algorithms, see table 5.10. The execution time for the Cluster algorithms is twice as high when the mean equals 1,000. One explanation for the increase in time with a larger mean is the reduction in the number of perfect matches found in the beginning due to the broader range of values for the uniform distribution (between 1 and $2 * \mu - 1$). As a result, order sizes are more widely distributed when the mean is larger.

Table 5.10: Algorithm execution times (msec) for different means for the size of 10,000 and buy order ratio = 50 %

Algorithm	**Mean 500**	**Mean 1000**
Cluster 2-1	20.76	46.45
Cluster 3-1	1144.19	6477.07
Cluster 2-1 & 1-2	19.70	48.12
Cluster 3-1 & 1-3	26.77	60.76
Unsorted	5.66	5.91
Repeated Sort	18.14	20.96
Repeated Sort & Match	31.55	36.00
Sorted	9.70	8.07

In the conducted experiments, it was observed that the execution time of all algorithms, except the Sorted algorithm, increases as the mean value increases. It is noteworthy and unexpected that the Sorted algorithm was affected by changes in the mean, and therefore, it is possible that random factors influenced the results. When sorting the order of the lists is changed, which could affect the execution time. Additionally, it is important to mention that the order of the fastest algorithms changes with varying means, indicating that the execution time varies for different means and affects the algorithms differently.

The findings from the analysis of execution time for the generated data sets indicate that there are notable variations among all algorithms for different data sets. It is generally observed that larger and more complex data sets lead to longer execution time, which aligns with the expected theoretical time complexities. An important observation to be highlighted during the discussion is that the order of algorithms and the magnitude of differences between them are not consistent across different data sets. Therefore, it is crucial to consider the specific characteristics of the data sets and execution time requirements to select an appropriate algorithm.

5.4.2 Transactions

To give the full picture of the performance of the generated data sets the transaction performance is presented in the following chapter. The number of transactions is analyzed for the same data sets as for the execution time. The main difference is that the Cluster 3-1 algorithms are included in all figures presented in the following chapter, as those results do not make these graphs unreadable.

The results of the transactions are presented in two ways, the first is the number of transactions produced by each algorithm. The second way the results are presented is the gap between the generated solution and the simple theoretical lower bound. The lower bound does not necessarily mean that it is a feasible solution, but there can never be a lower number of transactions than the lower bound.

The two following figures, figure 5.6 and figure 5.7 are the transaction results for varying data sizes with the mean of 500 and 50 % buy order ratio. In figure 5.6 the y-axis is the number of transactions.

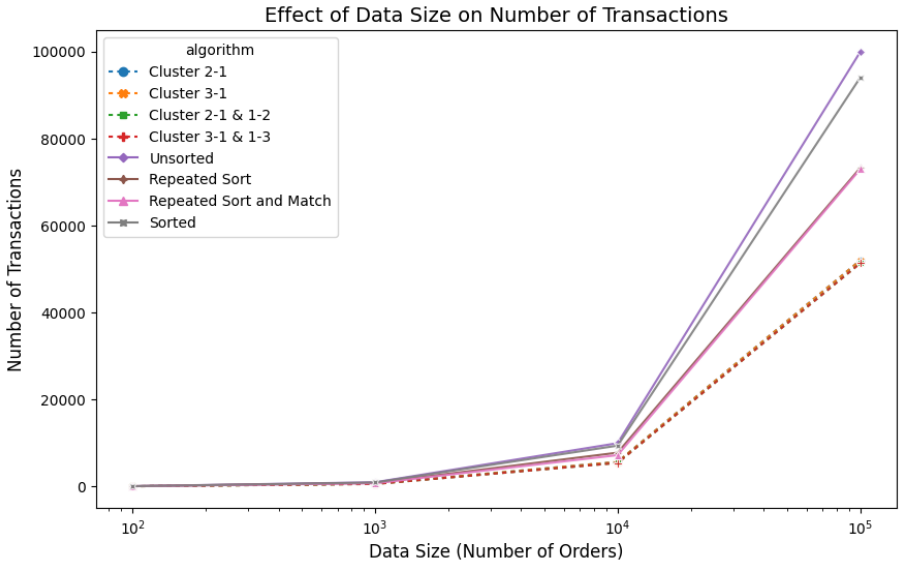

Figure 5.6: Number of transactions for different data sizes, mean = 500, buy order ratio 50 %.

Similar to figure 5.3, it is essential to note that figure 5.7 also has a logarithmic scale on the x-axis, where the size of the data set is presented. However, in contrast, to figure 5.3, the cluster algorithms show the lowest slope incline compared to the simple greedy algorithms. As the lower bound increases for larger data sets, it is reasonable to observe an increase in the number of transactions for larger data sizes. It is difficult to distinguish between the three smaller data sizes. Nonetheless, differences can be observed in the table presented below and in the 5.6 section, whereas more distinct differences can be noticed in figure 5.7.

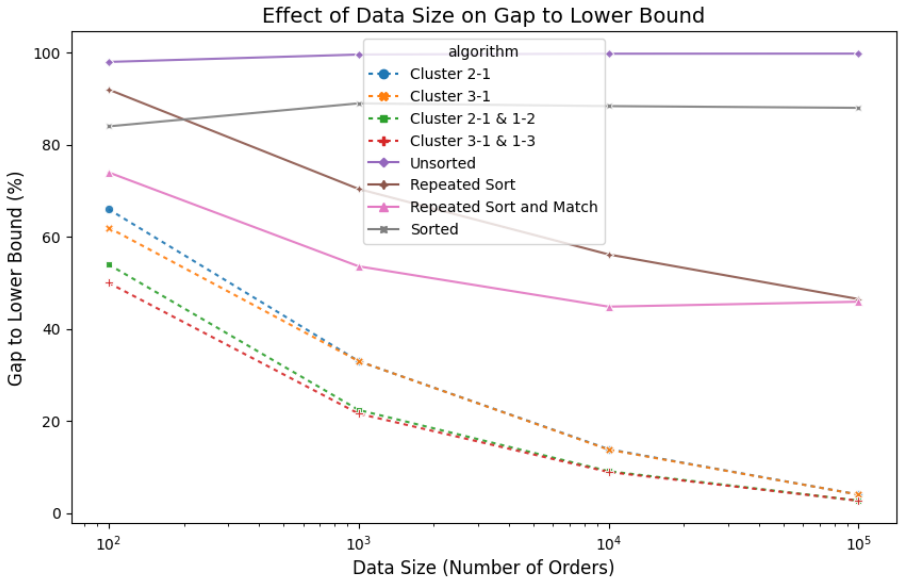

Figure 5.7: Lower bound gap for different data sizes, mean = 500, buy order ratio 50 %.

In figure 5.7 the y-axis is the gap to the lower bound. A small gap means that the obtained solution is close to the theoretically best solution for the given data set. The Unsorted and the Sorted algorithms have the worst performances and have a rather stable performance regarding the gap for all sizes. The other six algorithms have a decreasing gap for larger data sets. All cluster algorithms are reaching similar results with a gap below 10 % for the 100,000 sizes indicating that the algorithms produced solutions less than 10 % from the optimal solution. The Cluster 2-1 and Cluster 3-1 are reaching the same solutions for 1,000 and 10,000 in size, the same can be stated for Cluster 2-1 & 1-2 and the Cluster 3-1 & 1-3 algorithms.

It is clear that the size affects the performance of the algorithms, although the effect varies between the developed algorithms. It has also been shown that the buy order ratio has an effect on the number of transactions which can be seen in figure 5.8.

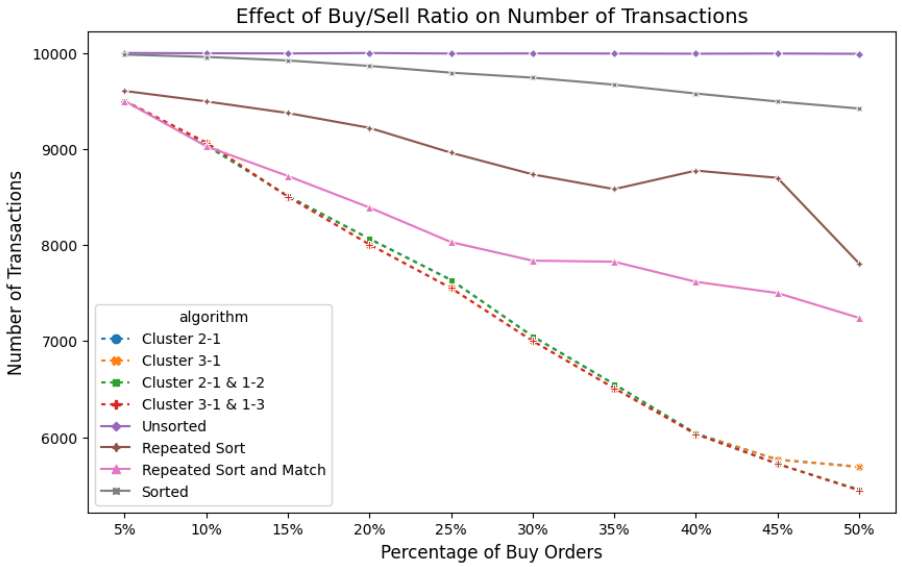

Figure 5.8: Buy order ratio for the size of 10 000 and mean = 500.

Once again the Sorted and Unsorted algorithms stand out with a rather stable result across all ratios. The other six algorithms have a similar amount of transactions when the buy order ratio is low and the number of transactions decreases when the split of buy and sell orders is more equal. It is important to note that even though the number of transactions decreases with an even split of buy and sell orders, it does not mean that the performance is better.

The paradox that decreasing the number of transactions does not have to indicate better performance in terms of gap to lower bound can partially be explained by figure 5.9. The reason is that the lower bound decreases when the length of the sell order list decreases.

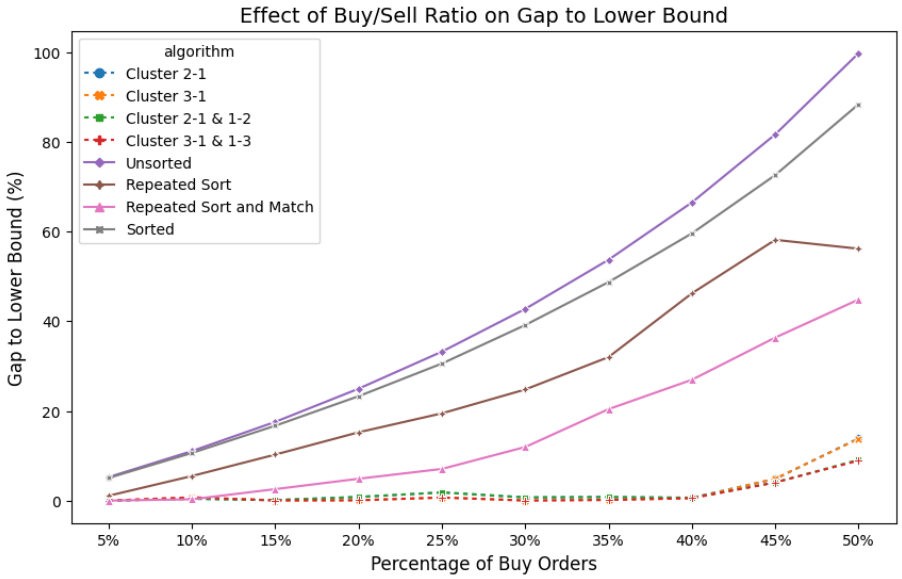

Figure 5.9: Buy order ratio for the size of 10 000 and mean = 500.

Figure 5.9 shows that for all algorithms except for the cluster algorithms, the gap to the lower bound steadily increases when the buy order ratio increases. The cluster algorithms perform very similarly compared to each other and have a small increase in the gap for the different data sets.

The reason why the algorithms are closer in performance when the buy order ratio is low is that the gap between the lower bound and the theoretical higher bound is the tightest then. The lower bound is still the length of the longest list, the upper bound is the length of both lists summed up which is also the same as the size. The potential solutions can therefore only differ by 5 % when the buy order ratio is 5 % but when the buy order ratio is 50 % the difference can be up to 100 % since that is the gap between the upper and lower bound.

The gap to the lower bound is never larger than 100 %, which can be seen in figure 5.10 and figure 5.11. In the two figures, the effect of different means is illustrated for two different means. In figure 5.10 the effect of different means is illustrated with the number of transactions on the y-axis.

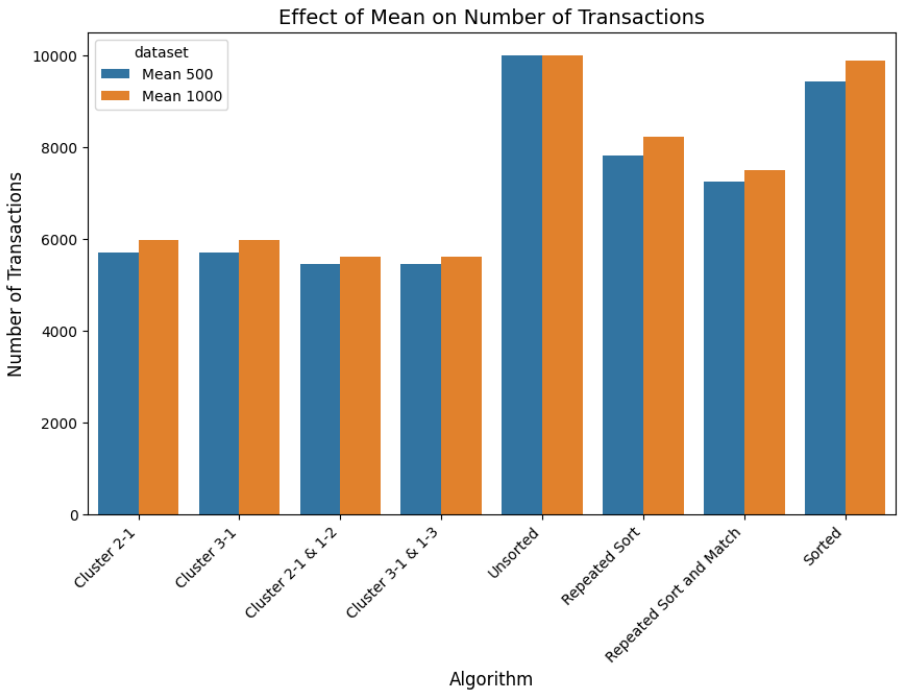

Figure 5.10: Number of transactions for different means for the data size 10,000 and 50 % buy order ratio.

There are two main observations to be made from figure 5.10 and figure 5.11. The first is the effect of the mean on the number of transactions, and the second is the relative performance of the algorithms. The mean of 1,000 appears to result in a slightly higher number of transactions for all algorithms except for Unsorted, which is unaffected. However, the impact of a larger mean on the number of transactions does not seem to be as significant as it was for the execution time aspect in figure 5.5. The same conclusion can be drawn for figure 5.11.

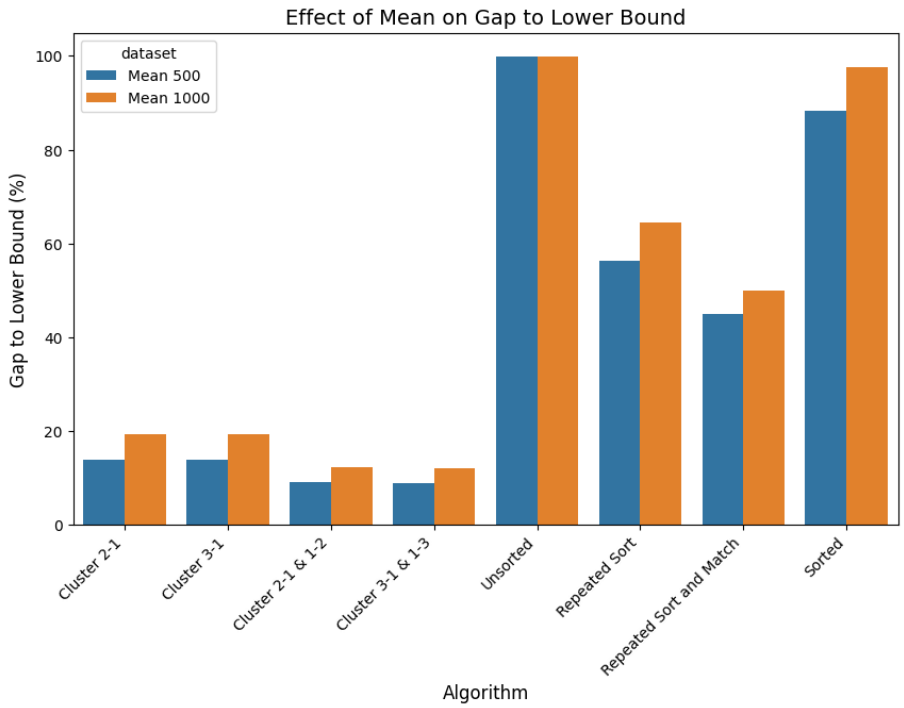

Figure 5.11: Lower bound gap for different means for the data size 10 000 and 50 % buy order ratio.

In this section, the performance of various algorithms for matching buy and sell orders has been evaluated based on execution time and number of transactions produced by the algorithms. The results indicate that the number of transactions produced by the algorithms is influenced by the size and complexity of the data set, as well as the ratio between buy and sell orders. The cluster algorithms, particularly the 3-1 cluster algorithms, are the most sensitive in terms of execution time to the differences in the data size. The mean of the data set also affects the execution time and the number of transactions, although the impact is not as significant as the data set size and ratio.

5.5 Algorithm Performance on Stock Data

In this section, the performance of the eight developed algorithms on trade data from ten different stocks in the OMXS30 is evaluated. Feasible solutions were found for all data sets, although they have not been tested with solvers. It should be noted that the data sizes for the ten stocks vary significantly, and previous results in this chapter have shown that size has a significant impact on both execution time and the number of transactions. To understand the characteristics of the stock data, further analysis is presented in section 5.1, and its implications are discussed in section 6.1.

To evaluate the performance of the developed algorithms on real-world data, the execution time and the number of transactions are presented for all ten stocks included in the OMXS30. This is important to determine whether the algorithms' performance differs from that of the generated data. The execution time of the algorithms on the ten stocks is shown in Table 5.11.
Table 5.11 displays the execution time for all ten stocks. It is observed that some algorithms, such as Unsorted and Sorted, have a rather consistent execution time across all data sets, whereas other algorithms, such as Cluster 3-1, have a higher variation in execution time. This

Table 5.11: Algorithm execution times (msec) for different stocks

Algorithm	ABB	ERICb	INVEb	HEXAb	VOLVb	SBBb	NIBEb	SHBb	EVOG	AZN
Cluster 2-1	1.28	3.02	4.40	1.46	4.48	2.71	21.13	5.44	6.71	111.77
Cluster 3-1	4.48	23.46	25.52	4.19	19.67	7.88	607.02	25.95	21.90	2424.86
Cluster 2-1 & 1-2	1.29	2.87	4.43	1.54	5.38	2.78	22.51	6.93	7.63	113.04
Cluster 3-1 & 1-3	2.02	5.22	26.69	3.53	23.38	4.46	115.93	29.79	17.16	949.60
Unsorted	0.77	1.24	1.06	1.58	2.26	2.68	2.93	2.47	4.97	14.07
Repeated Sort	1.24	2.77	3.40	3.28	6.66	5.58	8.53	7.29	12.39	52.22
Repeated Sort & Match	2.24	4.55	6.67	5.45	11.10	7.68	15.01	12.46	21.36	67.61
Sorted	1.01	1.86	1.41	2.31	2.93	3.31	4.18	3.16	6.36	16.43

variation was also observed in the results from the generated data. Figure 5.12 below presents a graphical representation of the average execution time for the ten stocks.

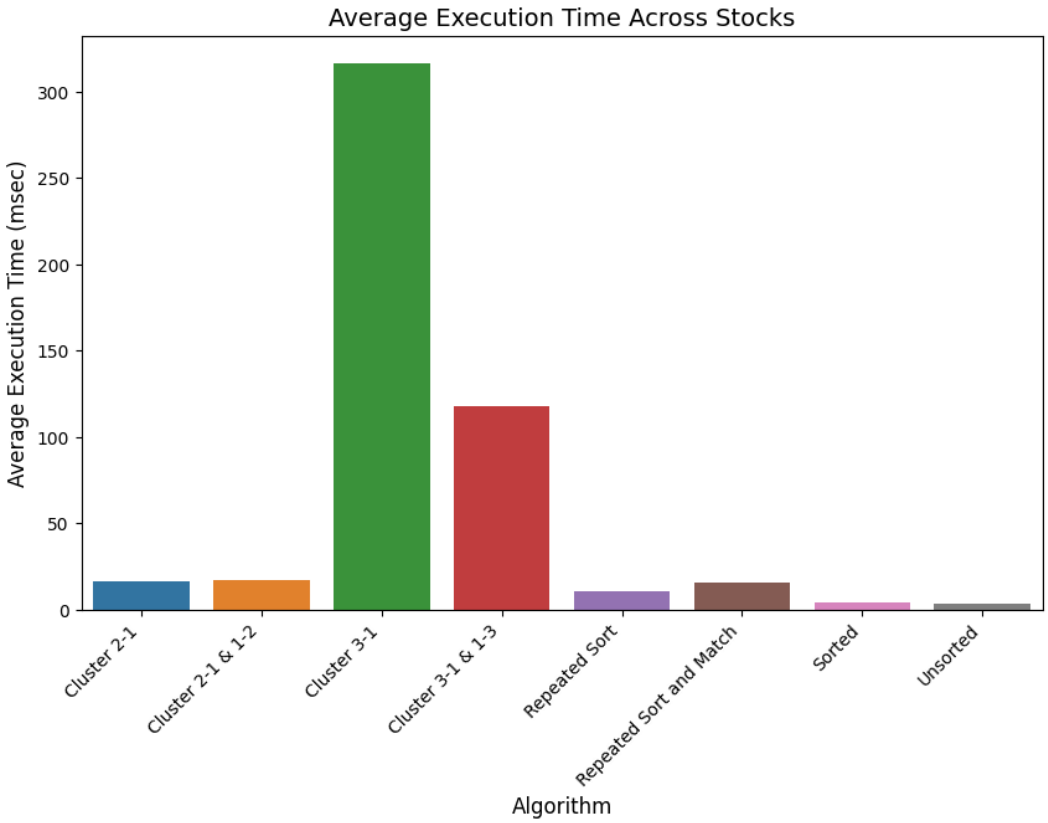

Figure 5.12: Average execution time per algorithm across stocks in milliseconds.

Based on the findings presented in figure 5.12, it is evident that the Cluster 3-1 algorithm has the highest average execution time, followed by Cluster 3-1 & 1-3 and other Cluster algorithms. The relative order of the algorithms varies between different stock data sets, similar to the results obtained from generated data. Interestingly, the Repeated Sort and Match algorithm can be one of the slowest algorithms for some data sets, while it can be one of the fastest for others, which is consistent with the observations made for both stock data and generated data. It is important to note that the variation in execution time is higher for Cluster algorithms compared to Unsorted and Sorted algorithms, a trend also observed in the generated data. Overall, these results provide insights into the relative performance of the developed algorithms on real-world stock data sets.

To evaluate the performance of the algorithms both time and number of transactions need to be evaluated. To be able to evaluate the transactions performance gap to the lower bound is used as a reference since the optimal number of transactions differs between the data sets. The results of the gap to the lower bound are presented in table 5.12.

Table 5.12: Algorithm gap to theoretical lower bound (%) for different stocks

Algorithm	**ABB**	**ERICb**	**INVEb**	**HEXAb**	**VOLVb**	**SBBb**	**NIBEb**	**SHBb**	**EVOG**	**AZN**
Cluster 2-1	8.27	12.80	0.31	1.15	0.28	5.39	17.54	0.20	0.65	9.87
Cluster 3-1	8.38	12.80	0.13	0.91	0.16	5.58	17.54	0.07	0.40	10.24
Cluster 2-1 & 1-2	7.02	11.08	0.31	0.91	0.28	5.17	15.62	0.20	0.63	9.59
Cluster 3-1 & 1-3	6.80	11.00	0.13	0.67	0.16	5.05	15.42	0.07	0.40	9.52
Unsorted	94.11	96.25	42.06	83.07	65.49	97.23	99.35	56.55	84.59	97.63
Repeated Sort	24.01	51.34	25.11	27.61	38.12	18.53	55.42	28.22	27.75	56.02
Repeated Sort & Match	14.84	34.72	16.45	16.63	21.76	14.85	41.85	15.26	23.78	35.29
Sorted	92.30	92.18	40.18	73.24	61.43	81.35	96.07	51.18	72.40	94.35

For five of the ten stocks, the Cluster algorithms find solutions that are less than 1 % which implies that the solutions are less than 1 % from the optimal solution. The Unsorted algorithm has the largest gap for all ten data sets which is reflected in figure 5.13 below where the average gap for the ten data sets is presented.

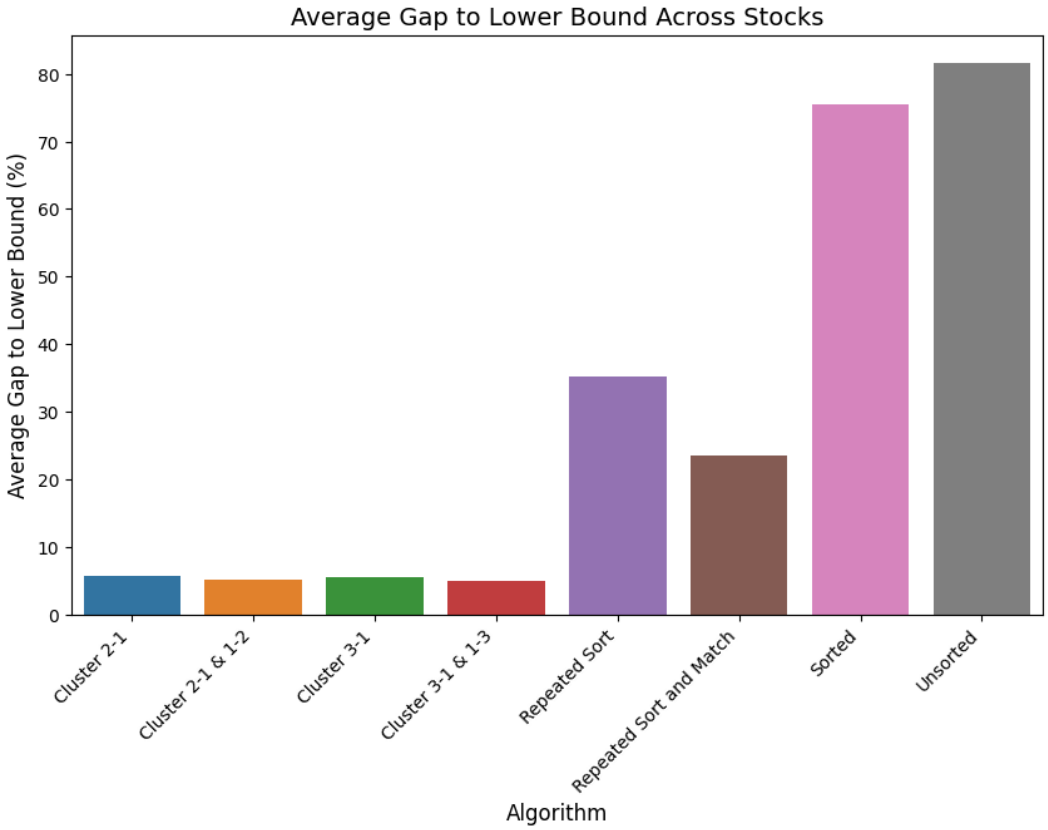

Figure 5.13: Average gap to theoretical lower bound per algorithm across stocks in milliseconds.

The average gap presented in Figure 5.13 indicates that the four Cluster algorithms outperform the other algorithms in terms of accuracy. Among the Cluster algorithms, the Cluster 3-1 & 1-3 has the lowest gap to the lower bound and shows slightly better results than the other three algorithms. The remaining four algorithms demonstrate the same relative order of results as observed for most of the analyst data sets.

The similarity between the results from the stock data and the generated data is evident. However, it is not possible to conclude that the algorithms perform identically for both data types since the relative order of the algorithms varies across different data sets. Nevertheless, the cluster algorithms continue to exhibit the lowest number of transactions for both real and generated data, while the Unsorted and Sorted algorithms remain the fastest for all analysed data sets.

The stability of the algorithm's performance on the stock data suggests that they are comparable to their performance on the generated data. Moreover, the fact that all developed algorithms were able to find feasible solutions in a reasonable amount of time indicates that they can be applied to solve real-world problems. This implies that the characteristics of the generated data are similar to those of real-world data, which supports the validity of the approach used in this study.

5.6 Performance Conclusion

In the following section, a summary of the results will be presented. This will be done by presenting how the algorithms perform in combination with transactions and time. There will also be presented the relative order of the time and transactions for all data set on average.

First, the results will be presented for three selected data sets in the form of a pareto graphs where execution time is on the x-axis and the number of transactions is on the y-axis. The algorithms are scattered on the graph and the pareto frontier is drawn between the pareto optimal solutions. A Pareto optimal solution is one where no other solution can achieve a better outcome in one aspect without making another aspect worse. In the context of low time and low transactions, a Pareto optimal solution would be one where the time is minimized while keeping the number of transactions low.

The three presented data set is a good representation of all data sets that showed similar results. The three different data sets have a mean of 500, 50 % buy order distribution for three different sizes, 100, 1,000, and 10,000. The first presented data set in figure 5.14 is for when the size was equal to 100. To get a better illustration of the pareto frontier all data sets will be plotted twice, the first with all algorithms included and the second time only the pareto optimal algorithms will be included in the graph.

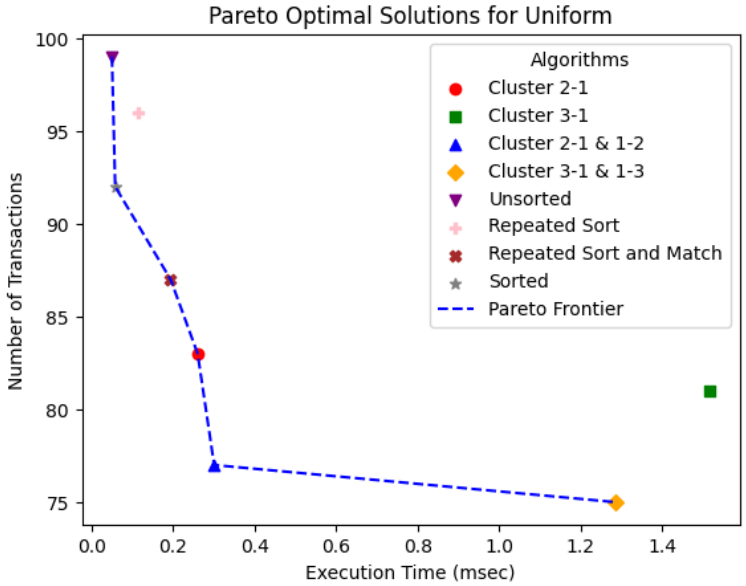

Figure 5.14: Pareto diagram of all developed algorithms for uniform distribution with mean = 500, size = 100, and the buy ratio of 50 %.

In figure 5.14 above, six out of the eight developed algorithms are located on the pareto frontier. The Sorted algorithm is the fastest algorithm but it is also the algorithm with the highest number of transactions. The Cluster 3-1 & 1-3 was the algorithm with the lowest number of transactions, while the slowest algorithm was Cluster 3-1. The two algorithms that were not on the pareto frontier were the Sorted algorithm and the Cluster 3-1. Since none of the algorithms that are not included in the pareto frontier have significantly longer execution time than any other algorithm, figure 5.15 below has almost an identical appearance as figure 5.14.

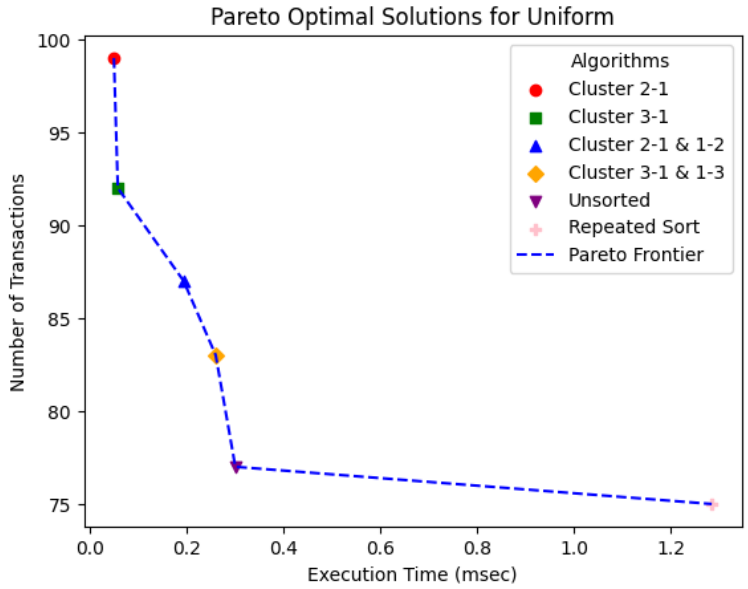

Figure 5.15: Pareto diagram of all developed algorithms for uniform distribution with mean = 500, size = 100, and the buy ratio of 50 %, pareto optimal solutions only.

The visual appearance is slightly different in figure 5.16 although similar patterns can be identified. There are still six algorithms that are included in the pareto frontier. The two algorithms that are excluded this time are the Cluster 2-1 and Cluster 3-1 algorithms. The Cluster 3-1 algorithm has a substantially longer execution time compared to the rest of the algorithms. The Cluster 2-1 algorithm is located close to the pareto frontier but is not considered pareto optimal.
Cluster 3-1 & 1-3 algorithm has the lowest amount of transactions even though the Cluster 2-1 & 1-2 appears to be close in figure 5.18. The pareto frontier in figure 5.19 is more readable when the algorithms that are not pareto optimal are excluded. It is important to note that the pareto frontier in figure 5.18 is the same as in figure 5.19 but the latter gives a more detailed representation of how the pareto optimal solutions for the given data set relate to each other.

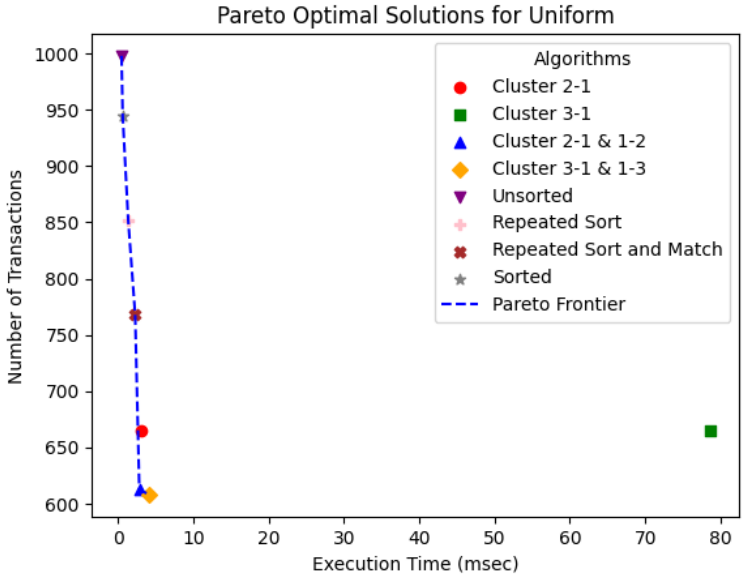

Figure 5.16: Pareto diagram of all developed algorithms for uniform distribution with mean = 500, size = 1,000, and the buy ratio of 50 %.

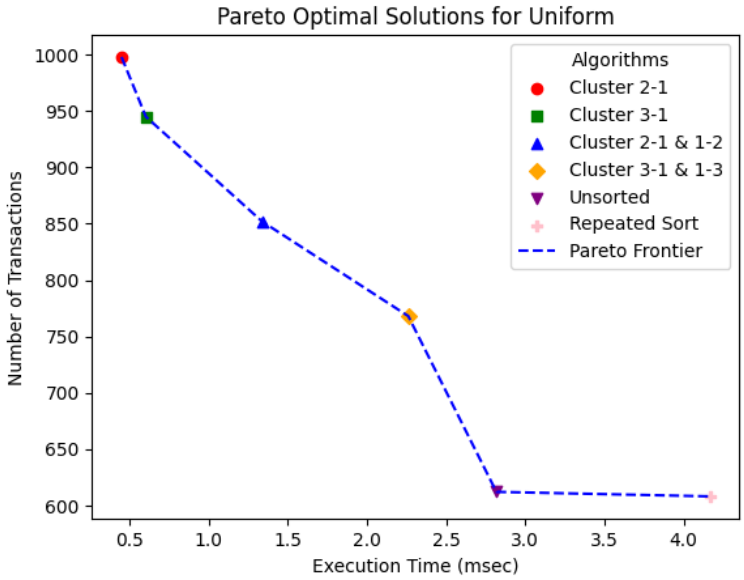

Figure 5.17: Pareto diagram of all developed algorithms for uniform distribution with mean = 500, size = 1,000, and the buy ratio of 50 %, pareto optimal solutions only.

The last presented data set are when the size is equal to 10,000 and thus the biggest of the presented data sets. The time difference gets increasingly larger when the size of the data sets increases. This can be explained by the theoretical time complexity for each algorithm presented in the table 5.7. Since the simplest algorithm has a linear increase and the more complex cluster algorithms time complexity increases with various power functions. An illustration of the pareto frontier and all algorithms scattered in the plot can be found in figure 5.18 below.

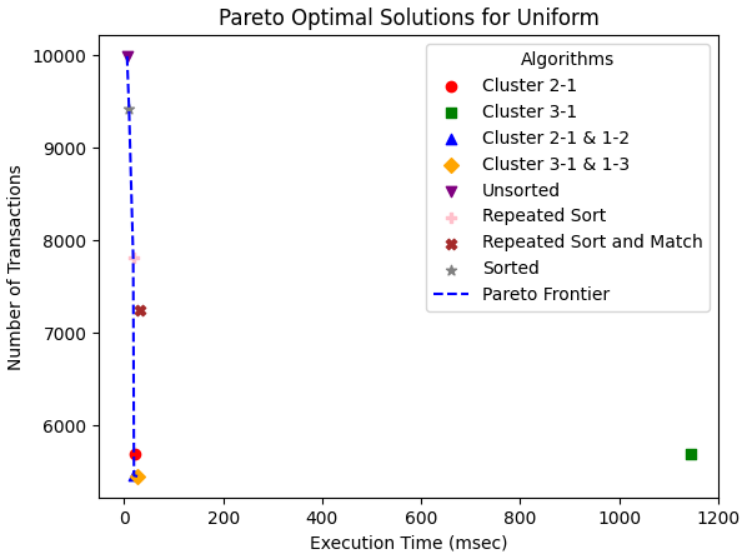

Figure 5.18: Pareto diagram of all developed algorithms for a uniform distribution with mean = 500, size = 10,000 and the buy ratio of 50 %.

It is obvious that the Cluster 3-1 algorithm has a significantly longer execution time compared to the rest of the algorithms. To get a clear picture of which algorithms are included in the pareto frontier figure 5.19 gives a better understanding of the pareto frontier.
Five algorithms are included in the pareto frontier, and the time difference between them is illustrated in Figure 5.19, varying by a factor of five from the slowest to the quickest algorithm in the frontier. It is also noteworthy that Cluster 2-1 & 1-2 and Cluster 3-1 & 1-3 have a similar number of transactions but a distinct time difference. The Cluster 2-1 algorithm was close to being included in the frontier but had a worse solution compared to the closets algorithm Cluster 2-1 & 1-2.

Analysis of the three data sets and their pareto graphs reveals that the algorithms on the pareto graphs differ, and the differences between the data sets become more pronounced as the size of the data sets increases, which has been observed in previous sections.

The pareto graphs do not recommend a particular algorithm but rather indicate that the algorithms not included in the frontier are not optimal. It is even more difficult to exclude any algorithms since it differs from the data sets which algorithms that are included in the frontier. The selection of the best algorithm depends on the user's requirements in terms of execution time and the number of transactions, and which parameter is given more importance.

In order to evaluate and summarize the performance of the algorithms across the parameters that makes a difference in performance, figure 5.20 below portrays the average execution time across all sizes and all buy/sell order ratios. In the following figures the Cluster 3-1

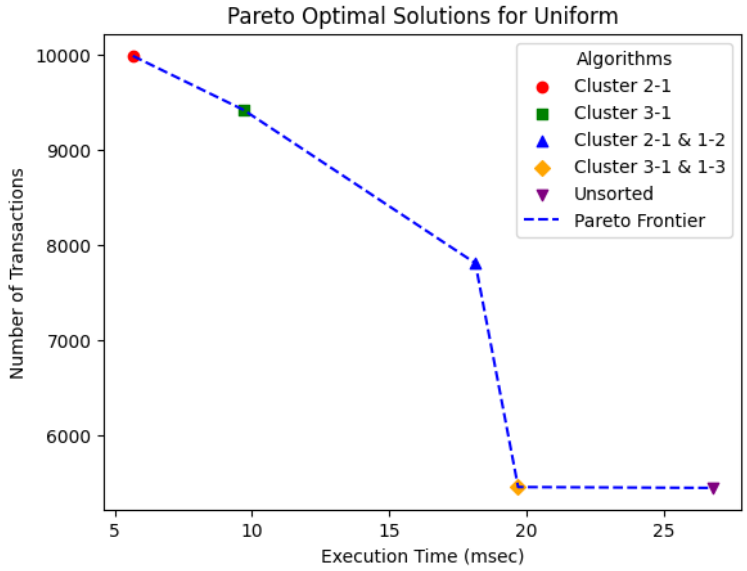

Figure 5.19: Pareto diagram of all developed algorithms for a uniform distribution with mean = 500, size = 10,000 and the buy ratio of 50 %, pareto optimal solutions only.

algorithms are excluded since they could not find a feasible solution in a reasonable amount of time for all data sets.

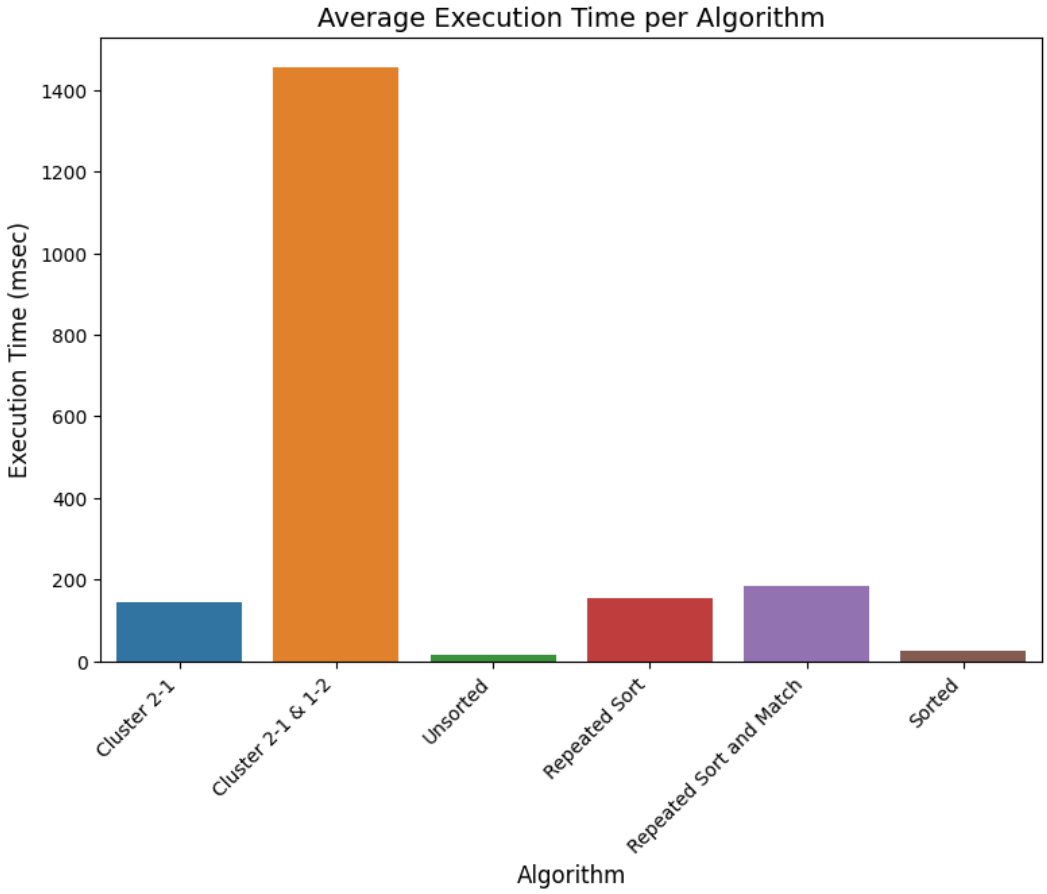

Figure 5.20: Average execution time per algorithm in milliseconds. Mean = 500, all sizes and ratios.

Looking at the average execution time it is clear that the Cluster 2-1 & 1-2 algorithm is significantly slower on average than the rest of the included algorithms. The difference can be contributed to the fact that Cluster 2-1 & 1-2 performs worse when one list is significantly longer than the other. Cluster 2-1, Repeated Sort, and Repeated Sort and Match are close to equal in average execution time, while Unsorted and Sorted are the fastest.

Figure 5.21 below portrays the average performance in terms of how the solutions compare to the lower bound.

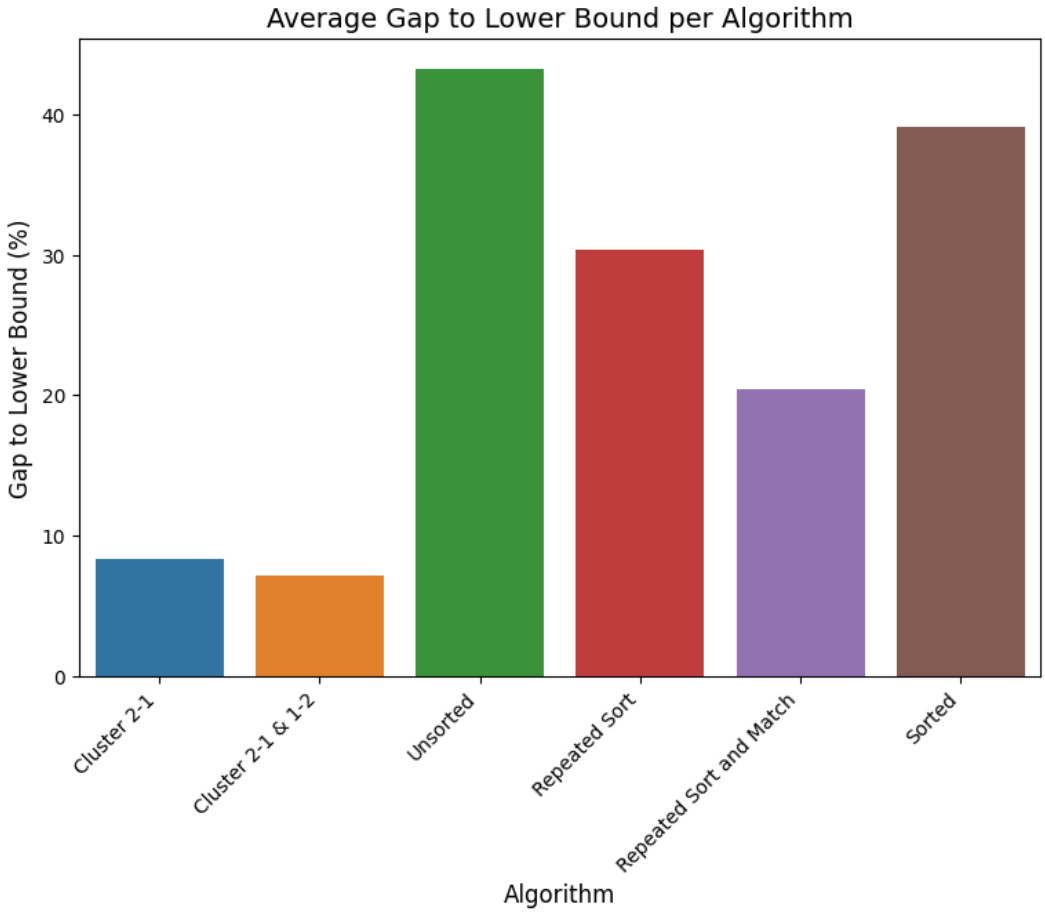

Figure 5.21: Average gap to lower bound per algorithm in percent. Mean = 500, all sizes and ratios.

It is clear that Cluster 2-1 & 1-2 on average produce the best solutions, followed closely by Cluster 2-1. While Cluster 2-1 & 1-2 on average produces better solutions, it is much slower in some cases, which results in a slow average execution time. In this figure the Cluster 3-1 algorithms are excluded since they could not find a feasible solution in a reasonable amount of time for all data sets and therefore have an execution time longer than 600,000 msec for the largest data sets.

5.7 Algorithm Validation

This section presents the results of the t-tests conducted on each algorithm and data set, along with an analysis of the consistency of each algorithm for each execution. Additionally, the verification tests performed to ensure that the solutions obtained are valid are analyzed.

The verification of all obtained solutions revealed that none of the solutions violated any of the constraints presented in equation 4.1. It should be noted, however, that the results generated by the GUROBI-solver and Cluster 3-1 were excluded due to their inability to find a feasible solution within the specified time constraints for all data sets. Notably, for all data sets where the algorithms were allotted sufficient time to complete, no infeasible solutions were generated.

Since all algorithms were tested on each data set ten times the number of data points generated from the tests combined was large but the selected data sets will be illustrated below. Two cases of data will be illustrated that represent the data and the results from the tests. The two selected data sets that are illustrated below both have a mean 500 and a size of 10,000. The difference is that in 5.22 the buy order ratio is 50 % and in 5.23 the buy order ratio is 5 %. In 5.22 all runs for a single data set are illustrated, execution time and number of transactions for all developed algorithms are illustrated separately.

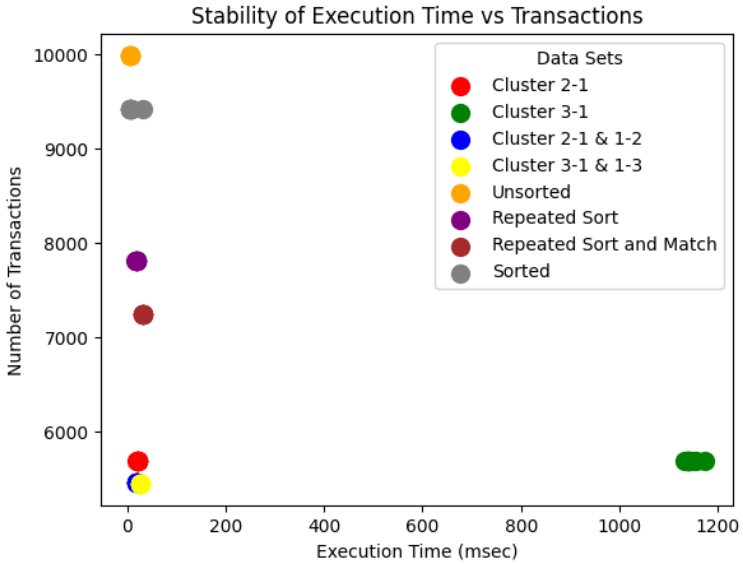

Figure 5.22: Stability test of a uniform distribution with mean = 500, size = 10,000 and the buy ratio of 50 %. Ten tests was conducted on the data set for each algorithm and all results are presented in the graph.

In figure 5.22, it is evident that seven of the algorithms exhibit similar execution times, while one algorithm, namely the Cluster 3-1 & 1-3 algorithm, stands out due to its considerably longer execution time. Notably, the Cluster 3-1 & 1-3 algorithm also displays the greatest variation in execution time across the ten runs, with the Sorted algorithm exhibiting the second-largest variation. All algorithms were tested with the same number of transactions across all runs and data sets. To get a better representation of how the algorithms relate to each other when the Cluster 3-1 algorithms are excluded additional graphs are available in the appendix section A.1.

The results depicted in figure 5.22 are representative of the overall results obtained across all data sets. Although the results for different means and sizes displayed a similar trend, the difference in both time spread and performance variation was more pronounced for larger sizes and means compared to smaller ones. Although, when the buy order ratio changed, the results were slightly different. This is exemplified in figure 5.23, where the buy order ratio is 5% as compared to 50% in figure 5.22.

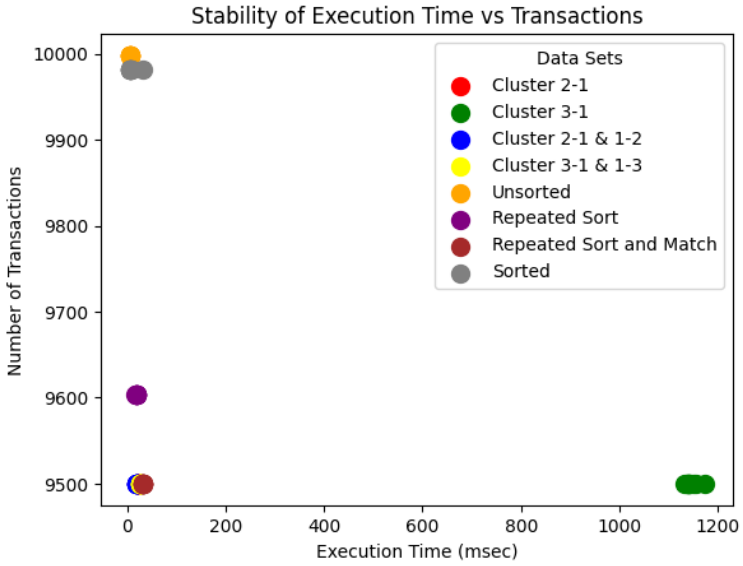

Figure 5.23: Stability test of a uniform distribution with mean = 500, size = 10,000 and the buy ratio of 5 %. Ten test was conducted on the data set for each algorithm and all results are presented in the graph.

As illustrated in the above figure, the Cluster 3-1 & 1-3 algorithm has the longest execution time, followed by the Cluster 2-1 & 1-2 algorithm. Notably, all algorithms except for the Cluster 3-1 & 1-3 algorithm exhibit a reduced time spread. It is noteworthy that the execution times remained relatively stable throughout the testing phase, with no algorithm exhibiting a time spread greater than 10 % across any data set.

A t-test was performed for all algorithms and data sets to determine if one algorithm significantly outperformed the others in terms of execution time. The results of the t-test are summarized in table 5.13 for each algorithm and data set combination. A "Yes" in a given cell indicates that the algorithm corresponding to the row significantly differs in time to the algorithm named in the corresponding column for all data sets. A "No" indicates that the results between two algorithms are not significantly different.

Table 5.13: Significance test using a t-test at 5% confidence level with a uniform distribution, mean equals to 1,000 and 5% buy order ratio

Algorithm Name	Significant, Yes or No							
	Cluster 2-1, V1	Cluster 2-1, V2	Cluster 3-1, V3	Cluster 3-1, V4	Unsorted	Sort and Match	Repeated Sort and Match	Sorted
Cluster 2-1, V1	-	-	-	-	-	-	-	-
Cluster 2-1, V2	Yes	-	-	-	-	-	-	-
Cluster 3-1, V3	Yes	Yes	-	-	-	-	-	-
Cluster 3-1, V4	Yes	Yes	Yes	-	-	-	-	-
Unsorted	Yes	Yes	Yes	Yes	-	-	-	-
Sort and Match	Yes	Yes	Yes	Yes	Yes	-	-	-
Repeated Sort and Match	Yes	Yes	Yes	Yes	Yes	No	-	-
Sorted	Yes	Yes	Yes	Yes	No	Yes	Yes	-

The tests performed to determine if there were any significant differences in execution time between the developed algorithms revealed that only two pairs of algorithms which did not exhibit significant differences. The first pair comprised the Sort and Match and Repeated Sort and Match algorithms, while the second pair consisted of the Sorted and Unsorted algorithms. It is worth noting that the four algorithms involved in the pairs are the simplest of the developed algorithms and only differ in terms of sorting and matching processes. In contrast, the remaining algorithms exhibited significant differences in execution time, indicating that it is possible to conclusively determine the relative performance of the algorithms

with regard to execution time.

In conclusion, with regard to the validation of the evaluated algorithms, it is crucial to note that among the algorithms that were able to complete their runs, all generated solutions were feasible. Additionally, it is noteworthy that the time spread for a single algorithm across a specific data set is generally low, indicating that the execution time is relatively stable for a given data set for all algorithms.

6 Discussion

This chapter presents the discussions and conclusions based on the results and analysis of the study. Firstly, a discussion on the data used for testing the algorithms and the implications of the choice of data is presented. Secondly, the performance of all the developed heuristics and the optimization solvers are discussed. Additionally, a method discussion is included to provide insights into the certain path of the method. Ethical aspects of the study are also discussed, followed by suggestions for further research. The chapter concludes with the overall conclusions and recommendations of the study.

6.1 Data Evaluation

The data analysis conducted in this study focused on ten stocks included in the OMXS30 index. The obtained data was trade data rather than order data from the buy and sell side, which is necessary to run tests of the algorithms. The available volume from the trade data was used to provide insights into the stocks' trading patterns.

The first phase of data analysis aimed at investigating the average traded volume and the average stock price for each stock. In table 5.2, the ten analyzed stocks are presented along with their respective average volume and price. Significant differences were found in the average traded volume among the stocks. A negative correlation was observed between the stock price and the average traded volume. This correlation can be attributed to the total value of the trade, which is calculated by multiplying the stock price by the traded volume. The total traded value plays a crucial role in explaining the average traded volume.

The subsequent phase of data analysis aimed to determine the most frequently occurring volume sizes for each stock and whether any common volumes existed across the ten stocks. Table 5.3 presents the ten most common volume sizes for each stock. Although all trades were analyzed to identify the most common volume sizes, the top ten for each stock were presented as a representative sample.

The next phase of data analysis investigated how common it is for one volume to occur more than once. It was found that multiple trades with the same volume size occurred frequently. This observation could be attributed to traders using the same trading algorithm,

which could lead to similar trading patterns.

In the conclusive phase of the data analysis, the investigation centered around the final digits of the traded volume. Despite a thorough examination, no significant pattern was discernible. However, it was noticeable that a higher frequency of trading volumes ended with a zero in comparison to the other digits. In addition, trades that concluded with two or three zeros were also scrutinized, and in some cases, a more frequent occurrence was evident. Nevertheless, it was challenging to ascertain any particular pattern of commonly traded volumes.

The hypothesis under investigation posits that the volume size is contingent upon the total value of the trade. Our initial findings were inconclusive, although some indications suggest that the hypothesis could be true. However, we contend that two major factors have been overlooked that impact the hypothesis, namely, the sell-side of the trade's desire to sell the full position in the instrument, and the fact that the analyzed data is trade data and not order data, resulting in a different order size from what the trader intended from the beginning.

The nature of the analyzed data as trade data and not order data presents a significant challenge in data analysis, as it reflects only a portion of reality and not the entire picture. We have evaluated data solely based on ten stocks included in the OMXS30 index traded on the Stockholm exchange. Furthermore, the analyzed data is only for continuous trading, while the purpose of the report is to match trades in call auction pricing after the price has been set. Hence, we suggest that actual order data would significantly improve the analysis. Additionally, a broader investigation of other instruments and stocks listed on other exchanges would provide more comprehensive insights into how the order volume differs between different instruments and exchanges.

Given the uncertainties surrounding the data analysis and the absence of real-life call auction order data, we generated various types of data sets to test the algorithms on both small and large data sets and other variable factors. This approach is believed to lend depth and credibility to the performance analysis of the algorithms. To further strengthen the data analysis trades over a longer set of time could be investigated and further examine the correlation between the traded price and volume.

Our findings suggest that the volume size is not the critical factor in trading, but rather a consequence of the trader's desire to trade for a specific value. However, given the limitations of the data analyzed, further research is necessary to confirm these findings. This includes investigating a more extensive range of instruments, orders, and exchanges, as well as exploring the differences in order volumes between continuous trading and call auctions.

In summary, the data analysis conducted on the ten stocks included in the OMXS30 index provided insights into the stocks' trading patterns. The average traded volume, the most frequently occurring volume sizes, and the last digits of the traded volume were analyzed. The analysis revealed significant differences in the average traded volume among the stocks, a negative correlation between the stock price and the average traded volume, and frequent occurrences of multiple trades with the same volume size. These findings can be useful in developing and optimizing trading algorithms for similar instruments.

6.2 Algorithm Evaluation

In the following chapter, the performance of the developed algorithm as well as the commercial solver will be discussed. The section will discuss the results and how it changes over the tested data sets lifting potential reasons for why the algorithms have performed in

the way they did. There will also be a part regarding the optimal solution and how close to the optimal solution the developed algorithm could be as well as a discussion regarding the validation and time aspects.

One of the intriguing outcomes of the experimentation is the underperformance of the GUROBI-solver, and the inability of the SCIP-solver to discover feasible solutions in a reasonable amount of time. The GUROBI-solver only managed to close the gap between the lower bound and the obtained solution twice. Once the gap is closed, it is guaranteed that the optimal solution has been attained. Consequently, the algorithms can only be assessed against a proven optimal solution in two scenarios. A possible reason for the underperformance of the solvers, could be explained by the fact that number of possible partitions become exponentially large for bigger integers.

To provide a reference point for other data sets, the outcomes have been benchmarked against a theoretical lower bound. The length of the longest list of the buy and sell order that was supposed to be matched served as the lower bound, although it may not be the best choice to evaluate against, as it could be far from the optimal solution. However, the lower bound was chosen since it could be easily found for all the analyzed data sets. Using the GUROBI-solver's lower bound would have resulted in varying lower bound accuracies across data sets, as GUROBI works quicker on smaller data sets and it is easier to find the lower bound when the buy order ratio is low compared to when it is close to 50 %. All Cluster algorithms and Repeated Sort and Match have found the lower bound for various data sets. Although the data only set when the optimal solution has been found is when the buy order ratio is 5 %.

The analysis of the data sets revealed variations in the number of transactions generated by the algorithms. Specifically, for all the data sets examined, the four cluster algorithms consistently exhibited the lowest number of transactions. Conversely, the Sorted and Unsorted algorithms produced inferior results across all the data sets. Despite the Unsorted algorithm outperforming the Sorted algorithm in some cases, the former can be regarded as a random solution. Notably, although the Unsorted algorithm was found to be the fastest in most cases, it was not statistically significantly faster than the Sorted algorithm across all the data sets.

The cluster algorithms did exhibit the lowest number of transactions but were also the algorithms with the longest execution time in certain scenarios. The Cluster 3-1 algorithms could not finish the tests within the 10 minute time limit for the largest data set in the cases where the ratio between buy and sell orders was large. On some occasions the Cluster 3-1 algorithm needed more time compared to the Cluster 3-1 & 1-3 algorithm. That is explained by the fact that the 2-1 process is much quicker than the 3-1 process. The double-sided 3-1 algorithm benefits from the fact that the double-sided 2-1 process has been carried out before the 3-1 process starts and fewer 3-1 matches need to be identified.

The effect ratio has on the algorithms can be explained using the time complexity analysis. The greedy algorithms are primarily affected by the total number of orders, while the cluster algorithms matching functions are heavily dependent on the number of indexes in the dictionaries. For example, if we compare the case of 500 buy orders and 9,500 sell orders with the case of 5,000 buy orders and 5,000 sell orders, the matching process of the greedy algorithms will go through 10,000 iterations no matter the ratio of buy and sell orders, which explains why the execution time is constant for all greedy algorithms in figure 5.4. The functions for finding cluster matches in the cluster algorithms are instead dependent on the number of indexes in each dictionary. In the case of 500 buy and 9,500 sell orders, in the worst case where each volume is unique there will be 9,500 indexes. This explains figure 5.4 and why the ratio has an apparent effect on the execution time for the cluster algorithms.

The Cluster 2-1 algorithm are displaying stable results both regarding time and accuracy compared to the other cluster algorithms that expose a greater variety in time and has a significantly larger execution time compared to the greedy algorithms. This is true especially when the buy order ratio was low, the low buy order ratio generated longer execution time for the doubled-sided cluster algorithms. The time required for the developed algorithms can be explained by their respective theoretical time complexity.

A larger mean appeared to have a significant effect on the algorithms that has a perfect match mechanism. Since a larger mean meant that the same number of orders were distributed over a larger distribution resulting in fewer orders with the same volume. The same can be said for when the buy order ratio is low then the mean differs greatly between the buy order and sell order lists resulting in fewer orders with the same volume. The findings are that the cluster algorithms and Repeated Sort and Match perform poorer when there exist few perfect matches in the data sets.

In section 3.7, it was proved that a perfect match never deteriorates a solution. The solutions obtained from the algorithm that incorporates perfect matching mechanisms were consistently more accurate than those obtained from the three algorithms without such mechanisms. These results support the validity of the proof and suggest that removing all perfect matches is an advantageous strategy for addressing the problem.

We also propose that besides finding perfect matches, the case could be made that performing 2-1 matches, and subsequently 3-1 matches, and so forth, after finding perfect matches should not deteriorate a solution either. The argument could be made that performing a 2-1 match once there are no perfect matches left, is an optimal decision. Performing a perfect match results in two orders being resolved through one transaction, while a 2-1 match results in three orders being resolved through two transactions, and a 3-1 match results in four orders being matched through three transactions. Thus, a perfect match matches two orders in one transaction, a 2-1 match matches three orders in two transactions which can be rewritten to $3/2$ orders per transaction, and a 3-1 match matches $4/3$ orders in one transaction. Generating transactions that resolve as many orders as possible should result in more orders being resolved with fewer transactions.

In the pareto graphs presented, various algorithms are included in the frontier for the analyzed data sets. However, the fact that the frontier looked different for different data sets implies that drawing conclusive remarks from the presented graphs is challenging. Notably, the Cluster 3-1 algorithm is the only algorithm that is not included in any pareto frontier. Therefore, it is unlikely to be an algorithm that would offer a viable solution for the order matching problem.

The results from the stock trade data appear to have similar results as the generated data. There is however hard to draw any strict conclusions regarding the results and the relative order for the algorithm for both time and accuracy. But the algorithms that were quick for the generated data were also quick for the stock data and the same for the accuracy.

The verification of the solutions obtained by each algorithm revealed that all solutions generated were feasible. The ability to generate only feasible solutions does not give any specific advantage to any algorithm but is an important aspect to consider if an algorithm is to be implemented for commercial purposes. Another important observation is that all algorithms demonstrated equivalent stability, as each algorithm generated the same number of transactions across ten runs for each data set, despite slight variations in time.

It was found in the theory chapter 3 that the problem of matching orders by volume in order to generate as few transactions as possible could be formulated as the 2-MCIP found in previous research. The studies regarding 2-MCIP took a more theoretical computational approach of discussing how close an approximation could be achieved through algorithms. The latest paper on the 2-MCIP problem developed an algorithm, which found 6/5 approximations of the optimum, with a time complexity of $\mathcal{O}((m+n)^{186})$. In this study, we did not make a mathematical argument for the approximation but found in our test cases that the cluster algorithms consistently generated solutions less than 20 % from the optimum. Of course, this is not the same as a mathematically proven 6/5 approximation, but it does give context to the quality of a 20 % gap to the lower bound. Given that the cluster algorithms are also significantly faster in terms of time complexity than the algorithms proposed in the 2-MCIP studies, we find that the cluster algorithms developed in this study strike a good balance between solution accuracy and execution time.

6.3 Method Discussion

The method used in this study had several strengths that made it effective in addressing the research question. One of the strengths was the careful selection of algorithms based on their proven performance in similar optimization problems. However, there were also some weaknesses of the method that need to be considered. For example, the effectiveness of the method may be limited by the size of the sample used in the study. A limited number of data sets were used, which could have affected the performance analysis compared to a larger and more diverse set of data. Additionally, the method may not be able to control for all variables that may have influenced the study's findings, which could affect the accuracy and reliability of the results.

The choice of a greedy heuristic for the algorithms has its strengths and weaknesses. The main strength of this approach is its simplicity and efficiency. However, a potential weakness of the greedy approach is that it may not always find the optimal solution, as it may overlook potential trade-offs that could lead to better solutions. Several other heuristic methods could have been considered as an alternative to the greedy algorithm, such as column generation, local search algorithms, and simulated annealing, as they have been shown to perform well in a variety of optimization problems. However, the greedy algorithm was chosen due to its simplicity and efficiency in addressing the specific order matching problem.

It is also important to note that the evaluation of all possible algorithms was limited by the time constraints of the study. The selection of the greedy algorithm and other heuristic methods was based on a thorough review of the literature and the time available for testing and analysis. While it is possible that other algorithms could have yielded better results, the method used in this study provided valuable insights and generated meaningful findings. Future research could explore the use of other algorithms and compare their performance with the method used in this study.

The limitations of this study should be acknowledged and discussed to provide clarity and context to the results and conclusions. One significant limitation of this study is that it focuses solely on the pairing of buy and sell orders in a call auction trading mechanism after the price has been established. Therefore, it does not consider how the price is determined, the price spread, or other pricing factors that could impact the suitability of the algorithms used in this study. As a result, the applicability of the developed algorithms may be limited in practice, as the algorithms' performance may be affected by other pricing factors not considered in this study.

Furthermore, the developed algorithms are evaluated quantitatively based on execution time and solution accuracy only. Other quantitative factors, such as algorithm robustness and space complexity, may also affect the suitability of the algorithms for implementation. However, these factors are not examined in this report, and thus, the conclusions drawn may not fully reflect the performance of the algorithms in practice. The main focus of this report is on execution time and solution accuracy, as illustrated in figure 1.1.

Another limitation of this study is the use of data generated using predefined distributions, as order data is not publicly available. This approach may not fully capture the complexity of real-world trading environments, and the performance of the algorithms may differ when applied to actual trade data. Additionally, all tested algorithms are evaluated on the same hardware with identical system settings. The results may not be generalizable to other hardware or system configurations, and further testing may be necessary to confirm the algorithms' performance under different conditions.

The use of a single laptop for execution probably has influenced the execution time. If more powerful hardware or servers were used, the execution time could have been drastically lower, which would have affected the results. Future studies could explore the use of more powerful hardware to further optimize the performance of the algorithm. Overall, while the method used had strengths in generating accurate and reliable results, the limitations and weaknesses must also be considered when interpreting the findings.

It should be noted that this study did not specify for which traded elements or on what exchange the algorithm should be optimal. As such, the developed algorithms were designed to be generalizable and applicable to a wide range of trading scenarios. This decision was made to ensure that the algorithms would be useful for a broad audience and to avoid limiting their potential applications. However, it should be acknowledged that different trading scenarios may have unique characteristics and requirements, and as such, the effectiveness of the developed algorithms may vary depending on the specific context in which they are used. Nonetheless, the generalization of the method is a strength as it makes the developed algorithms more versatile and widely applicable.

This study tested only two optimization solvers, SCIP and Gurobi. And even though the research suggested that Gurobi was among the most capable on the market, it is possible that other solvers may have performed better for this problem. Although, since other competitive solvers, such as CPLEX, employ similar algorithms and heuristics as Gurobi, it is doubtful that they would perform better. Furthermore, regarding optimization models, it's important to take into consideration that optimization problems are significantly affected by their implementation. There may exist alternate model definitions for this problem that would have made it easier for an optimization solver to generate better solutions faster. It was not deemed within the scope of this study to look further into alternate models and techniques, but it is something that could be considered in further research.

6.4 Ethical Aspects

In conducting this research, it is important to consider the ethical implications of the results and their potential impact on the market. The optimization problem addressed in this thesis is of great importance to the financial sector and could have significant consequences if implemented in real-life trading. Therefore, it is necessary to ensure that any solution proposed is ethically sound and does not violate any regulatory frameworks.

One important aspect to consider is market manipulation. The proposed solution may

provide an opportunity for traders to manipulate the market by placing large orders in a strategic manner to affect the algorithm's output. To address this concern, measures could be put in place to monitor and detect any suspicious activity and prevent market abuse.

Additionally, it is important to consider the impact of the proposed solution on market transparency. The optimization problem could result in the matching of orders that were not previously visible to the market, potentially affecting the market's overall transparency. Therefore, it is crucial to carefully consider the impact of any proposed solution on market transparency and ensure that it does not have any negative effects on the market's efficiency and fairness.

6.5 Further Research

When contemplating the results of this thesis there are some areas we believe it would be interesting to dig further into. There exist multiple ways to solve this optimization problem and this report only reports a few of the potential methods that could be applied. It would therefore be interesting to investigate other heuristic methods. One, in particular, would be to develop a column-generating algorithm that generates columns with a transaction list and improves the transaction list iteratively until no more improvements are made.

Other research that would be interesting to dig further into is why the optimization struggles to find an optimal solution and if there exists a way of formulating the problem in a way that makes the problem easier to solve. There would be interesting to further investigate if a solver would be more efficient if the parameter settings were optimized to fit the studied optimization problem.

Finally, future research could also explore the use of the method in other applications and fields beyond order matching. For example, the method could be used in logistics optimization, computational biology, or scheduling problems, where there is a need to identify the optimal solution among a large set of possibilities. By applying the method in different contexts, it may be possible to identify additional strengths and weaknesses of the approach and further refine its application.

6.6 Conclusion

The purpose of this study was to address an optimization problem involving the matching of buy and sell orders in a call auction to minimize the number of transactions. To accomplish this, various algorithms were developed, evaluated, and tested to determine which approach offers an appropriate balance between execution time and solution quality.

The results section of the study presented findings for eight developed algorithms and one optimization solver. The complexity of the problem exceeded initial expectations, and the commercial solver could only find solutions for small and simple data sets. This limitation led to the exclusion of the solver as a potential solution to the problem. Consequently, the eight developed algorithms were assessed based on execution time and solution accuracy, as outlined in the delimitation section 1.2.

A challenge arose in the study from the lack of specification regarding data characteristics of real instrument order volumes, as pre-trade data of call auctions is not publicly available. The results exhibited considerable variation across data with differing characteristics, highlighting that the structure of data is a critical factor when choosing an appropriate algorithm. Unfortunately, we were unable to obtain actual order data for any instrument,

limiting the certainty of our results. Even though it is uncertain how the algorithms would perform in a production scenario, the test data that was used enabled an analysis of how a broad range of factors affected the performance of the algorithms.

The developed algorithms effectively found feasible solutions for the given problem. However, the results were not entirely conclusive and presented some challenges in interpretation, as they were influenced by factors such as the data structure and the specific requirements of the implementing party.

The difficulty in determining the best algorithm stems from the trade-off between execution time and solution accuracy. The importance of these factors must be weighed before deciding on the algorithm to be implemented. If execution time is solely prioritized over solution accuracy, the Unsorted algorithm should be implemented. Conversely, if solution accuracy is the only important factor, the Cluster 3-1 & 1-3 should be implemented, despite its inability to find solutions in a reasonable time for all test cases.

Ideally, a solution that performs well in both execution time and accuracy should be identified to fulfill the study's purpose. A thorough analysis of the results for all algorithms and data sets revealed that the cluster algorithms had the fewest transactions, with only slight variations between the different cluster algorithm variations. Among the cluster algorithms, the Cluster 2-1 algorithm typically exhibited the lowest execution time and remained stable even when the buy order ratio was low compared to double-sided cluster algorithms.

In conclusion, while no single algorithm can be definitively labeled as the best, Cluster 2-1 strikes the most effective balance between execution time and solution accuracy, while also remaining relatively stable in performance for all test cases. The study's purpose has been achieved by developing eight algorithms that solve the given problem and suggesting the algorithm that finds the appropriate balance between execution time and solution quality.

Bibliography

[1] Tobias Achterberg. "Constraint integer programming". PhD thesis. 2007.

[2] Tobias Achterberg, Timo Berthold, and Gregor Hendel. "Rounding and propagation heuristics for mixed integer programming". In: (2012), pp. 71–76.

[3] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. "Constraint integer programming: A new approach to integrate CP and MIP". In: (2008), pp. 6–20.

[4] Andrea Alberizzi, Paolo Grisi, and Alessandro Zani. "Analysis of the SIDC market on relationship between auctions and continuous trading". In: (2021), pp. 1–5. DOI: 10.23919/AEIT53387.2021.9627048.

[5] Robert Almgren and Neil Chriss. "Optimal execution of portfolio transactions". In: *Journal of Risk* 3 (2001), pp. 5–40.

[6] Rimmi Anand, Divya Aggarwal, and Vijay Kumar. "A comparative analysis of optimization solvers". In: *Journal of Statistics and Management Systems* 20.4 (2017), pp. 623–635. DOI: 10.1080/09720510.2017.1395182. eprint: https://doi.org/10.1080/09720510.2017.1395182. URL: https://doi.org/10.1080/09720510.2017.1395182.

[7] Donatas Bakšys and Leonidas Sakalauskas. "SIMULATION AND TESTING OF FIFO CLEARING ALGORITHMS". In: *INFORMATION TECHNOLOGY AND CONTROL* 39.1 (2010), pp. 25–31. ISSN: 1392-124X.

[8] Michael Bender, Martin Farach-Colton, and Miguel Mosteiro. "Insertion Sort is O(n log n)". In: *Theory of Computing Systems* 39 (June 2006). DOI: 10.1007/s00224-005-1237-z.

[9] Tobias Brünner and René Levínský. "Price discovery and gains from trade in asset markets with insider trading". In: 0.0 (2022), pp. 1–23. DOI: https://doi.org/10.1080/14697688.2020.1849782.

[10] Esra Buyuktahtakin. "Dynamic Programming Via Linear Programming". In: (Feb. 2011). DOI: 10.1002/9780470400531.eorms0277.

[11] W Matthew Carlyle, Johannes O Royset, and R Kevin Wood. "Lagrangian relaxation and enumeration for solving constrained shortest-path problems". In: *Networks: an international journal* 52.4 (2008), pp. 256–270.

[12] Chunand Chiang Chang, Yao-Min, Yiming Qian, Ritter, and Jay R. "Pre-market Trading and IPO Pricing". In: *Prev Sci* 23 (2016), pp. 774–786. DOI: https://doi.org/10.1007/s11121-021-01284-x. URL: https://doi.org/10.1007/s11121-021-01284-x.

[13] Xin Chen, Lan Liu, Zheng Liu, and Tao Jiang. "On the minimum common integer partition problem". In: *Algorithms and Complexity: 6th Italian Conference, CIAC 2006, Rome, Italy, May 29-31, 2006. Proceedings 6*. Springer. 2006, pp. 236–247.

[14] Xin Chen, Lan Liu, Zheng Liu, and Tao Jiang. "On the minimum common integer partition problem". In: *ACM Transactions on Algorithms (TALG)* 5.1 (2008), pp. 1–18.

[15] Carole Comerton-Forde and James Rydge. "Call auction algorithm design and market manipulation". In: *Journal of Multinational Financial Management* 16.2 (2006), pp. 184–198. ISSN: 1042-444X. DOI: https://doi.org/10.1016/j.mulfin.2005.06.002.

[16] Bradford Cornell and Marc R Reinganum. "Forward and futures prices: Evidence from the foreign exchange markets". In: *The Journal of Finance* 36.5 (1981), pp. 1035–1045.

[17] Gérard Cornuéjols, Javier Peña, and Reha Tütüncü. *Mixed Integer Programming: Theory and Algorithms*. 2nd ed. Cambridge University Press, 2018, pp. 140–160. DOI: 10.1017/9781107297340.009.

[18] S.A. Curtis. "The classification of greedy algorithms". In: *Science of Computer Programming* 49.1 (2003), pp. 125–157. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2003.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0167642303000340.

[19] Robert J Dakin. "A tree-search algorithm for mixed integer programming problems". In: *The computer journal* 8.3 (1965), pp. 250–255.

[20] K. Doksum. "Distribution-Free Statistics Based on Normal Deviates in Analysis of Variance". In: *Revue de l'Institut International de Statistique / Review of the International Statistical Institute* 34.3 (1966), pp. 376–388. ISSN: 03731138. URL: http://www.jstor.org/stable/1401257 (visited on 01/31/2023).

[21] Andreas Drexl and Alf Kimms. "Optimization guided lower and upper bounds for the resource investment problem". In: *Journal of the Operational Research Society* 52.3 (2001), pp. 340–351.

[22] Mayo-Wilson E., Grant S., and Supplee L.H. "Clearinghouse Standards of Evidence on the Transparency, Openness, and Reproducibility of Intervention Evaluations". In: *The Review of Financial Studies* 30.3 (2022), pp. 835–865. DOI: https://doi.org/10.1093/rfs/hhw032. URL: https://academic.oup.com/rfs/article/30/3/835/2669951.

[23] Cheoljun Eom and Jong Won Park. "Effects of the fat-tail distribution on the relationship between prospect theory value and expected return". In: *The North American Journal of Economics and Finance* 51 (2020), pp. 3–20. DOI: https://doi.org/10.1016/j.najef.2019.101052.

[24] Vladmir Estivill-Castro and Derick Wood. "A Survey of Adaptive Sorting Algorithms". In: *ACM Comput. Surv.* 24.4 (Dec. 1992), pp. 441–476. ISSN: 0360-0300. DOI: 10.1145/146370.146381. URL: https://doi.org/10.1145/146370.146381.

[25] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman och Co., 1990. ISBN: 0716710455.

[26] Robert S Garfinkel and George L Nemhauser. "Optimal political districting by implicit enumeration techniques". In: *Management Science* 16.8 (1970), B–495.

[27] Fred Glover. "Future paths for integer programming and links to artificial intelligence". In: *Computers & operations research* 13.5 (1986), pp. 533–549.

[28] Anupriya Gogna and Akash Tayal. "Metaheuristics: review and application". In: *Journal of Experimental & Theoretical Artificial Intelligence* 25.4 (2013), pp. 503–526.

[29] Ralph Gomory. *An algorithm for the mixed integer problem*. Tech. rep. RAND CORP SANTA MONICA CA, 1960.

[30] Ralph E Gomory. "Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem". In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art* (2010), pp. 77–103.

[31] Giulio Guerrieri. "Towards a Semantic Measure of the Execution Time in Call-by-Value lambda-Calculus". In: *CoRR* abs/1904.10800 (2019), pp. 57–72.

[32] Jiahao He, Jiheng Zhang, and Rachel Zhang. "A reduction from linear contextual bandit lower bounds to estimation lower bounds". In: (2022), pp. 8660–8677.

[33] Terrence Hendershott. "Electronic Trading in Financial Markets". In: *IT Professional Magazine* 3.4 (2003), pp. 10–14. DOI: 10.1109/MITP.2003.121622. eprint: https://www.proquest.com/docview/206330659?pq-origsite=gscholar&fromopenview=true. URL: https://www.proquest.com/docview/206330659?pq-origsite=gscholar&fromopenview=true.

[34] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. "Traveling salesman problem". In: *Encyclopedia of operations research and management science* 1 (2013), pp. 1573–1578.

[35] David L Huff. "Defining and estimating a trading area". In: *Journal of marketing* 28.3 (1964), pp. 34–38.

[36] John H Jackson. "The world trading system". In: *Law Quadrangle (formerly Law Quad Notes)* 34.1 (1989), p. 8.

[37] Michal Kaut and W Stein. *Evaluation of scenario-generation methods for stochastic programming*. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät ..., 2003.

[38] Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani, Abdallah Mahmoud Ibrahim AlTurani, and Nabeel Imhammed Zanoon. "Review on sorting algorithms a comparative study". In: *International Journal of Computer Science and Security (IJCSS)* 7.3 (2013), pp. 120–126.

[39] Tae Kyun Kim. "T test as a parametric statistic". In: *Korean journal of anesthesiology* 68.6 (2015), pp. 540–546.

[40] Thorsten Koeppl, Cyril Monnet, and Ted Temzelides. "Optimal clearing arrangements for financial trades". In: *Journal of Financial Economics* 103.1 (2012), pp. 189–203.

[41] Jiayi Li, Sumei Luo, and Guangyou Zhou. "Call auction, continuous trading and closing price formation". In: *Quantitative Finance* 21.6 (2021), pp. 1037–1065. DOI: 10.1080/14697688.2020.1849782. eprint: https://doi.org/10.1080/14697688.2020.1849782. URL: https://doi.org/10.1080/14697688.2020.1849782.

[42] Guohui Lin and Weitian Tong. "An improved approximation algorithm for the minimum common integer partition problem". In: *Information and Computation* 281 (2021), p. 104784.

[43] Jeff T Linderoth and Martin WP Savelsbergh. "A computational study of search strategies for mixed integer programming". In: *INFORMS Journal on Computing* 11.2 (1999), pp. 173–187.

[44] Xing-Hua Liu, Wen-Jin Wang, Yu-Xin Zhang, and Miao-Miao Lu. "Research on fat-tail phenomenon via artificial stock market modeling". In: *2020 International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)* (2020), pp. 38–41. DOI: 10.1109/ISCEIC51027.2020.00016.

[45] Vasilios Mavroudis and Hayden Melton. "Libra: Fair Order-Matching for Electronic Financial Exchanges". In: (). eprint: http://arxiv.org/abs/1910.00321. URL: http://arxiv.org/abs/1910.00321.

[46] B Meindl and Matthias Templ. "Analysis of commercial and free and open source solvers for linear optimization problems". In: (Aug. 2013).

[47] Alistair Milne. "The industrial organization of post-trade clearing and settlement". In: *Journal of Banking and Finance* 31.10 (2007), pp. 2945–2961. ISSN: 0378-4266, DOI: https://doi.org/10.1016/j.jbankfin.2007.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0378426607000829.

[48] Prabhaker Mishra, Uttam Singh, Chandra M Pandey, Priyadarshni Mishra, and Gaurav Pandey. "Application of student's t-test, analysis of variance, and covariance". In: *Annals of cardiac anaesthesia* 22.4 (2019), p. 407.

[49] G Montgomery and C. Runger. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*. Vol. 158. Blackwell Publishing Inc.; John Wiley and Sons; JSTOR; Wiley; Wiley (Blackwell Publishing), 1995, pp. 1–355. DOI: 10.1046;10.1111;10.2307.

[50] A Murtiyoso, F Matrone, M Martini, A Lingua, P Grussenmeyer, and R Pierdicca. "AUTOMATIC TRAINING DATA GENERATION IN DEEP LEARNING-AIDED SEMANTIC SEGMENTATION OF HERITAGE BUILDINGS". In: *Spatial Inf. Sci.* 2 (2022), pp. 317–324. DOI: https://doi.org/10.5194/isprs-annals-V-2-2022-317-2022.

[51] Severin Nilsson. "Introduction to Trade Matching". 2023.

[52] MS Pagano and RA Schwartz. "A closing call's impact on market quality at Euronext Paris". In: *Journal of financial economics* 68.3 (2003), pp. 439–484. ISSN: 0304-405X. DOI: 10.1016/S0304-405X(03)00073-4.

[53] Laurent Perron and Vincent Furnon. *OR-Tools*. Version v9.5. Google, Nov. 25, 2022. URL: https://developers.google.com/optimization/.

[54] N.V. Ploskas N.and Sahinidis. "Review and comparison of algorithms and software for mixed-integer derivative-free optimization". In: *J Glob Optim* 82 (Aug. 2021), pp. 433–462. DOI: https://doi.org/10.1007/s10898-021-01085-0.

[55] Luis F. Rojas-Muñoz, Santiago Sánchez-Solano, Macarena C. Martínez-Rodríguez, and Piedad Brox. "True Random Number Generator based on RO-PUF". In: *Conference on Design of Circuits and Integrated Circuits (DCIS)* 37 (2022), pp. 01–06. DOI: 10.1109/DCIS55711.2022.9970032.

[56] Omar Romero-Herna´ndez, Miguel de Lascurain Morhan, David Mun~oz Negro´n, Sergio Romero Herna´ndez, David G, Mun~oz Medina, Arturo A. Palacios Brun, Manuel A, Oneto Suberbie, and Jose E. Detta Silveira. "Business process modelling for a central securities depository". In: *Business Process Management Journal* 3.1 (2008), pp. 419–431. ISSN: 1463-7154. URL: https://www.emerald.com/insight/content/doi/10.1108/14637150810876706/full/html#abstract.

[57] Gerard Sierksma and Yori Zwols. *LINEAR AND INTEGER OPTIMIZATION*. Vol. 3. 2015, pp. 1–676. DOI: https://doi.org/10.1016/j.jbankfin.2007.03.002.

[58] David B Stewart. "Measuring execution time and real-time performance". In: 141 (2006), pp. 1–15. URL: http://www.inhand.com.

[59] Joseph E. Stiglitz. "The Role of the State in Financial Markets". In: *The World Bank Economic Review* 7.1 (1993), pp. 19–52. DOI: https://doi.org/10.1093/wber/7.suppl_1.19.

[60] *Stocks traded, total value*. 2023. URL: https://data.worldbank.org/indicator/CM.MKT.TRAD.CD?end=2020&start=1975&view=chart&year=2019.

[61] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009.

[62] My T Thai. "Approximation algorithms: LP relaxation, rounding, and randomized rounding techniques". In: *Lecture Notes, University of Florida* (2013).

[63] Weitian Tong and Guohui Lin. "An improved approximation algorithm for the minimum common integer partition problem". In: *Algorithms and Computation: 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*. Springer. 2014, pp. 353–364.

[64] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[65] Rahul Verma and Gökçe Soydemir. "The impact of US individual and institutional investor sentiment on foreign stock markets". In: *The Journal of Behavioral Finance* 7.3 (2006), pp. 128–144.

[66] Wang Xiang. "Analysis of the Time Complexity of Quick Sort Algorithm". In: *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*. Vol. 1. 2011, pp. 408–410. DOI: 10.1109/ICIII.2011.104.

[67] Wenjie Xu, Yuning Jiang, Emilio T Maddalena, and Colin N Jones. "Lower bounds on the worst-case complexity of efficient global optimization". In: *arXiv preprint arXiv:2209.09655* (2022).

[68] You Yang, Ping Yu, and Yan Gan. "Experimental study on the five sort algorithms". In: *2011 Second International Conference on Mechanic Automation and Control Engineering*. 2011, pp. 1314–1317. DOI: 10.1109/MACE.2011.5987184.

A Appendix

A.1 Results

In this section, two complimentary graphs for the result are provided.

Figure A.1: Stability test of a uniform distribution with mean = 500, size = 10000 and the buy ratio of 50 %. Ten tests were conducted on the data set for each algorithm and all results are presented in the graph with the with the Cluster 3-1 and Cluster 3-1 & 1-3 algorithms excluded

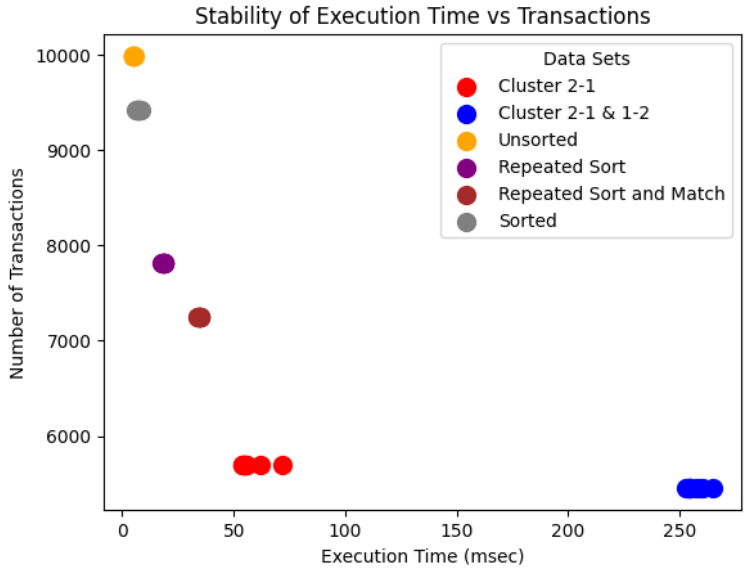

Figure A.2: Stability test of a uniform distribution with mean = 500, size = 10000 and the buy ratio of 5 %. Ten tests were conducted on the data set for each algorithm and all results are presented in the graph with the with the Cluster 3-1 and Cluster 3-1 & 1-3 algorithms excluded

A.2 Heuristic Methods

In this section, the logic for two described heuristics methods used by the solvers is presented in the form of pseudo code.

Algorithm 8 Branch and Bound

```
1: while the priority queue is not empty do
2:     dequeue the node with the highest priority
3:     if the node's lower bound is greater than the current upper bound then
4:         prune the node
5:     else if the node is a leaf node then
6:         update the upper bound if necessary
7:     else
8:         generate all child nodes of the current node
9:         compute the lower bound of each child node
10:        insert each child node into the priority queue with a priority based on its lower
   bound
11:    end if
12: end while
```

Algorithm 9 Cutting Plane

```
1: procedure CUTTINGPLANE
2:     Initialize the linear programming problem and set an initial feasible solution.
3:     Solve the linear programming problem to obtain a basic feasible solution.
4:     while the current solution is not optimal do
5:         if the current solution is infeasible then
6:             Add a constraint that excludes the current solution and go to step 2.
7:         else
8:             Solve the linear programming problem to obtain a new basic feasible solution.
9:             if the new solution is worse than the current solution then
10:                Add a constraint that excludes the new solution and go to step 2.
11:            else
12:                Update the current solution and go to step 2.
13:            end if
14:        end if
15:    end while
16:    Return the optimal solution.
17: end procedure
```

.