University of Cape Town

STA5071Z

Optimisation

Optimising Cryptocurrency Arbitrage

Author:
Jared Lakhani

Student Number:
LKHJAR001

30 September 2024

Project Repository

Access the source code and project files for this report on GitHub:
https://github.com/LKHJAR001/STA5071Z-Optim.git

Contents

1 Introduction

In this report, we aim to explore various optimisation methods to determine the best approach for exploiting arbitrage opportunities between two distinct cryptocurrencies traded on different exchanges. Arbitrage, the practice of capitalizing on price differences between markets, poses several challenges, including determining the optimal timing and the amount to of the particular cryptocurrency to arbitrage in order to maximize profits. We address these questions by formulating them as an optimization problem. Our approach combines the rigor of Linear Programming, which we extend into Goal Programming to handle multiple objectives, with the flexibility of heuristic optimization techniques, specifically Simulated Annealing and Genetic Algorithms. Through these methods, we investigate when and how much one should arbitrage to achieve the most profitable outcome.

Now, any practical optimisation solution must account for a set of real-world restrictions. We introduce three overarching constraints that apply across all optimization methods, ensuring that the solution remains grounded in the operational realities of cryptocurrency arbitrage. Cryptocurrency exchanges typically impose daily volume limits for regulatory compliance, meaning traders cannot exceed a certain amount of trades per cryptocurrency each day. This ensures that the total trading volume remains within allowable limits, keeping strategies in line with exchange-imposed caps. Additionally, arbitrage opportunities rely on available liquidity—trades can only be executed if there is sufficient liquidity to fulfill buy or sell orders. This constraint ensures that our optimization only includes feasible trades based on market conditions. Finally, handling one trade at a time minimizes risks like execution errors and market impact, preventing simultaneous requests from overwhelming the exchange and ensuring trades are processed smoothly.

2 Data: Historical Price and Volume

Our study focuses on two prominent cryptocurrencies, Bitcoin (BTC) and Ethereum (ETH), using their spot market data. Historical price information is sourced from two cryptocurrency exchanges, Binance and Kucoin. To maximize the potential for arbitrage, we have selected a day marked by high volatility - Monday, August 5^{th} of 2024 - when price disparities were most pronounced. Price differences are calculated using the 1-minute candle closes, meaning arbitrage opportunities are evaluated only at the top of each minute. Additionally, available liquidity is assumed to be a fraction of the volume traded during that minute.

One can access the data online via Binance's Market Data Archive (Closing prices in Column 5 with Volume traded in Column 6) and Kucoin's Market Data Archive (Closing prices in Column 3 with Volume traded in Column 6) by merely searching for the corresponding asset: spot (not any derivatives product) 'BTCUSDT' or 'ETHUSDT' and ensure that the '1m' granularity is selected.

3 Literature Review

Optimising cryptocurrency arbitrage is a complex process that focuses on maximizing profit by exploiting price discrepancies across multiple exchanges while managing risks and costs. Several studies have developed strategies combining mathematical optimization methods, such as linear programming (LP) and goal programming, with computational algorithms to tackle the inherent challenges of cryptocurrency markets. The highly volatile and fragmented nature of these markets presents both opportunities and risks, making optimization techniques essential for successful arbitrage trading.

A significant body of literature on arbitrage optimisation exists within both traditional finance and cryptocurrency markets. For example, [22] utilized LP to model cryptocurrency arbitrage with constraints such as transaction fees, withdrawal limits, and exchange delays. This methodology allows traders to determine the optimal capital allocation across trades, aiming to maximize net returns while adhering to various restrictions. [17] expanded upon this by incorporating more advanced non-linear optimization algorithms, such as genetic algorithms (GA) and simulated annealing (SA), which are capable of handling the dynamic nature of cryptocurrency markets. These stochastic methods optimize arbitrage strategies even under highly fluctuating price differences, a common scenario in cryptocurrency markets due to their volatility.

Latency, or the time delay in executing trades across different exchanges, is another critical factor that impacts arbitrage. If the latency is too high, price discrepancies may vanish before trades are completed, reducing profitability. Algorithmic trading systems are widely used to automate arbitrage strategies, reacting to price differences in real-time. Studies such as [5] have examined the role of high-frequency trading (HFT) in optimizing arbitrage, noting that minimizing latency is crucial for success in fast-moving markets like cryptocurrencies.

Machine learning models, particularly reinforcement learning (RL), have also gained prominence in optimizing cryptocurrency arbitrage. [19] explored how RL models can dynamically adjust trading strategies based on historical price movements and adapt to changing market conditions. These systems can continuously improve by learning from market environments, making them ideal for handling the fast-paced and unpredictable nature of cryptocurrency trading.

Liquidity management plays an essential role in optimising arbitrage, as exchanges with low liquidity can present significant challenges. Large trades on low-liquidity platforms can result in slippage, where the price of the asset moves unfavorably during trade execution. [12] highlighted liquidity constraints in cryptocurrency markets, emphasizing the importance of managing liquidity when executing arbitrage strategies. By optimising for liquidity, traders can avoid adverse market movements that reduce profitability, and instead target exchanges with sufficient market depth to execute large trades without significant price shifts.

3.1 Linear Programming

Linear Programming (LP) is a mathematical method used to optimize an objective function, typically maximizing or minimizing, subject to linear equality and inequality constraints. The decision variables in an LP are continuous, and both the objective function and constraints must be linear [7]. Now Mixed-Integer Programming (MIP) is an extension of linear programming that allows some decision variables to be restricted to integer values. MIPs are useful for modeling situations where decisions are inherently discrete. Solving MIP problems is more complex than LP because the integer constraints make the problem non-convex, often requiring combinatorial algorithms like branch-and-bound or cutting planes [23].

Dual variables (or shadow prices) in linear programming provide crucial sensitivity information. They represent the change in the optimal objective function value if the right-hand side of a constraint is increased by one unit. For binding constraints, a positive dual variable suggests that relaxing the constraint would improve the objective function, while for non-binding constraints, the dual variable is zero [20]. Furthermore, duality theory applies cleanly to continuous problems. When you introduce integer variables, the concept of duality becomes less straightforward, and many solvers (including 'Rglpk') do not return dual variables for mixed-integer problems.

Additionally, MIP problems are computationally difficult and time-consuming to solve due to their combinatorial nature, especially for large instances. Solving large MIP problems can require significant computational resources [20].

3.2 Goal Programming

Goal programming is a technique used in multi-objective optimization where multiple goals or objectives must be satisfied simultaneously. It extends linear programming by accommodating the possibility that all goals may not be achievable in their entirety. Instead of optimizing a single objective, GP minimizes the deviations from predefined goals.

In Archimedean goal programming, a weighted sum of deviations from each goal is minimized. The user assigns weights to each goal based on its importance, and the optimization focuses on minimizing the weighted sum of deviations. However, the solution may be sensitive to the choice of weights, leading to biased results if the weights are not carefully selected [4].

In contrast, Tchebychev goal programming minimises the maximum deviation across all goals, emphasizing the worst-performing goal. This approach is useful when the decision-maker wants to avoid large deviations in any goal, ensuring a more balanced solution [27]. It is less sensitive to the weighting scheme compared to the Archimedean method but may lead to more conservative solutions, as it focuses on reducing the largest deviation rather than optimizing the overall performance across goals [24].

3.3 Simulated Annealing

Simulated annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reduce defects, resulting in a more stable structure. In SA, the algorithm searches for the global minimum of an objective function by mimicking this process. It starts with a high "temperature" that allows the algorithm to explore a wide range of solutions, including worse ones, to avoid local minima. As the temperature decreases, the search becomes more refined, focusing on local improvements [18].

Now in comparison, LP is a deterministic optimization method used for solving linear objective functions subject to linear constraints. It guarantees the optimal solution when a feasible solution exists, provided the problem is convex [3]. SA, on the other hand, is a heuristic or metaheuristic method used to solve non-linear, non-convex, or combinatorial problems. It is well-suited for problems where the search space is large and complex, where deterministic methods like LP might fail or become inefficient [6]. Unlike LP, which relies on a gradient or deterministic search method, SA's probabilistic nature allows it to escape local minima by accepting worse solutions with a certain probability, which decreases as the temperature lowers.

One of the main advantages of SA is its flexibility. It can handle problems with complex or poorly behaved objective functions that other algorithms might struggle with [1]. SA does not require the objective function to be differentiable or continuous, which is a limitation in gradient-based methods like LP. However, SA has its drawbacks. It can be computationally expensive, particularly for large problem spaces, as the solution process may involve evaluating a large number of solutions before convergence. Furthermore, it does not guarantee finding the global optimum, and the quality of the solution heavily depends on the cooling schedule and initial temperature [9].

3.3.1 Cooling Schedules

The cooling schedule in SA governs how the temperature decreases over time. Two common cooling schedules are geometric and logarithmic. In a geometric cooling schedule, the temperature is reduced by a constant factor at each step (that is, $T_i = T_0\alpha^i$ where the cooling rate $0 < \alpha < 1$). The cooling rate typically ranges between 0.8 and 0.99 [11]. A faster rate (lower α) decreases the temperature quickly, leading to quicker convergence but potentially missing the global minimum. A slower rate (higher α) ensures thorough exploration but increases computation time [16].

Logarithmic cooling follows the schedule $T_i = \frac{T_0}{1+\alpha \log(1+i)}$. While this approach can lead to a more thorough exploration of the solution space, it often requires more iterations to converge, making it less efficient in practice [14].

The best cooling schedule and initial temperature depend on the problem. Generally, geometric cooling is preferred for its balance between speed and effectiveness. The initial temperature should be high enough to allow exploration across the solution space, but not so high that it becomes inefficient [1].

3.4 Genetic Algorithms

Genetic algorithms (GAs) are optimization techniques modeled after the process of natural evolution. They operate by maintaining a population of candidate solutions that evolve over time through selection, crossover (recombination), and mutation. At each generation, individuals are evaluated based on a fitness function, which determines their quality as potential solutions [13]. The fittest individuals are selected to produce offspring for the next generation, aiming to improve the population's overall quality with each iteration.

Pros of GAs include their global search capability, which allows them to explore large, complex, and non-linear solution spaces effectively [15]. Unlike gradient-based methods, GAs do not require differentiable functions, making them suitable for a wide range of problems, including discrete and multimodal optimization challenges. Additionally, they are less likely to get trapped in local optima compared to traditional methods due to the diversity in the population [21].

However, cons of GAs include their slower convergence rate compared to some optimization techniques, such as simulated annealing. GAs require careful tuning of parameters, such as mutation rate, population size, and crossover rate to balance exploration (searching new areas of the solution space) and exploitation (refining existing solutions). Poor tuning may result in suboptimal performance or premature convergence, where the algorithm converges to a local optimum without exploring the global solution space fully.

3.4.1 Selection

Selection is one of the key genetic operators in genetic algorithms, responsible for choosing individuals from the current population to create offspring for the next generation. The main goal of selection is to ensure that individuals with better fitness, or higher performance according to a fitness function, have a higher likelihood of passing on their traits [15]. However, it is also important to maintain diversity in the population by allowing some lower-performing individuals to contribute, preventing premature convergence on suboptimal solutions [21].

Two common methods of selection are informal tournament selection and fitness-proportional selection, which we employ in this study. There are other selection types that exist, although we will not detail them here.

Informal selection works by randomly choosing a group of individuals (the ”tournament”) from the population and then selecting the individuals with the best fitness within that group. This method creates high selection pressure, meaning that the best individuals in each tournament are more likely to be selected to reproduce, thereby increasing the likelihood of quickly finding an optimal solution [13]. High selection pressure tends to speed up convergence, but it can also lead to premature convergence, where the population loses diversity too quickly and gets trapped in local optima [26].

On the other hand, fitness-proportional selection, such as roulette wheel selection, selects individuals probabilistically based on their fitness [8]. Individuals with higher fitness have a greater probability of being selected, but even those with lower fitness still have a chance of being chosen. This method leads to lower selection pressure, as the selection process allows a broader range of individuals to contribute to the next generation. While this helps maintain diversity and reduces the risk of premature convergence, it can result in slower convergence towards an optimal solution, as the algorithm spends more time exploring various possibilities before focusing on high-performing individuals [21].

3.4.2 Recombination (Crossover)

Recombination, also known as crossover, is a genetic algorithm operator that combines the genetic information from two parent solutions to produce offspring [13]. It mimics biological reproduction by mixing parent traits, allowing for the exploration of new areas in the solution space. Recombination promotes diversity and innovation by generating novel combinations of genes, which can enhance the algorithm’s ability to explore and avoid local optima [21].

Uniform crossover works by randomly deciding, at each gene position, which parent will contribute that gene. This results in more random mixing of genetic material, promoting higher genetic diversity [21]. The randomness, however, can lead to slower convergence, as offspring inherit small, scattered gene fragments from both parents, making it harder to preserve useful gene structures across generations [26]. The increased diversity can help avoid premature convergence to suboptimal solutions, but it slows the algorithm’s ability to fine-tune and exploit good solutions [8].

N-point crossover, on the other hand, selects one or more crossover points where the genetic material is split and swapped between parents [13]. This method transfers larger, more cohesive blocks of genes, allowing offspring to inherit intact gene sequences, which can result in faster convergence as well-performing gene blocks are more likely to be preserved [15]. However, this reduced genetic diversity can cause the population to converge too quickly, increasing the risk of missing better solutions.

3.4.3 Mutation

Mutation is a critical operator in genetic algorithms that introduces variability into the population of solutions by altering individual components of a candidate solution. This process helps maintain genetic diversity within the population, preventing premature convergence to local optima and enabling exploration of the search space. Without mutation, the algorithm may rely too heavily on the existing population's genetic material, potentially stagnating in suboptimal regions of the solution space [10]. Mutation is essential because, while other operators like crossover exploit existing solutions to combine promising features, mutation explores new regions of the search space by introducing small random changes. [2].

There are many different mutation strategies, yet we will only be focusing on scramble and swap mutation. Scramble mutation randomly shuffles a subset of elements in a candidate solution. This method introduces large but localized changes, effectively reordering portions of the solution without disrupting the overall structure too drastically [8]. Since the order of the genes in the scrambled subset changes completely, it encourages exploration of new regions in the solution space. The extent of exploitation in scramble mutation is low since it disrupts the order significantly, making the solution less focused on local optimization.

In contrast, swap mutation randomly selects two positions within a solution and swaps their values. This results in smaller, more localized changes compared to scramble mutation [21]. This mutation type tends to explore less extensively, as it modifies the chromosome only slightly. However, swap mutation favors exploitation because of its localized nature. By making small, incremental changes, it can help fine-tune the solution without significantly altering its structure [21].

Now, increasing the mutation rate would lead to more frequent alterations in candidate solutions, increasing the algorithm's exploration capabilities. While this can enhance the algorithm's ability to escape local optima, an excessively high mutation rate risks overwhelming the search process with random changes, making it harder for the population to converge on high-quality solutions [10].

3.4.4 Replacement

Replacement in genetic algorithms is the process of determining how new individuals (offspring) replace members of the population, maintaining the size of the population constant. Effective replacement ensures diversity within the population while gradually improving the fitness of individuals over generations [9]. The two main types of replacement strategies are generational replacement and steady-state replacement.

Generational replacement refers to the process where the entire population is replaced by offspring after each generation. This approach promotes exploration as the algorithm can radically shift the population, potentially escaping local optima. However, this can also lead to a loss of useful information, as well-performing individuals may be discarded [21].

In contrast, steady-state replacement introduces offspring incrementally, replacing only a few individuals at a time, typically based on their fitness levels. This method tends to favor exploitation since only the fittest individuals survive and are gradually replaced by better-performing offspring. However, the downside is that steady-state replacement can suffer from getting stuck in local optima due to insufficient exploration [2]. There are several types of steady-state replacement strategies in genetic algorithms, yet this study will employ Elitist replacement: where the best individuals are always retained, ensuring that the highest fitness solutions are never lost [8].

4 Linear Programming

We start this section by previewing the notation to be used in this section (as well as for the rest of this report). Afterwhich, we detail the objective function and what the corresponding constraints are. We include an example at the end for our specific problem: where this specific arbitrageur's specifications will be used throughout the study.

4.1 Notation

Since we are working with two cryptocurrenices, let i denote the particular crypto: $i = 1$ = BTC and $i = 2$ = ETH.

Our price data originates from the '1-min' closes from the entire day of Monday the 5th of August, hence we let t denote time, and since there are 1440 minutes in a day: $t = 1, 2, \ldots T = 1440$.

Now our main variables of concern are $x_{i,t}$ - being the amount of the i^{th} crypto arbitraged at the t^{th} time (continuous non-negative variables). Furthermore, we let $\Delta P_{i,t}$ be the absolute price difference between the two exchanges for the i^{th} crypto at the t^{th} time.

4.2 Objective and Constraints

Objective: Maximise Profit The core objective of any arbitrageur is to make as much profit as one can, hence our objective function is to maximise the profit from arbitraging. Hence we maximise:

$$\begin{aligned} f(x_{i,t}) =& \Delta P_{1,1}x_{1,1} + \Delta P_{1,2}x_{1,2} + \ldots + \Delta P_{1,1440}x_{1,1440} + \\ & \Delta P_{2,1}x_{2,1} + \Delta P_{2,2}x_{2,2} + \ldots + \Delta P_{2,1440}x_{2,1440} \\ =& \sum_{i=1}^{2}\sum_{t=1}^{T=1440} \Delta P_{i,t}x_{i,t} \end{aligned} \tag{1}$$

Constraint I: Daily Volume Restriction Since cryptocurrency exchanges impose daily volume limits on traders (usually part of regulatory compliance), we find it apt to include this as a constraint for both BTC and ETH. Hence the total daily trading volume for the i^{th} crypto is V_i, hence:

$$\sum_{t=1}^{T=1440} x_{i,t} \leq V_i \;\forall i = 1, 2 \tag{2}$$

Constraint II: Liquidity Available Now one can only arbitrage a particular crypto i at time t, if other traders are willing to buy/sell that amount of crypto at that time. That is, if the liquidity $L_{i,t}$ is available for the i^{th} crypto at time t. Hence:

$$x_{i,t} \leq L_{i,t} \;\forall i, t \tag{3}$$

Constraint III: One-at-a-Time Now it is prudent for an arbitrageur to minimize the overall number of requests (buy/sell orders) they send to the crypto exchange (or broker). Additionally, handling one order at a time ensures that each request is processed and confirmed by the exchange individually, reducing the risk of errors or failures affecting multiple orders. Being such, we find it fruitful to restrict the arbitraging such that only a single crypto i may be traded at a time t, and is done to prevent excess 'traffic' to the crypto exchange.

Now to satisfy this One-at-a-Time constraint, we need to utilize Big-M constraints, where we first define non-negative binary variables:

$$z_{i,t} = \begin{cases} 1 & x_{i,j} > 0 \\ 0 & x_{i,j} = 0 \end{cases}$$

Hence $\forall t$ our Big-M constraints are, for $M >> c$:

$$\begin{aligned} x_{1,t} &\leq M(1 - z_{2,t}) \\ x_{2,t} &\leq M(1 - z_{1,t}) \end{aligned} \tag{4}$$

with

$$z_{1,t} + z_{2,t} = 1 \tag{5}$$

Now as an example, since $x_{1,1} \leq M(1 - z_{2,1})$, this implies that if $x_{2,1} > 0$ (ETH at $t = 1$ was arbitraged), this would mean $z_{2,1} = 1$ hence ($z_{1,1} = 0$) implying that $x_{1,1} = 0$ (BTC at $t = 1$ can not be arbitraged).

Linear Programming Summary We recap what was done prior, where we state the objective function with constraints of the LP.

- Maximise: $\sum_{i=1}^{2} \sum_{t=1}^{T=1440} \Delta P_{i,t} x_{i,t}$
- Subject to:

$$\begin{aligned}
\sum_{t=1}^{T=1440} x_{i,t} &\leq V_i \text{ for } \forall i \\
x_{i,t} &\leq L_{i,t} \ \forall i, t \\
x_{1,t} &\leq M(1 - z_{2,t}) \ \forall t \\
x_{2,t} &\leq M(1 - z_{1,t}) \ \forall t \\
z_{1,t} + z_{2,t} &= 1 \ \forall t \\
x_{i,t}, z_{i,t} &\geq 0 \ \forall i, t
\end{aligned}$$

4.2.1 Matrix Forms

This section details the matrix forms of Equations 1, 2, 3, 4 and 5. We know our solution vector (column form) looks as such:

$$\mathbf{x} = \begin{bmatrix} x_{1,1}, x_{1,2}, \ldots x_{1,1440}, x_{2,1}, x_{2,2}, \ldots x_{2,1440}, z_{2,1}, \ldots z_{2,1440}, z_{1,1}, \ldots z_{1,1440} \end{bmatrix}'$$

Objective: Maximise Profit Maximising our objective function as in Equation 1 is equivalent to maximising:

$$\begin{bmatrix} \Delta P_{1,1} & \ldots & \Delta P_{1,1440} & \Delta P_{2,1} & \ldots & \Delta P_{2,1440} & 0 & \ldots & 0 \end{bmatrix} \mathbf{x}$$

Constraint I: Daily Volume Restriction Satisfying our first constraint as in Equation 2 is equivalent to ensuring:

$$\begin{bmatrix} \mathbf{1}_{1\times1440} & \mathbf{0}_{1\times1440} & \mathbf{0}_{1\times2880} \\ \mathbf{0}_{1\times1440} & \mathbf{1}_{1\times1440} & \mathbf{0}_{1\times2880} \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$$

Constraint II: Liquidity Available The second constraint as in Equation 3 is equivalent to ensuring:

$$\begin{bmatrix} \mathbf{I}_{2880} & \mathbf{0}_{2880\times2880} \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} L_{1,1} \\ \vdots \\ L_{2,1440} \end{bmatrix}$$

Constraint III: One-at-a-Time Lastly, our Big-M constraints in Equations 4 and 5 are equivalent to:

$$\begin{bmatrix} \mathbf{I}_{2880} & M\mathbf{I}_{2880} \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} M \\ \vdots \\ M \end{bmatrix}_{2880\times1}$$

and

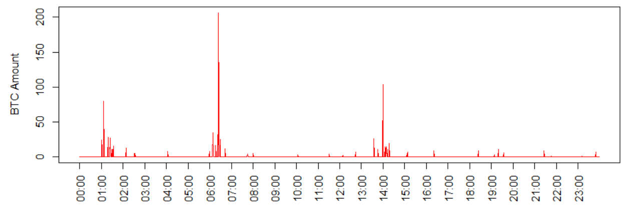
$$\begin{bmatrix} \mathbf{0}_{1440\times2880} & \mathbf{I}_{1440} & \mathbf{I}_{1440} \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{1440\times1}$$

4.3 Our Example Problem

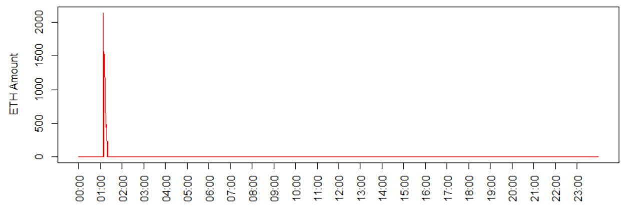
We showcase an example where we posit that there exists an arbitrageur whom is subject by the exchanges to trade a maximum of 1000 units of BTC, and 10000 units of ETH per day (crypto exchanges have different 'tiers' of clients who are subject to different daily volume limits). This example arbitrageur is utilized throughout the study, except in the Goal Programming Section 5 - where different goals are specified.

Figures 1a and 1b display the solutions, where one is able to scrutinize how much BTC and ETH one should arbitrage (and at what times during the day) where the maximum profit achieved was $\$234,612.9$. We note that the maximum units of both BTC and ETH daily volume limits were needed to be arbitraged to obtain this profit (that is, 1000 and 10000 units of BTC and ETH respectively). Additionally, where BTC or ETH were arbitraged, the amounts equalled the liquidity available at those time points: if $x_{i,t} > 0$ then $x_{i,t} = L_{i,t}\ \forall i, t$. We also note that the One-at-a-Time constraint was met. If one examines the price difference 'spike' at $\approx 01:00$ in Figures 2a and 2b (these 'spikes' occur at similar times due to ETH's high correlation with BTC), notice how ETH was arbitraged at this time, yet BTC was not - justifying our One-at-a-Time constraint being met.

Additionally, since our LP is a mixed-integer problem we do not have dual variables - one is unable to say how our maximum profit objective function is improved if one relaxes any of the constraints by one unit.

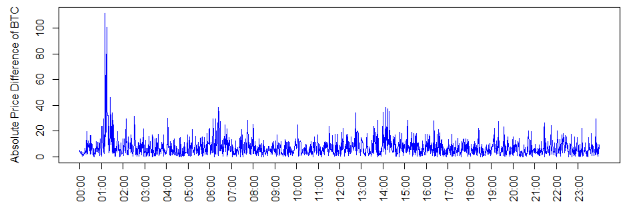


(a) Amount of BTC Arbitraged

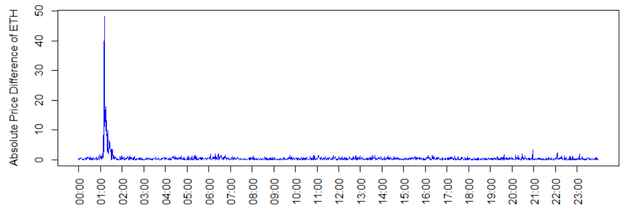


(b) Amount of ETH Arbitraged

Figure 1: Solution for Linear Programming Example



(a) Absolute Price Difference of BTC



(b) Absolute Price Difference of ETH

Figure 2: Absolute Price Differences between Exchanges

5 Goal Programming

Following the linear programming - which assumes all constraint to be strict, we find it apt to question what could occur if we assume some of the constraints are "soft" and can be relaxed or violated to some degree. A worthwhile endeavor would be to ascertain the extent to which the daily volume constraints could be relaxed to increase the profit made on that day. Being such, we transform our previous LP objective (ie. to maximise profit), as well as both daily volume constraints to corresponding goals in our multiobjective goal programming formulation.

Hence, our goals would be to arbitrage to make at least $\$g_1$ such that daily volume does not exceed g_2 units of BTC and g_3 units of ETH. We define $z_1^* = \sum_{i=1}^{2}\sum_{t=1}^{T=1440} \Delta P_{i,t} x_{i,t}$ (the LP objective), $z_2^* = \sum_{t=1}^{T=1440} x_{1,t}$ (BTC volume constraint) and $z_3^* = \sum_{t=1}^{T=1440} x_{2,t}$ (ETH volume constraint).

We present two different types of goal programming in the subsequent sections, where we present the goal programming formulation in a linear programming form. Afterwhich, we undergo a sensitivity analysis by varying the weights attributed to each goal.

5.1 Archimedean Goal Programming

We set our goals such that we would like to arbitrage to make at least $g_1 = \$100,000$ such that daily volume does not exceed $g_2 = 100$ units of BTC and $g_3 = 1,000$ units of ETH. We note the maximium profit that one could obtain from doing this to be \$59,450 (from LP - ie. assuming constraints are strict). Hence we know that we are certainly going to 'break' the constraints - the nature (the tradeoffs) by which we do must now be studied. The goal programming formulation is given as such:

Minimise: $w_1 d_1^- + w_2 d_2^+ + w_3 d_3^+$
Subject to:

$$\begin{aligned}
&z_1^* - d_1^+ + d_1^- = \$g_1 \\
&z_2^* - d_2^+ + d_2^- = g_2 \\
&z_3^* - d_3^+ + d_3^- = g_3 \\
&x_{i,t} \leq L_{i,t} \;\; \forall i, t \\
&x_{1,t} \leq M(1 - z_{2,t}) \;\; \forall t \\
&x_{2,t} \leq M(1 - z_{1,t}) \;\; \forall t \\
&z_{1,t} + z_{2,t} = 1 \;\; \forall t \\
&x_{i,t}, z_{i,t}, d_k \geq 0 \;\; \forall i, t, k
\end{aligned}$$

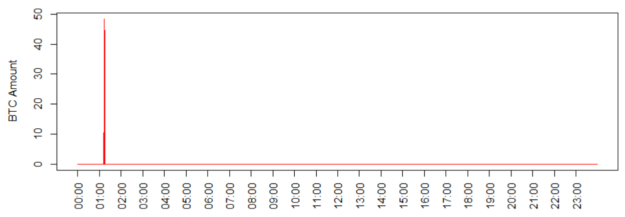
Being such, our solution vector $\mathbf{x}$ is now equivalent to the LP, except with deviational variables appended as such:

$$\mathbf{x} = \left[x_{1,1}, x_{1,2}, \ldots x_{1,1440}, x_{2,1}, x_{2,2}, \ldots x_{2,1440}, z_{2,1}, \ldots z_{2,1440}, z_{1,1}, \ldots z_{1,1440}, d_1^+, d_1^-, d_2^+, d_2^-, d_3^+, d_3^-\right]'$$

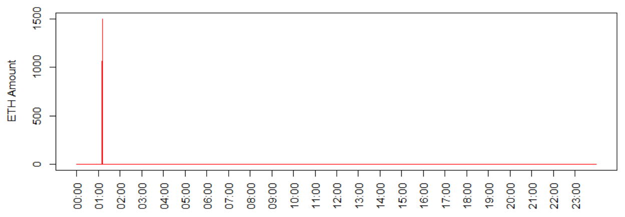
5.1.1 An Example:

An example of the archimedean goal programming is given as follows: we set $w_1 = 0.1, w_2 = 0.8$ and $w_3 = 0.1$ - implying that we are not as strict to achieve our $g_1 = \$100,000$ profit goal nor are we strict on satisfying the daily volume goal for ETH of $g_3 = 1000$ units. We are more rigid (in a relative sense) in meeting the daily volume goal for BTC however, seeing as we have attributed the largest weight as w_2.

Furthermore, we solve the problem to attain $d_3^+ = 1130.837$ with all other deviation variables equaling zero. This implies that we exceeded the daily ETH constraint of 1000 units by 1130, 837 units - suggesting a 1.13 (1130/1000) excess of daily volume of ETH, arbitraging to obtain a profit of $g_1 = \$100,000$. We note that the strict daily volume constraint of BTC was met as well as our profit goal g_1, since all other deviational variables were solved to be zero. Figures 3a and 3b illustrate the solutions (note that it may look like BTC and ETH are arbitraged at the same time, this is not the case however as shown in Figures 4a and 4b).

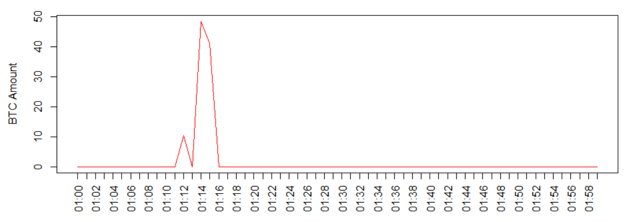


(a) Amount of BTC Arbitraged

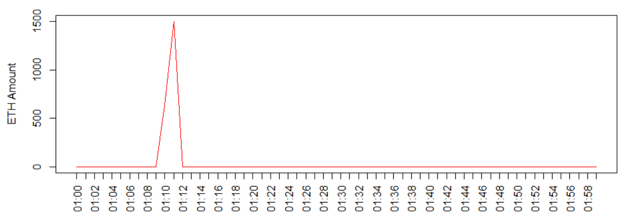


(b) Amount of ETH Arbitraged

Figure 3: Solution for Archimedean Example for the Entire Day



(a) Amount of BTC Arbitraged



(b) Amount of ETH Arbitraged

Figure 4: Solution for Archimedan Example from `01:00` to `02:00`

5.1.2 Sensitivity Analysis

We find it apt to now vary the weights w_1, w_2 and w_3 to get a greater understanding as to how one can break the daily volume constraints g_2 and g_3 to achieve the maximum profit g_1 (we ensure $\sum_{k=1}^{3} w_k = 1$ so relative importance may be conferred on the respective goals). We refer to large values of w_k as strict, as this implies that greater importance is placed on the corresponding k goal to be achieved.

Upon further review, we find that regardless of value of w_1, the maximum profit goal of $g_1 = \$100,000$ is always achieved. Hence, we fix $w_1 = 0.1$ and merely focus on varying w_2 and $w_3 = 1 - w_1 - w_2$ for $w_2 \in [0.1, 0.8]$.

Now Figure 5 illustrates how varying w_2 (and thus w_3) results in deviations from the respective daily volume constraint goal g_2 or g_3. A noticeable aspect of this is the sensitivity of d_2^+ (deviation from g_2 - the daily BTC volume goal) to w_2 (and thus w_3). For non-strict w_2 (so strict w_3), we can see that we are required to arbitrage BTC to almost $10\times$ the amount of g_2 (the daily BTC volume goal) to obtain the maximum profit of g_1 (where $d_3^+ = 0$ suggesting the ETH daily volume restriction is not breached). Contrarily, noting d_3^+ for non-strict w_3 (strict w_2), we are only required to arbitrage ETH to double the amount of g_3 (where $d_2^+ = 0$).

From this, we posit that if one were penalized by the crypto exchange for exceeding a daily crypto limit (example, charged additional fees per limit breach), yet wanted to still achieve a maximum profit of $g_1 = \$100,000$, one should do so by exceeding the ETH daily volume constraint (by only double) and not the BTC daily volume constraint (as it would require one to exceed said limit by $\approx 10\times$). There is, of course a 'middle ground', where both $d_2^+, d_3^+ > 0$ - suggesting an excess in both crypo daily limits occuring when $w_2 \in [0.4, 0.65]$.

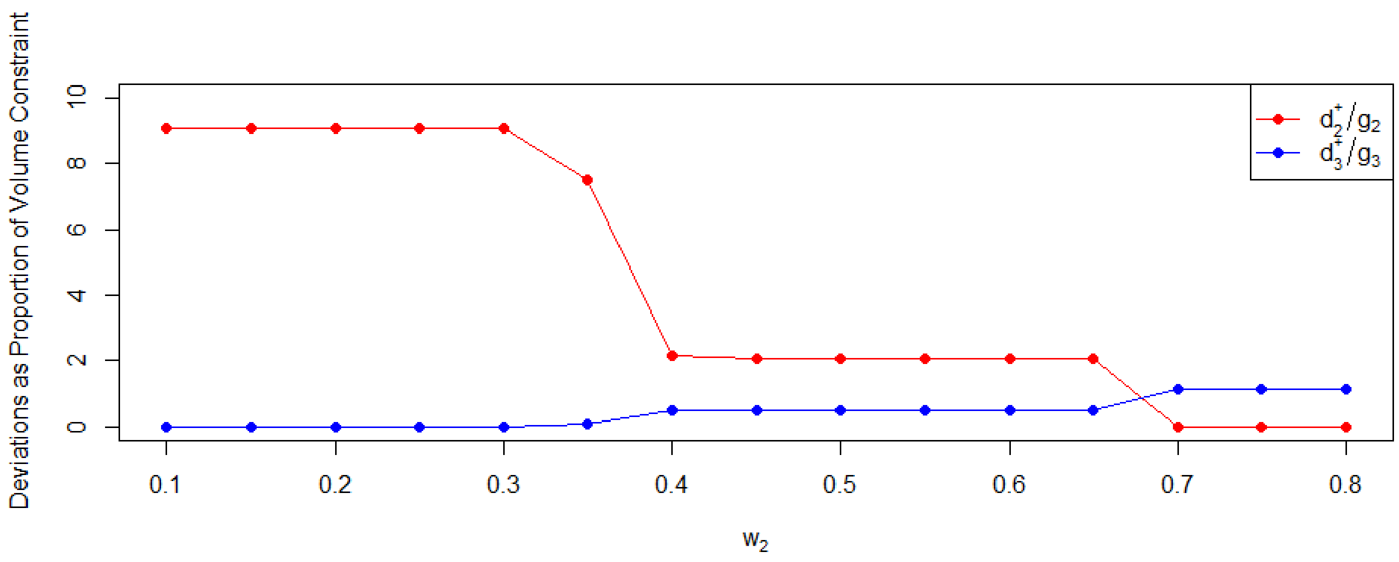


Figure 5: Deviations (d_2^+, d_3^+) as a Proportion of g_2, g_3

5.2 Tchebychev Goal Programming

An alternative approach to goal programming is Tchebychev goal programming, which is quite similar in formulation to Archimedean yet focuses on minimizing the maximum deviation (or the worst-case scenario) from the desired goals. The idea is to make the largest deviation from any goal as small as possible, as apposed to Archimidean which seeks to minimize a weighted sum of deviations from the goals. We formulate as follows (utilizing the same notation as prior with the same arbitrage specifications as with Archimedean GP):

Minimise: Δ
Subject to:

$$\begin{aligned}
&\Delta \geq w_1 d_1^- \\
&\Delta \geq w_2 d_2^+ \\
&\Delta \geq w_3 d_3^+ \\
&z_1^* - d_1^+ + d_1^- = \$g_1 \\
&z_2^* - d_2^+ + d_2^- = g_2 \\
&z_3^* - d_3^+ + d_3^- = g_3 \\
&x_{i,t} \leq L_{i,t} \;\; \forall i, t \\
&x_{1,t} \leq M(1 - z_{2,t}) \;\; \forall t \\
&x_{2,t} \leq M(1 - z_{1,t}) \;\; \forall t \\
&z_{1,t} + z_{2,t} = 1 \;\; \forall t \\
&x_{i,t}, z_{i,t}, d_k \geq 0 \;\; \forall i, t, k
\end{aligned}$$

Where our solution vector $\mathbf{x}$ is as such:

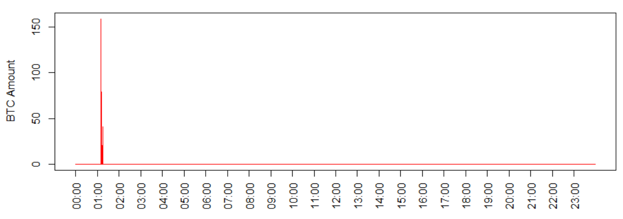
$$\mathbf{x} = \left[x_{1,1}, x_{1,2}, \dots x_{1,1440}, x_{2,1}, x_{2,2}, \dots x_{2,1440}, z_{2,1}, \dots z_{2,1440}, z_{1,1}, \dots z_{1,1440}, d_1^+, d_1^-, d_2^+, d_2^-, d_3^+, d_3^-, \Delta\right]'$$

5.2.1 An Example:

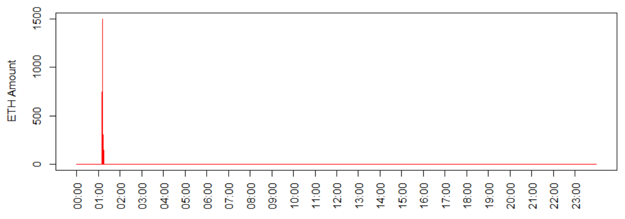
We employ the same weights utilized in our Archimedean example for comparability, hence $w_1 = 0.1, w_2 = 0.8$ and $w_3 = 0.1$. We solve the problem to obtain $d_1^- = 804.49$ - implying that the maximum profit we are able to achieve is $g_1 - \$804 = \$99,196$ (unlike with archimedean: where it was found we could actually achieve a maximum profit of g_1).

Additionally, $d_2^+ = 100.56$, implying we would need to exceed our daily BTC volume limit by ≈ 100 units (a 100% excess of our daily BTC volume restriction) in order to achieve the mentioned profit. Furthermore, we solve $d_3^+ = 804.49$, meaning we must arbitrage an 80% excess of g_3 to obtain the maximum profit. Recall in our archimedean example that only the ETH daily volume restriction g_3 needed to be exceeded, where we were able to achieve the full g_1 profit goal. Additionally, we found $\Delta = 80.45$, representing the maximum deviation across all three of the desired goals ($\Delta = w_1 d_1^- = w_2 d_2^+ = w_3 d_3^+$).

We note a similar looking solution, as our archimedean counterpart, in Figures 6a and 6b, and Figures 7a and 7b.

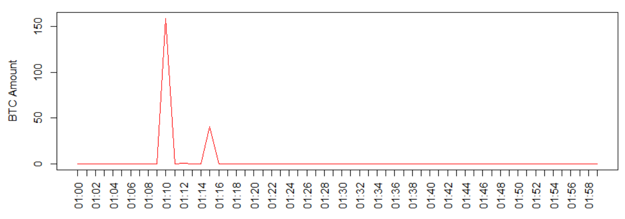


(a) Amount of BTC Arbitraged



(b) Amount of ETH Arbitraged

Figure 6: Solution for Tchebychev Example for the Entire Day


(a) Amount of BTC Arbitraged

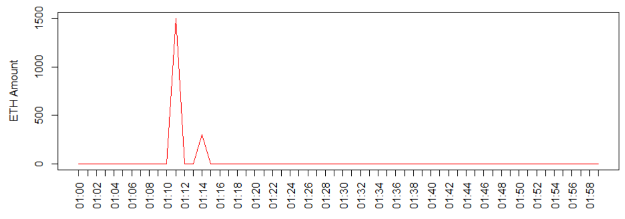

(b) Amount of ETH Arbitraged

Figure 7: Solution for Tchebychev Example from `01:00` to `02:00`

5.2.2 Sensitivity Analysis

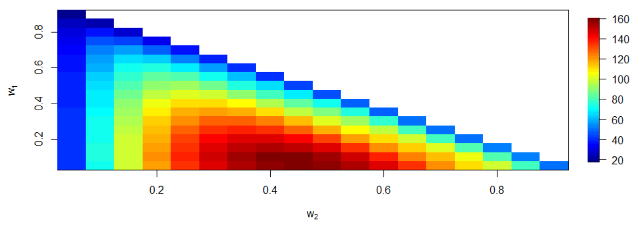
Now, unlike before, we find that varying w_1 does indeed alter the value of d_1^- - that is, the amount by which we fall short of our profit goal g_1. Hence, we need to include w_1 in our sensitivity analysis as follows: $w_1 \in [0.05, 0.9]$, where $w_2 = 1 - w_1 - w_3$ and ensuring $\sum_{k=1}^{3} w_k = 1$ as before.

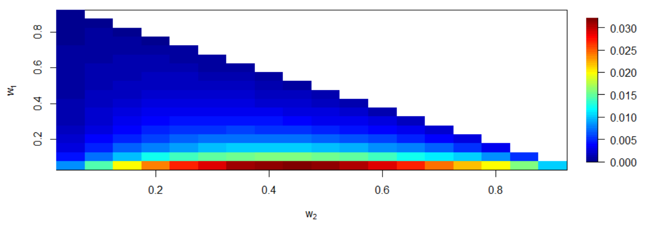
Now Figure 8a shows how Δ varies dependent on the values of w_1 and w_2 (and consequently w_3). Since Δ represents the the largest deviation from all three goals (with considering the weights assigned to each of the goals), it is apt to seek to find the lowest Δ - since this would imply that the worst-off goal (in terms of deviation from its target) is as close to its desired value as possible. Figure 8a suggests this value to be for strict w_1 and non-strict w_2 (hence non-strict w_3 as well).

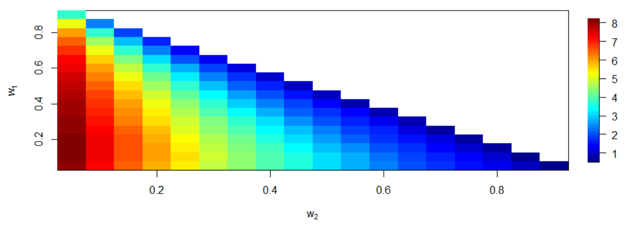
Now Figure 8b shows d_1^- as a proportion of our profit goal g_1. If one were to solely prioritize minimising the deviations from our maximum profit goal g_1, it seems fruitful to utilize non-strict w_2 weights for semi-strict to strict values of w_1 (look at the top-left region of Figure 8b).

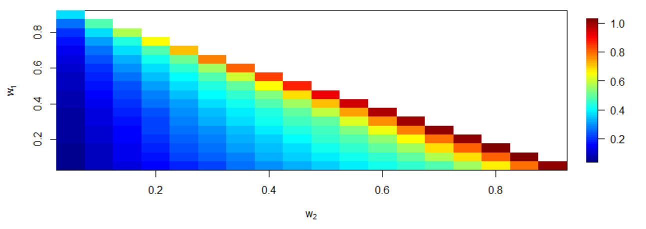
In a similar fashion, if one were to solely focus on not exceeding the daily BTC volume goal g_2, Figure 8c suggests that using strict w_2 with non-strict w_1 would be fruitful (bottom-right region of Figure 8c). On the contrary, if one wanted to solely minimise the daily ETH volume goal g_3 from being exceeded, Figure 8d suggests using non-strict w_1 with non-strict w_2 (hence strict w_3) to be apt.

Furthermore, if one were to not know which goals to necessary prioritize over the other, choosing weights which minimise Δ is a good indication of how to get the best overall solution.


(a) Δ


(b) Deviation d_1^- as a Proportion of g_1


(c) Deviation d_2^+ as a Proportion of g_2


(d) Deviation d_3^+ as a Proportion of g_3

In comparison to Archimedean goal programming, varying of the weight w_1 did actually give rise to different d_1 deviations (that is, deviations from our maximum profit goal g_1), unlike with Archimedean where our g_1

goal was obtained regardless of w_1. This may not always be fruitful (always obtaining our maximum profit goal g_1), since there might be cases where one would want to prioritize the daily volume goals rather than the maximum profit goal - that is, one may be willing to sacrifice profit to prioritize not breaching daily volume limits. In this way, we find Tchebychev goal programming to be more revealing of what to do in this case.

Additionally, Tchebychev goal programming focuses on minimising the maximum deviation across all goals, ensuring that no single goal is excessively violated. This leads to a more balanced solution where all goals are fairly satisfied, in comparison to Archimedean goal programming which merely minimises the weighted sum of deviations from the target goals.

6 Simulated Annealing

We start this section off by explaining how we configured the three core steps needed in Simulated Annealing: that is defining the initial solution, perturbing a given solution s, and evaluating said solution $f(s)$. We visualize how these three core steps 'tie-in' to the overall algorithm of simulated annealing given below. Furthermore, from Algorithm 1, we set $p(T, s', s) = \exp\left[\frac{f(s')-f(s)}{T}\right]$, since we have a maximisation problem. Hence, we are less likely to accept a worse solution s' if said worse solution is 'far' from the previous solution s.

Algorithm 1 Simulated Annealing

```
Generate initial solution $s$
Set initial temperature $T$
while termination conditions not met do
    Perturb solution to obtain new solution $s'$ from neighbourhood of $s$
    if $f(s') < f(s)$ then
        Replace $s$ with $s'$
    else
        Replace $s$ with $s'$ with probability $p(T, s', s)$
    end if
    Update temperature $T$
end while
```

Additionally, in this section, we apply the simulated annealing process to our example problem along with a sensitivity analysis which compares different parameters of two cooling schedules.

6.1 Algorithms

We showcase the dynamics of the core steps of SA in this section.

6.1.1 Initial Solution

In generating an initial solution, we know that each crypto amount $x_{i,t}$ is limited by liquidity available $L_{i,t}$. We must also ensure if $x_{i,t} > 0$ then $x_{-i,t} = 0$ - we are subject to the One-at-a-Time constraint. Hence, we generate a set of random indices $I = [1, 1440]$. Set $x_{1,t} = L_{1,t}$ for $t \subseteq I$ (only use a subset of I until $\sum_{t=1} x_{1,t} = V_1$). And we set $x_{2,t} = L_{2,t}$ for $t \notin I$ (again, only use a subset until $\sum_{t=1} x_{2,t} = V_2$). The process is given in Algorithm 2, with an example of the initial solution given in Figure 9. Noting that some indices do not require to be filled (non-filled indices given as white squares) as the daily volume constraints would have been met. Additionally, a filled index for a particular crypto (black square) must have a non-filled counterpart (white square) to satisfy the One-at-a-Time constraint.

Algorithm 2 Simulated Annealing: Initial Solution

```
while ETH daily volume is not met or BTC daily volume is not met do
    Choose whether to deal with ETH or BTC
    if ETH chosen then
        Generate an index $t \in [1, 1440]$ which has not been used before.
        Set $x_{2,t} = L_{2,t}$ - the maximum liquidity available for ETH at $t$.
        Store index $t$
    else
        Generate an index $t \in [1, 1440]$ which has not been used before.
        Set $x_{1,t} = L_{1,t}$ - the maximum liquidity available for BTC at $t$.
        Store index $t$.
    end if
end while
```

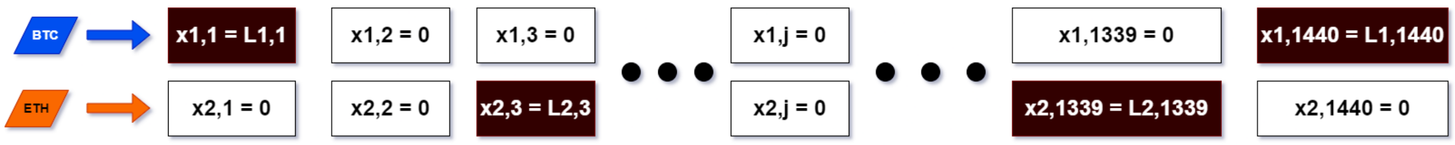


Figure 9: SA: Initial Solution Example

6.1.2 Perturb Solution

A similar dynamic is given to perturb a given solution. We start by randomly selecting n indices $t \in [1, 1440]$ for each crypto, and set $x_{i,t} = 0$ for the chosen t. We take note of the indices that are filled ie. t for which $x_{i,t} > 0$, called T. Afterwhich, we randomly select a crypto i and set $x_{i,t} = L_{i,t}$ (maximum liquidity available) by randomly choosing a $t \notin T$. We then set $x_{-i,t} = 0$ - the other crypto amount at time t. Now if $\sum_{t=1}^{1440} x_{i,t} > V_i$, then 'shave off' amount: $x_{i,t} = \%L_{i,t}$ until $\sum_{t=1}^{1440} x_{i,t} = V_i$. The process is given in Algorithm 3, with an accompanying example as to how the process works visually addressed in Figure 10.

Algorithm 3 Simulated Annealing: Perturb Solution

```
Set $x_{i,t} = 0$ for $n$ random indices $t \in [1, 1440]$ and for $i = 1, 2$.
Record all $t$ for which $x_{i,t} > 0$ called $T$
while ETH daily volume is not met or BTC daily volume is not met do
    Choose whether to deal with ETH or BTC
    if ETH chosen then
        Generate an index $t \notin T$.
        Set $x_{2,t} = L_{2,t}$ - the maximum liquidity available for ETH at $t$.
        Update $T$
        if ETH Volume constraint exceeded then
            Set $x_{2,t} = \%L_{2,t}$ - 'shave-off' maximum liquidity for ETH at $t$, until $\sum_{t=1}^{1440} x_{2,t} = V_2$
        end if
    else
        Generate an index $t \notin T$.
        Set $x_{1,t} = L_{1,t}$ - the maximum liquidity available for BTC at $t$.
        Update $T$
        if BTC Volume constraint exceeded then
            Set $x_{1,t} = \%L_{1,t}$ - 'shave-off' maximum liquidity for BTC at $t$, until $\sum_{t=1}^{1440} x_{1,t} = V_1$
        end if
    end if
end while
```

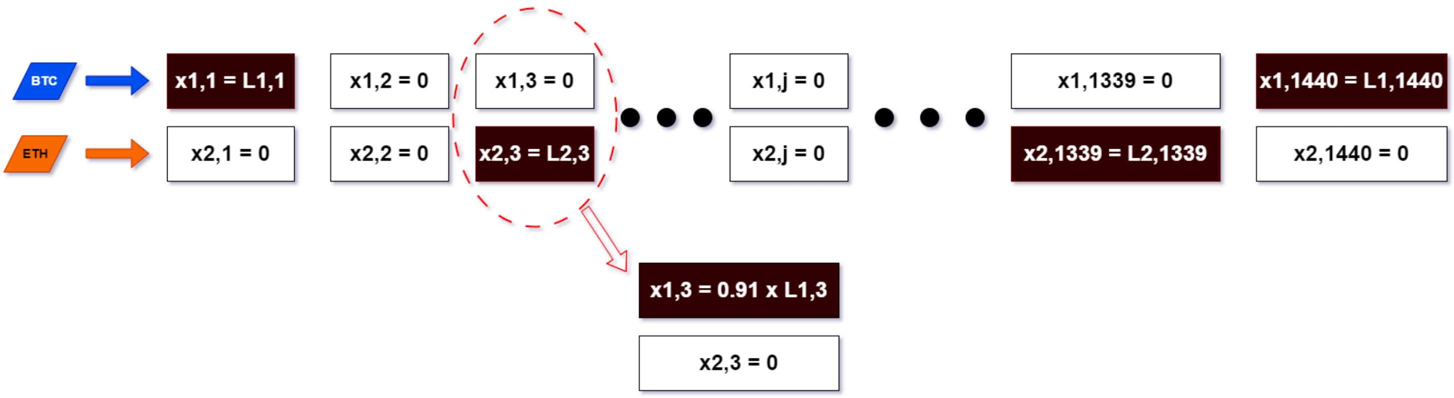


Figure 10: SA: Perturb Solution Example

6.1.3 Evaluate Solution

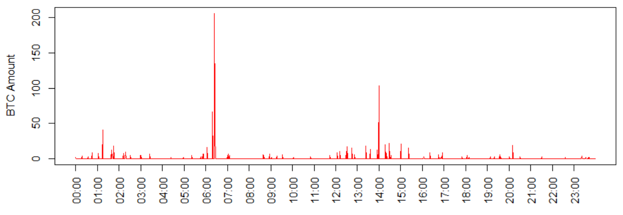
To evaluate the maximum profit, we merely multiply the absolute price vector with the given solution s to obtain $f(s)$ as $\sum_{i=1}^{2}\sum_{t=1}^{T=1440}\Delta P_{i,t}x_{i,t}$ equivalent to:

$$\begin{bmatrix}\Delta P_{1,1} & \dots & \Delta P_{1,1440} & \Delta P_{2,1} & \dots & \Delta P_{2,1440}\end{bmatrix}\begin{bmatrix}x_{1,1}\\ x_{1,2}\\ \vdots \\ x_{1,1440}\\ x_{2,1}\\ x_{2,2}\\ \vdots \\ x_{2,1440}\end{bmatrix}$$

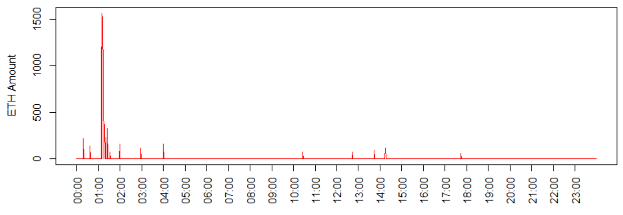
6.2 Our Example Problem:

We refer back to our example problem used in LP, but with the simulated annealing process by utilising a geometric cooling schedule, where our initial temperature $T_0 = 0.5$ and our cooling rate $\alpha = 0.99$. Additionally, we set our daily BTC volume constraint to $V_1 = 1000$ units and our daily ETH volume constraint to $V_2 = 10000$ units as before. Additionally, we set n in Algorithm 3 to be $1/4$ of the total 1440 indices.

The solution attained after 10000 iterations is given in Figures 11a and 11b, with a maximum profit achieved of $\$212,763$. The profit objective function per iteration is given in Figure 12 - we notice that even though it seems as if the maximum profit achieved may have 'plateaued' to some degree - there still seemed to be improvements made after the 7000^{th} and 9000^{th} iteration. We posit that more iterations would have resulted in us achieving a more competitive solution to the LP in Section 4.3.



(a) Amount of BTC Arbitraged



(b) Amount of ETH Arbitraged

Figure 11: Solution for Simulated Annealing Example

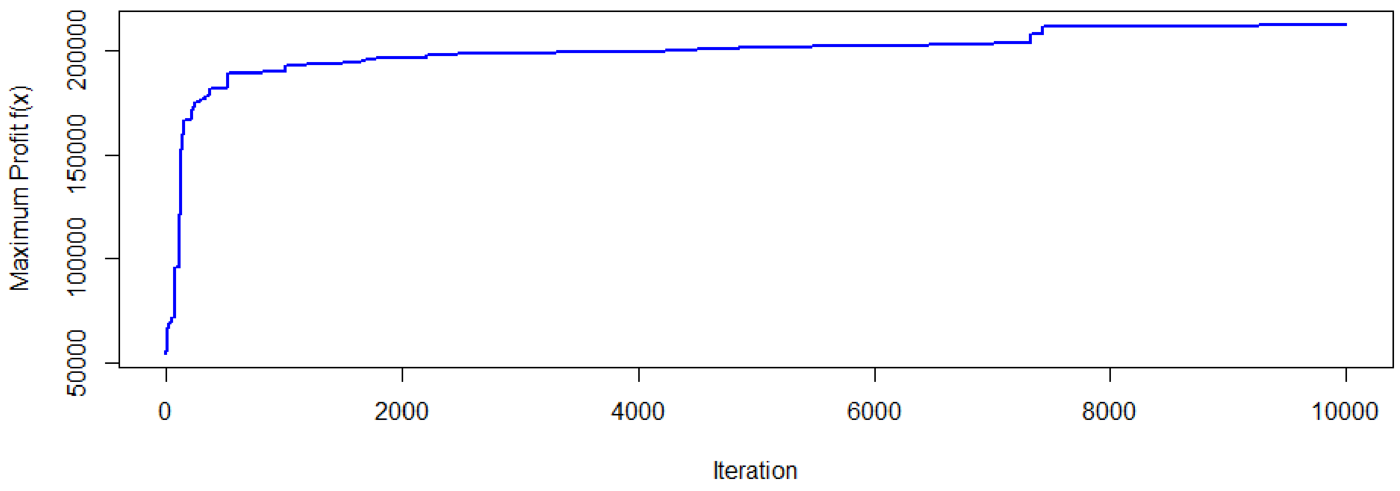


Figure 12: Profit Objective per Iteration for 10000 Iterations

6.3 Sensitivity Analysis

As displayed in Algorithm 3, we set up our Perturb Solution function as one that is dependent on n, which is the number of indices $t \in [1, 1440]$ for which we set $x_{i,t} = 0$ as an initialization process to perturb a given solution. Being such, we find it prudent to vary n to ascertain how sensitive our maximum profit objective is to n.

Additionally, as mentioned earlier, it is possible to update the cooling temperature T as in Algorithm 1 in two different ways, namely via a geometric cooling schedule or a logarithmic cooling schedule. We undergo these sensitivity analyses in this section as well. Furthermore, said sensitivity analyses will be undergone keeping in mind our example arbitrageur from LP, with the BTC daily volume restriction being $V_1 = 1000$ and the ETH daily volume restriction $V_2 = 10000$.

6.3.1 Number of n Random Indices

Figure 13 displays how our maximum profit objective varies per n (given as a proportion of the total number of indices $T = 1440$). It is clear that our profit objective reaches a peak when $\frac{n}{T} = 0.3$, implying that we should initialise the perturb process by choosing $n \approx 432$ random indices $t \in [1, T = 1440]$ and set $x_{i,t} = 0$ for $i = 1, 2$.

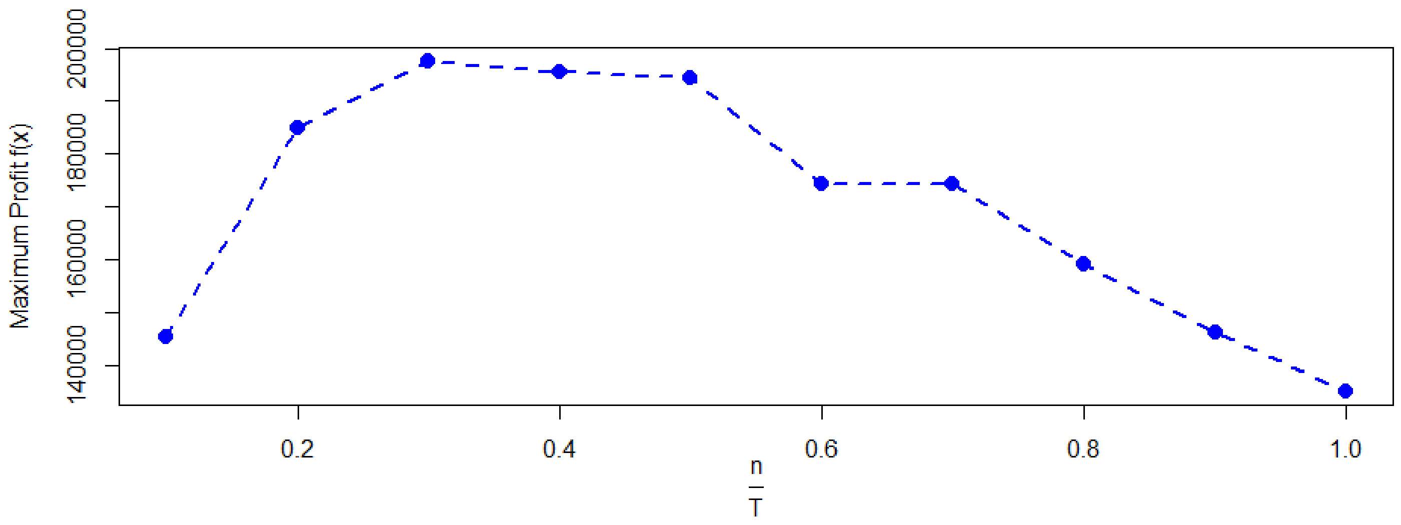
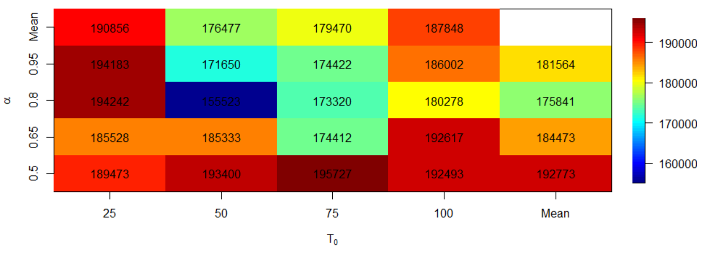


Figure 13: Profit Objective vs. $\frac{n}{T}$ for $T_0 = 0.5, \alpha = 0.99$ and for 1000 Iterations and with Geometric Cooling Schedule

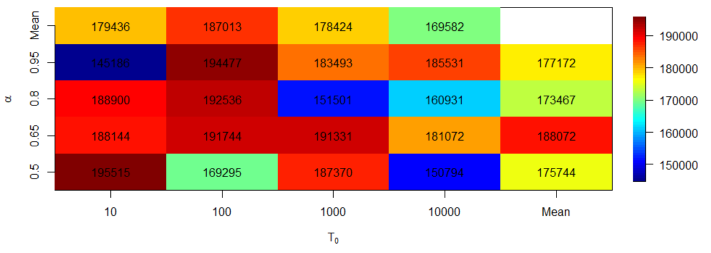
6.3.2 Geometric Cooling

A geometric cooling schedule as a means to update temperature T is given as: for the i^{th} iteration, the updated temperature is $T_i = T_0\alpha^i$ where T_0 is the initial temperature and α is the cooling factor. In this way, the cooling rate α^i is subject to an exponential decrease as there is a rapid change from exploration to exploitation. There is usually faster convergence, although this may lead to premature convergence.

Being such, we find it apt to vary our cooling factor $\alpha \in [0.5, 0.95]$ ([11] states that the typical values of α are $\in [0.8, 0.99]$) with two sets of starting temperatures $T_0^{low} = [25, 50, 75, 100]$ and $T_0^{high} = [10, 100, 1000, 10000]$ given in Figures 14a and 14b. Judging from said figures, it would seem that the simulated annealing process appears to be predominantly governed by randomness, as there does not seem to be a clearcut α or T_0 to choose which may result in overall better results. To reiterate: one is not able to infer that an increase or decrease of α nor T_0 results in an improved profit objective.



(a) Sensitivity Analysis with T_0^{low}



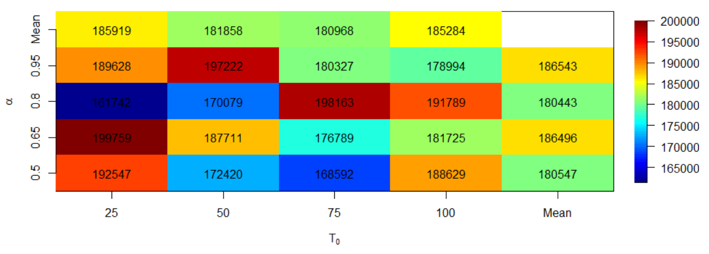
(b) Sensitivity Analysis with T_0^{high}

Figure 14: Sensitivity Analysis of Maximum Profit Objective for Geometric Cooling Schedule with $\alpha = [0.5, 0.65, 0.8, 0.95]$ with $T_0^{low} = [25, 50, 75, 100]$ and $T_0^{high} = [10, 100, 1000, 10000]$ (1000 iterations per α, T_0 combination).

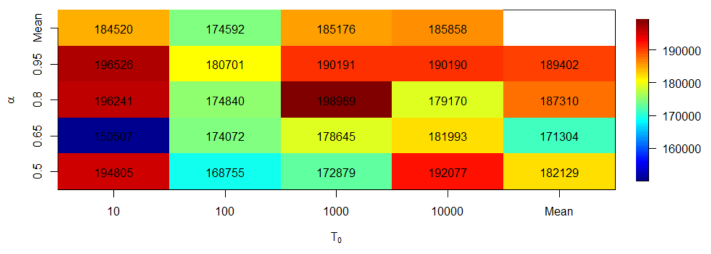
6.3.3 Logarithmic Cooling

The other cooling schedule is a logarithmic one where $T_i = \frac{T_0}{1+\alpha log(1+i)}$. In this way, the cooling rate is subject to a logarithmic decrease, and allows for a more thorough exploration of the search space, reducing the chance of getting stuck in local optima. However, it can take much longer to converge.

In a similar nature, we vary $\alpha = [0.5, 0.65, 0.8, 0.95]$ with two sets of low and high intial temperature sets $T_0^{low} = [25, 50, 75, 100]$ and $T_0^{high} = [10, 100, 1000, 10000]$ given in Figures 15a and 15b. Here again, there seems not to be any general pattern: one is not able to infer that an increase or decrease of α nor T_0 results in an improved profit objective. We notice from the α row means in Figure 15b however, that high cooling factors coupled with large T_0 tends to give improved results (that is, greater profit objectives achieved).



(a) Sensitivity Analysis with T_0^{low}



(b) Sensitivity Analysis with T_0^{high}

Figure 15: Sensitivity Analysis of Maximum Profit Objective for Logarithmic Cooling Schedule with $\alpha = [0.5, 0.65, 0.8, 0.95]$ with $T_0^{low} = [25, 50, 75, 100]$ and $T_0^{high} = [10, 100, 1000, 10000]$ (1000 iterations per α, T_0 combination).

We hypothesize that the choice of cooling factor α and initial temperature T_0 have somewhat of a negligible effect on improving the profit objective. This is likely due to the highly stochastic nature of our Perturb Solution algorithm 3, where the absence of a clear pattern can be attributed to the inherent randomness in the perturn solution process.

6.3.4 Geometric vs Logarithmic Cooling

We find it apt to now explore if any differences exist between both cooling schedules where we fix $\alpha = 0.95$ and $T_0 = 10$. Figure 16 displays the profit objective per iteration. We notice that the geomtric cooling schedule seems to converge slightly earlier than the logarithmic cooling schedule - the former converges at the $\approx 1200^{th}$ iteration while the latter at the $\approx 1500^{th}$ iteration. Now both cooling schedules seem to converge to approximately the same profit objective of $\approx \$210,000$ at the 10000^{th} iteration. The corresponding solutions at the 10000^{th} iteration are displayed in Figures 17a and 17b for the geometric cooling schedule and Figures 18a and 18b for the logarithmic cooling schedule. We note the great similarity between the two solutions, especially with regard to the identical times at which each crypto should be arbitraged (the amounts of each crypto to be arbirtaged at those particular times seem to differ however).

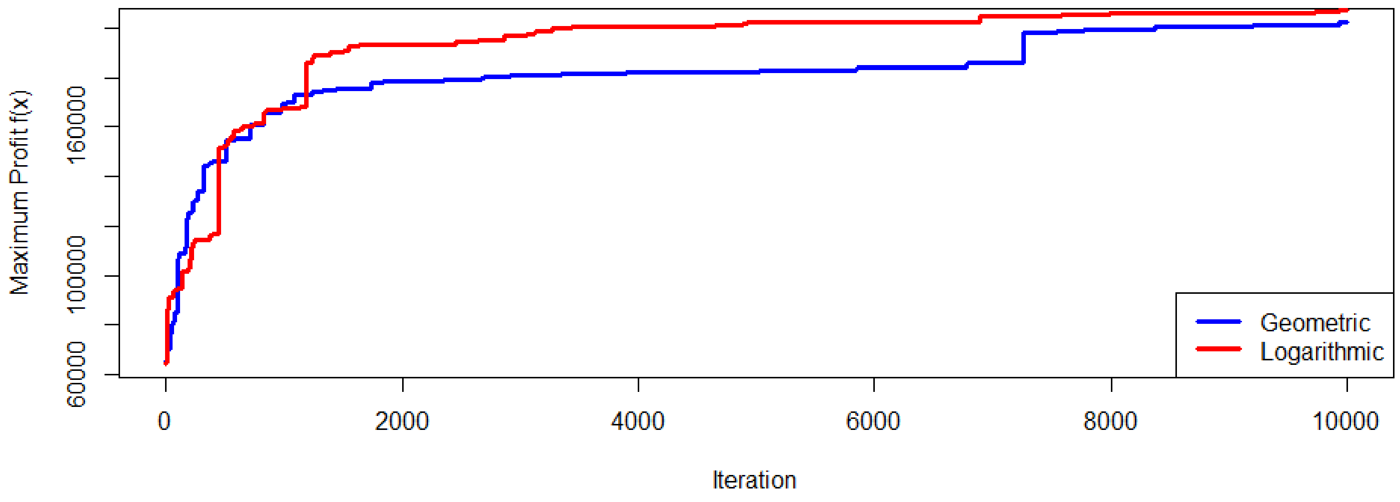
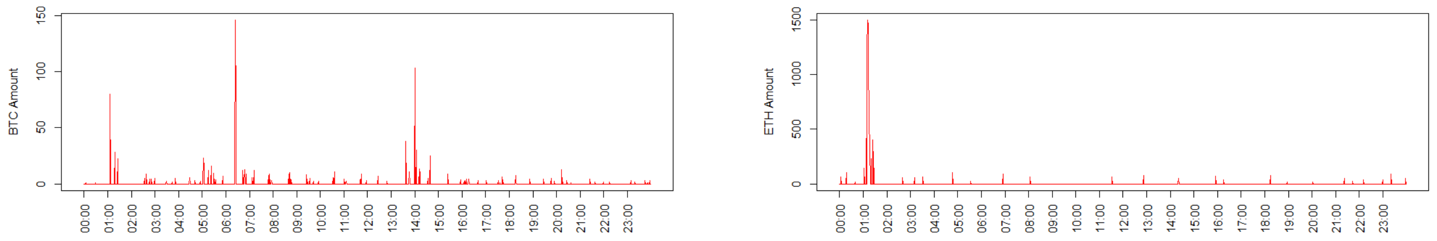
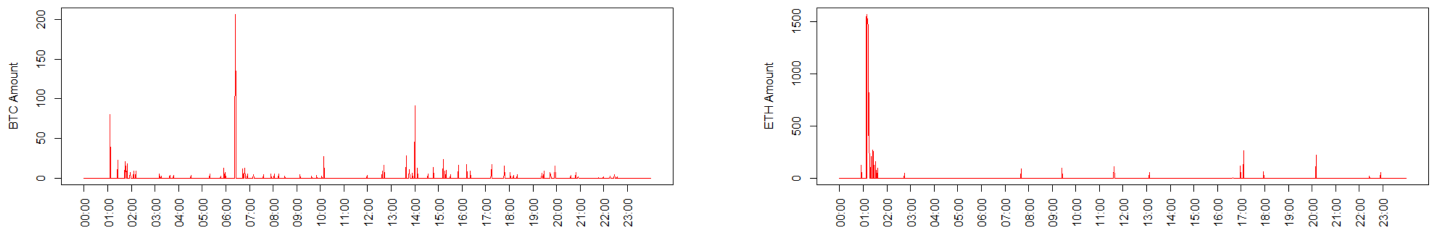


Figure 16: Profit Objective vs Iteration for Geometric and Logarithmic Cooling Schedule



(a) Amount of BTC Arbitraged (b) Amount of ETH Arbitraged

Figure 17: Solution with Geometric Cooling Schedule

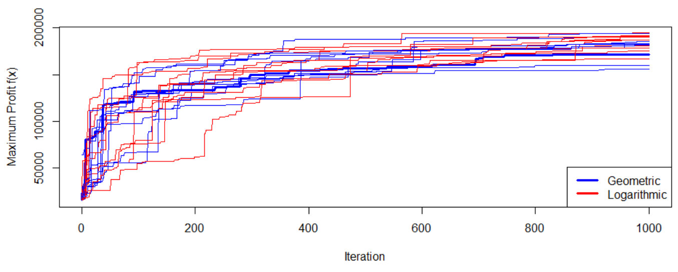


(a) Amount of BTC Arbitraged (b) Amount of ETH Arbitraged

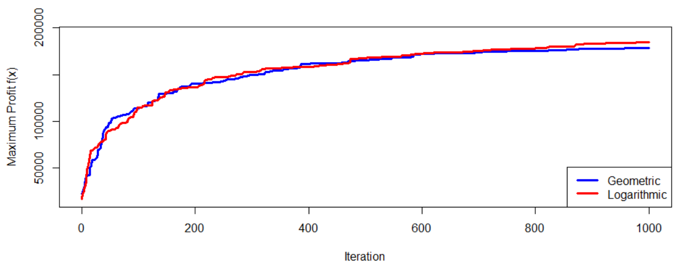
Figure 18: Solution with Logarithmic Cooling Schedule

6.3.4.1 Multiple Runs

Previously, we had run our simulated annealing algorithm once to obtain results during the comparison of the two cooling schedules. It is worthwhile to investigate whether the same results are obtained after multiple runs of the SA algorithm. Figures 19a and 19b illustrate that, when averaging the results of multiple runs at each iteration, there appears to be little difference between the SAs using different cooling schedules. Neither cooling schedule shows a clear advantage in terms of faster convergence or producing better solutions for our optimization problem. This further emphasizes the heavy stochastic nature of the SA algorithm - where it seems as if attempting to emphasize exploration over exploitation (or vice versa via utilising a specific cooling schedule) is futile since it seems to have a negligible effect on the altering optimisation outcome.



(a) Profit Objective vs Iteration for Geometric and Logarithmic Cooling Schedule (10 Runs Each with 1000 Iterations)



(b) Mean Profit Objective vs Iteration for Geometric and Logarithmic Cooling Schedule (10 Runs Each with 1000 Iterations)

Figure 19: Profit Objective vs Iteration for Geometric and Logarithmic Cooling Schedules

7 Genetic Algorithms

The core steps (or genetic operators) of the Genetic Algorithm are outlined in Algorithm 4. Now, for each genetic operator, namely: Selection, Recombination, Mutation and Replacement - there are two different algorithms outlined. Furthermore, between these two different algorithms, a sensitivity analysis is undergone to compare the two different algorithms.

Algorithm 4 Genetic Algorithm

```
  Initialize population of solutions
  Evaluate population of solutions
  while termination conditions not met do
      Select parents for reproduction
      Perform recombination to parents to produce offspring.
      Perform random mutations to offspring to avoid premature convergence.
      Select which offspring to replace parents for next generation.
      Evaluate population of solutions.
  end while
```

7.1 Genetic Operators (Steps of GA)

7.1.1 Initial Population

We employ the same initial solution algorithm as in our Simulated Annealing Section 6.1.1 to create a population of solutions. That is, a single initial solution created is considered a single member of our initial population of solutions.

7.1.2 Evaluate Population

To evaluate each solution in our population, we not only just evaluate the maximum profit generated per solution, as done in the Simulated Annealing Section 6.1.3 as $\sum_{i=1}^{2}\sum_{t=1}^{T=1440}\Delta P_{i,t}x_{i,t}$, but rather heavily penalize said solution member the further it is away from satisfying any of the two daily volume constraints V_1 and V_2. Being such, for said solution $\mathbf{x}$, the evaluation $f(\mathbf{x})$ is given as:

$$f(\mathbf{x}) = \left(\frac{\sum_{t=1}^{1440} x_{1,t}}{V_1}\right)^{p_1} \times \left(\frac{\sum_{t=1}^{1440} x_{2,t}}{V_2}\right)^{p_2} \times \sum_{i=1}^{2}\sum_{t=1}^{T=1440}\Delta P_{i,t}x_{i,t}$$

Meaning that the evaluation of a given solution is given as: (BTC daily volume of $\mathbf{x}$ as a proportion of $V_1)^{p_1}$ $\times$ (ETH daily volume of $\mathbf{x}$ as a proportion of $V_2)^{p_2}$ $\times$ (Profit Achieved of $\mathbf{x}$). Now p_1 and p_2 represent the extent to which we penalize 'bad' solutions - any real number may be used (the larger, the heavier the penalty for the respective proportion).

In this way, we allocate a low evaluation to solutions which do not closely satisfy the daily volume constraints (we know that the closer a solution is to fully satisfying a volume constraint, the more profit it will make since one would be able to arbitrage more volume).

7.1.3 Tournament Selection

We employ two methods of selection.

7.1.3.1 Informal Selection

Informal selection involves picking k members of the population at random and selecting the best evaluated amongst these. We repeat until we obtain the original population size as in Algorithm 5.

Algorithm 5 GA: Informal Selection Process

```
**Initialize** population of solutions.
**Set** selection size $k = \frac{1}{10}$ of population size.
**while** new population size does not equal the original population size **do**
    **Select** $k$ random members from the original population to form a temporary sub population.
    **Evaluate** the sub population.
    **Add** the best $\frac{k}{2}$ members of the sub population to the new population.
**end while**
```

7.1.3.2 Proportional to Fitness Selection

An alternate type of selection process is where we pick individual solutions based off a probability - this probability being proportional to said individual's fitness or evaluation. Being such, solution members with better evaluations have a higher chance of being selected to stay in the population. The algorithm is trivial so we will forego reciting it here.

7.1.4 Recombination (Crossover)

Recombination, also known as crossover, is a process which combines the genetic information of two parent solutions to generate two offspring. We note that only the daily volume constraints may be broken during this step - although we remedy this in our algorithm. Furthermore, we undergo two methods of crossover.

7.1.4.1 Uniform Crossover

We denote the pair of parent solutions as: $\mathbf{x_1}^1$ and $\mathbf{x_2}^1$ (representing the BTC and ETH 'part' of the first parent respectively), and $\mathbf{x_1}^2$ and $\mathbf{x_2}^2$ (representing the BTC and ETH 'part' of the second parent respectively). Similarly, we denote the offspings as $\mathbf{x_1^*}^1$ and $\mathbf{x_2^*}^1$ (representing the BTC and ETH 'part' of the first offsping respectively), and $\mathbf{x_1^*}^2$ and $\mathbf{x_2^*}^2$ (representing the BTC and ETH 'part' of the second offspring respectively). Now for each $t \in [1, 1440]$, we randomly choose which parent $j = 1, 2$ is used to create the subsequent offsprings as such: ${x_{1,t}^*}^1 = x_{1,t}^j$, ${x_{2,t}^*}^1 = x_{2,t}^j$ hence ${x_{1,t}^*}^2 = x_{1,t}^{-j}$, ${x_{2,t}^*}^2 = x_{2,t}^{-j}$. Figure 20 illustrates an example of what the two offspring might look like, reproduced from the two parent solutions. Furthermore, Algorithm 6 details the uniform crossover. Note how the algorithm also ensures the daily volume constraint for each crypto is not exceeded - where we 'shave-off' the amount for the corresponding index t if exceeded.

Algorithm 6 GA: Uniform Crossover

```
Shuffle and Pair the population into parent pairs $\mathbf{x_1}^1$, $\mathbf{x_2}^1$ and $\mathbf{x_1}^2$, $\mathbf{x_2}^2$
for each parent pair do
    for each index $t \in [1, 1440]$ do
        if random number $\leq 0.5$ then
            Assign ${x^*_{1,t}}^1 = x^1_{1,t}$, ${x^*_{2,t}}^1 = x^1_{2,t}$
            Assign ${x^*_{1,t}}^2 = x^2_{1,t}$, ${x^*_{2,t}}^2 = x^2_{2,t}$
        else
            Assign ${x^*_{1,t}}^1 = x^2_{1,t}$, ${x^*_{2,t}}^1 = x^2_{2,t}$
            Assign ${x^*_{1,t}}^2 = x^1_{1,t}$, ${x^*_{2,t}}^2 = x^1_{2,t}$
        end if
        if $\forall i, j$: $\sum_{t=1}^{t} {x^*_{i,t}}^j > V_i$ and ${x^*_{i,t}}^j > 0$ then
            Assign ${x^*_{i,t}}^j = \%x^j_{i,t} = \%L_{i,t}$
        end if
    end for
end for
```

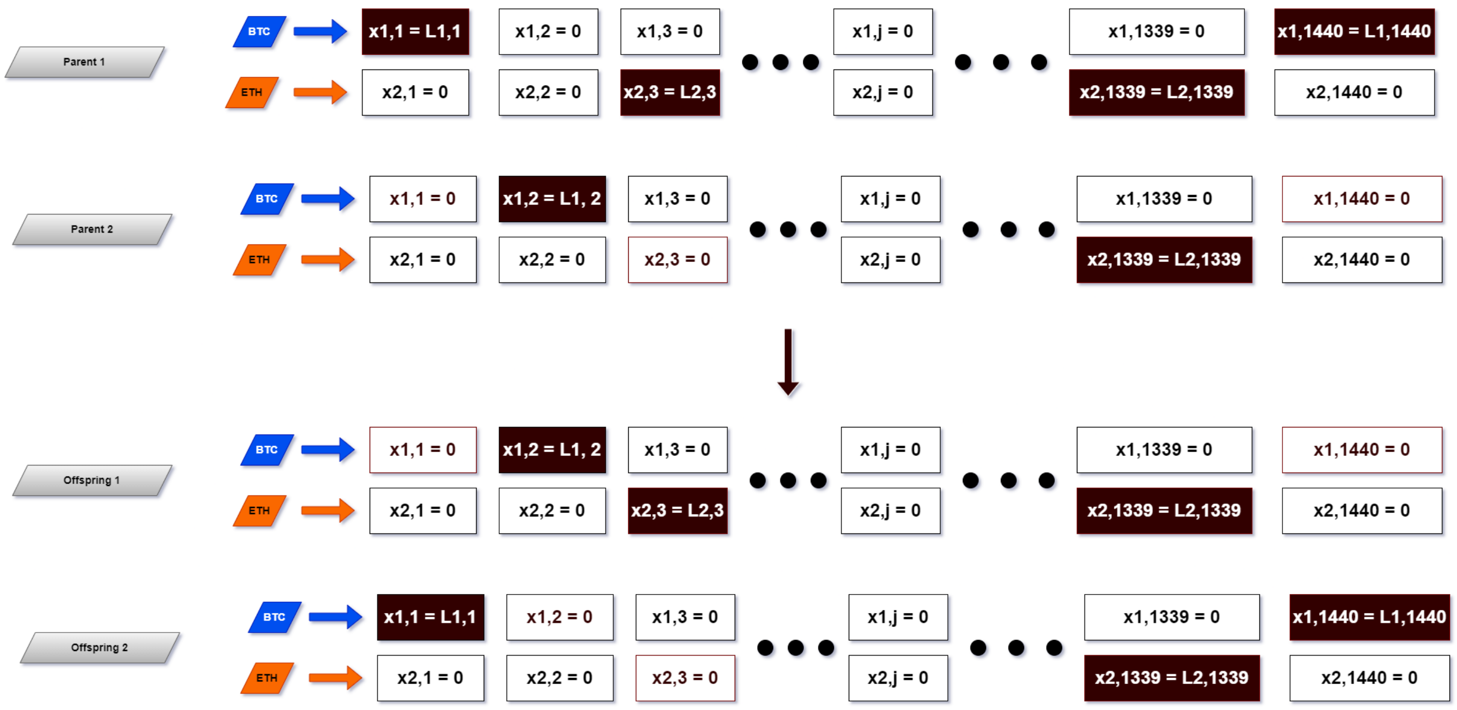


Figure 20: GA: Uniform Crossover Example

7.1.4.2 n-Point Crossover

A similar approach to uniform crossover, is n-point crossover. Instead of randomly choosing, for any $t \in [1, 1440]$, which parent passes on the corresponding 'gene' to a particular offspring - parents now pass on 'blocks' of their genes to offspring. The number and nature of these sets is determined by the number of point crossovers there are. Algorithm 7 and the example in Figure 21 aids in the understanding of this concept.

Algorithm 7 GA: n-Point Crossover

```
**Shuffle** and **Pair** the population into parent pairs $\mathbf{x_1}^1$, $\mathbf{x_2}^1$ and $\mathbf{x_1}^2$, $\mathbf{x_2}^2$
**Generate** crossover points such that $M = [1, m_1, m_2, m_3, \ldots m_n, 1440]$ where $n$ represents the number of cross-over points.
for each parent pair do
    for $j$ in $1 : (n+1)$ do
        if $j$ odd then
            Assign ${x^*_{1,M[j:j+1]}}^1 = x^1_{1,M[j:j+1]}$, ${x^*_{2,M[j:j+1]}}^1 = x^1_{2,M[j:j+1]}$
            Assign ${x^*_{1,M[j:j+1]}}^2 = x^2_{1,M[j:j+1]}$, ${x^*_{2,M[j:j+1]}}^2 = x^2_{2,M[j:j+1]}$
        else
            Assign ${x^*_{1,M[j:j+1]}}^1 = x^2_{1,M[j:j+1]}$, ${x^*_{2,M[j:j+1]}}^1 = x^2_{2,M[j:j+1]}$
            Assign ${x^*_{1,M[j:j+1]}}^2 = x^1_{1,M[j:j+1]}$, ${x^*_{2,M[j:j+1]}}^2 = x^1_{2,M[j:j+1]}$
        end if
        if $\sum_{t=1}^{M[j+1]} {x^*_{i,t}}^j > V_i$ then
            for $t \in M[j : j+1]$ such that ${x^*_{i,t}}^j > 0$ do
                Assign ${x^*_{i,t}}^j = \%x^j_{i,t} = \%L_{i,t}$
            end for
        end if
    end for
end for
```

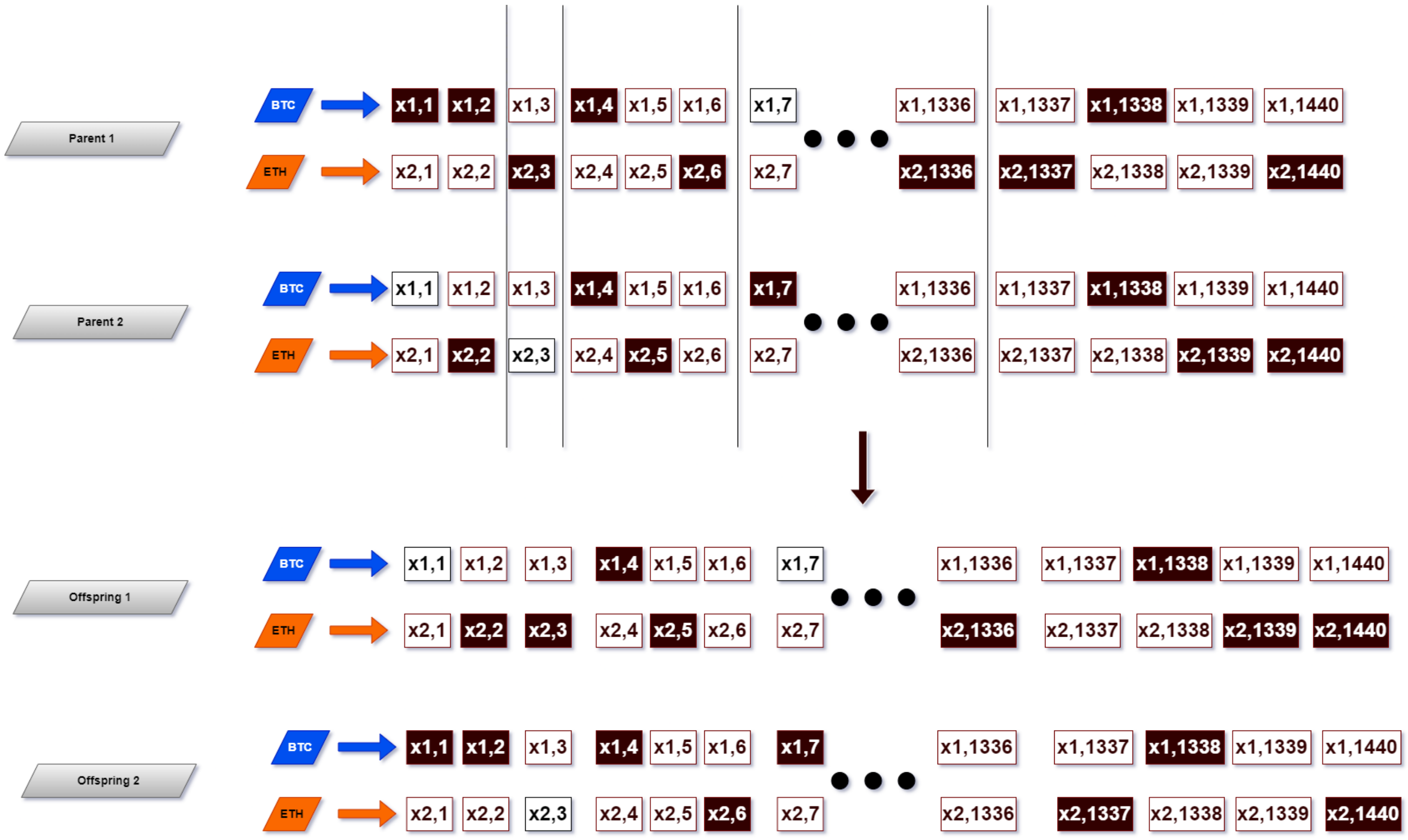

Parent 1
BTC
x1,1 x1,2 x1,3 x1,4 x1,5 x1,6 x1,7 x1,1336 x1,1337 x1,1338 x1,1339 x1,1440
ETH
x2,1 x2,2 x2,3 x2,4 x2,5 x2,6 x2,7 x2,1336 x2,1337 x2,1338 x2,1339 x2,1440
Parent 2
Offspring 1
Offspring 2

Figure 21: GA: n-Point Crossover Example

7.1.5 Mutation

Now an essential aspect for maintaining genetic diversity within the population, preventing premature convergence to suboptimal solutions, and exploring the search space more thoroughly, is the mutation step.

We note that only the liquidity constraints may be broken during this step - although we remedy this in our algorithm. Furthermore, we explore two different mutation methods.

7.1.5.1 Scramble Mutation

This process is merely picking a subset of genes of an offspring and randomly rearranging the positions. Figure 22 illustrates this for a single offspring - note how we ensure the liquidity constraints are met. Furthermore, we denote $\mathbf{x_1}^i$, $\mathbf{x_2}^i$ as the BTC and ETH part of an i^{th} offspring solution in this section (we exclude $*$ to denote offspring as we no longer need to distringuish between parent and offspring). The process is described in Algorithm 8.

Algorithm 8 GA: Scramble Mutation

```
for each $i^{th}$ offspring ($\mathbf{x_1}^i$, $\mathbf{x_2}^i$) do
    if $i^{th}$ offspring randomly selected to mutate then
        Select two random indices $m_1, m_2$ (ordered) such that $M = [m_1, t_{m_1+1}, t_{m_1+2}, \dots, t_{m_2-1}, m_2]$ of $n$ length.
        Shuffle $M$ to obtain $M^* = [m_1^*, m_2^*, \dots, m_n^*]$.
        Assign $x_{1,M^*} = x_{1,M}$.
        Assign $x_{2,M^*} = x_{2,M}$.
        if $\forall i : \sum_{t=1}^{1440} x_{i,t} > V_i$ then
            for $t \in M^*$ such that $x_{i,t} > L_{i,t}$ do
                Assign $x_{i,t} = L_{i,t}$
            end for
        end if
    end if
end for
```

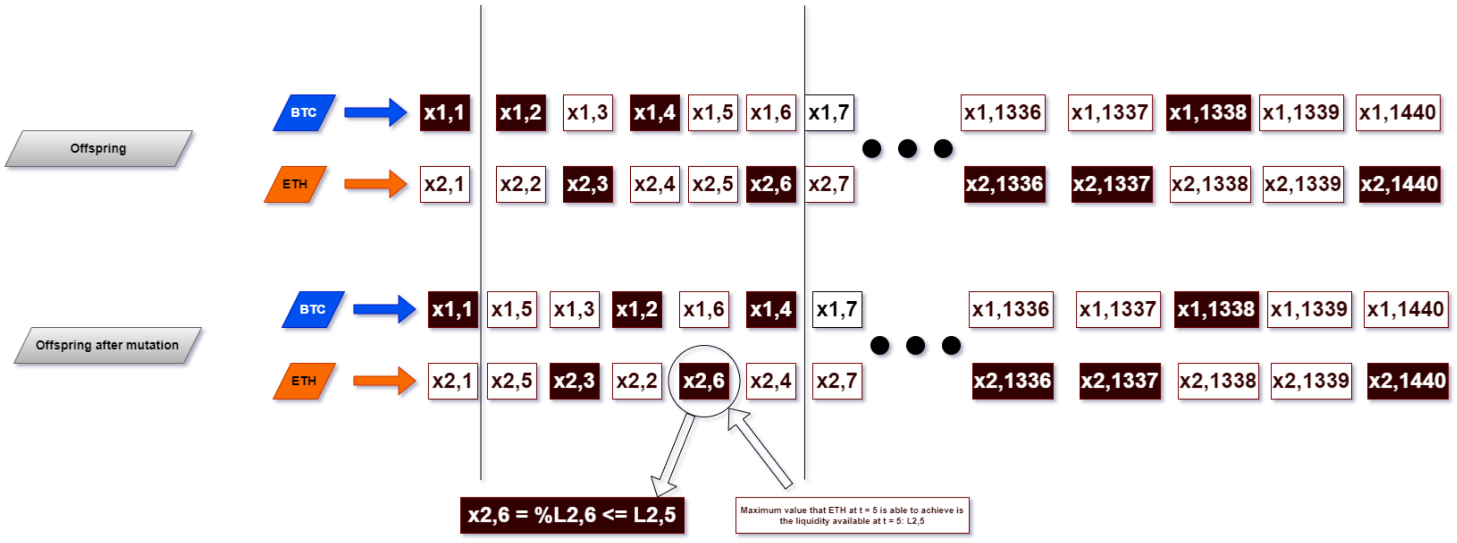


Figure 22: GA: Scramble Mutation Example

7.1.5.2 Swap Mutation

Another mutation approach, rather similar to scramble mutation, is swap mutation. We merely just pick two genes at random (of an offspring) and swap them. Hence, the process remains the same as Algorithm 8 except $M = [m_1, m_2]$ and $M^* = [m_2, m_1]$.

7.1.6 Replacement

Replacement refers to the process of deciding which individuals from the current population will be replaced by the new individuals (offspring) generated through crossover and mutation. This step is crucial in determining how the population evolves over generations. We present two methods of replacement.

7.1.6.1 Generational

Generation replacement is such that all individuals survive for only a single generation - that is, all parents are replaced by the offspring.

7.1.6.2 Steady-State

Steady-state replacement occurs when only a few individuals (often the worst-performing ones) in the population are replaced by the new offspring in each generation. The rest of the population remains unchanged. We undergo 'Elitist' steady-state replacement, where the new population is formed by selecting the individuals with the highest fitness from a combined pool of parents and offspring. The process is given formally in Algorithm 9. Now, this method ensures that only the fittest individuals survive, promoting gradual improvement of the population's overall fitness. Yet, by always selecting the top individuals based on fitness, the population can quickly become dominated by similar, highly fit individuals. This reduces genetic diversity, which could lead to premature convergence where the algorithm could get stuck in a local optimum and fail to explore other potentially better solutions in the search space.

Algorithm 9 GA: Fitness-Based Steady-State Replacement

```
Combine parents and offspring into a combined population.
Evaluate the fitness of the combined population.
Sort the combined population based on fitness in descending order.
Select the top individuals from the sorted combined population to form the new population, retaining
the original population size.
```

7.2 Sensitivity Analysis

We conduct sensitivity analyses between each of the two genetic operators discussed in the previous section. A population size sensitivity analyses is undergone at the end.

Furthermore, said sensitivity analyses will be undergone keeping in mind our example arbitrageur used throughout the study, with the BTC daily volume restriction being $V_1 = 1000$ and the ETH daily volume restriction $V_2 = 10000$.

7.2.1 Tournament Selection

We compare the two selection types previously mentioned in Figure 23 (both GAs use uniform crossover, scramble mutation and steady-state replacement). We notice that the genetic algorithm using informal selection results in faster convergence (at $\approx 20^{th}$ iteration) rather than selection proportional to fitness (at $\approx 40^{th}$ iteration). Both GAs eventually converge to similar profit objectives of $\approx \$223,000$ with the similar solutions after genetic convergence (complete genetic homogeneity: all individuals in the last population are the same) displayed in Figures 24a and 24b for selection proportional to fitness and Figures 25a and 25b for informal selection. This genetic homogeneity is why we see the highest-evaluated individual being equal to the mean evaluation of the population at convergence (the dotted lines converge to the dotted points in Figure 23).

Since informal tournament selection works by selecting a group of individuals and then choosing the best one out of that group - it tends to create higher selection pressure because individuals with the best fitness in the tournament are favored more aggressively. In contrast, fitness-proportional selection (like roulette wheel selection) selects individuals probabilistically, meaning that even individuals with relatively low fitness still have a chance of being selected. This results in lower selection pressure and slower convergence.

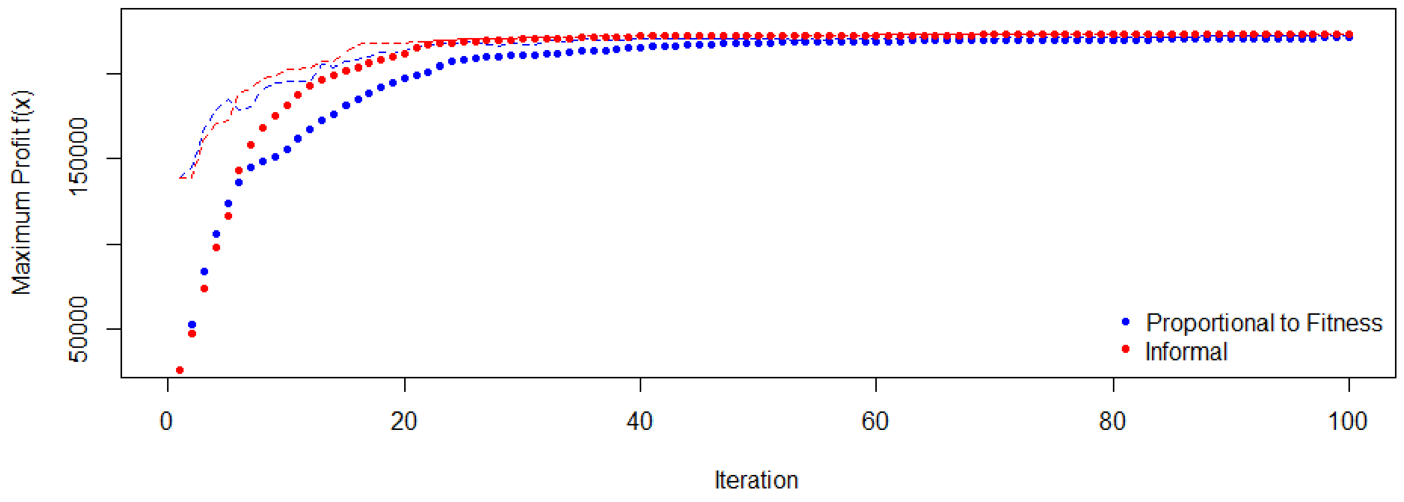
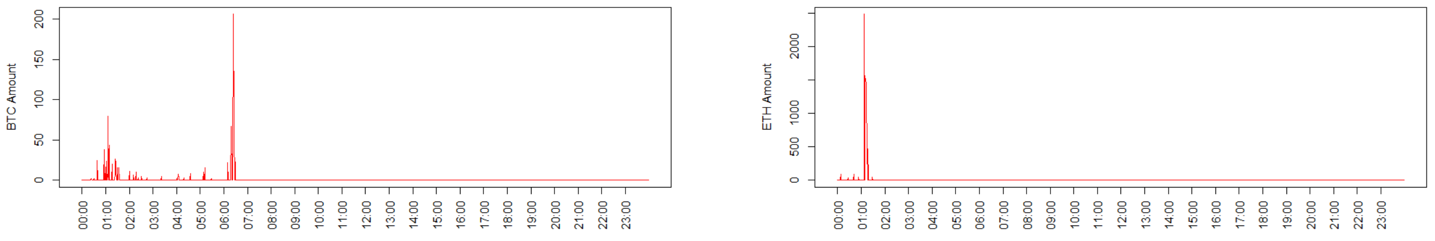


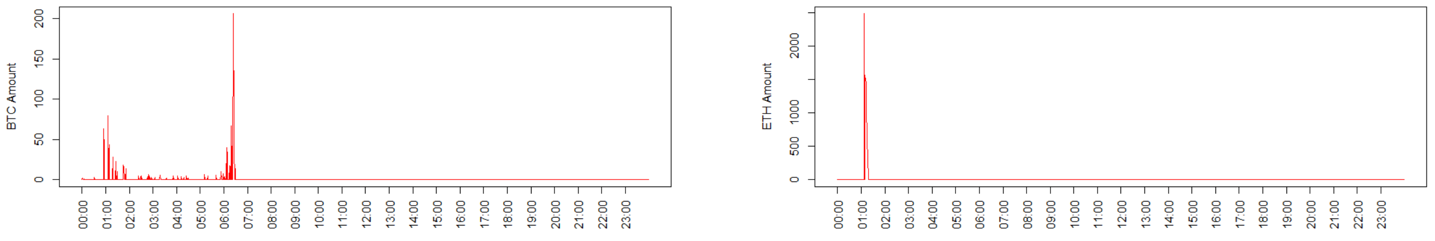
Figure 23: Sensitivity Analysis between Selection Proportional to Fitness and Informal Selection (Maximums of Population are Dotted Lines, with Corresponding Means being Circular Points)



(a) Amount of BTC Arbitraged

(b) Amount of ETH Arbitraged

Figure 24: Solution of GA at Convergence with Selection Proportional to Fitness



(a) Amount of BTC Arbitraged

(b) Amount of ETH Arbitraged

Figure 25: Solution of GA at Convergence with Informal Selection

7.2.2 Recombination

7.2.2.1 Comparing Uniform with n-Point Crossover

Similarly, we compare the two recombination operators, namely uniform and n-point crossover, using $n = 250$ in Figure 26 (both GAs utilize selection proportional to fitness, scramble mutation and steady-state replacement). We notice that there is not much difference between the two recombination methods in terms of rate of convergence, nor the maximum profit objective reached at convergence.

In theory, uniform crossover is supposed to result in more random mixing of genetic material since it operates independently at each gene position, promoting greater diversity. Whereas n-point crossover tends to keep larger segments of genetic material intact, so offspring may inherit more cohesive blocks of genes from each parent. Uniform crossover generally leads to slower convergence as the gene mixing is more thorough, whereas n-point crossover can lead to faster convergence because entire blocks of genes are transferred.

Yet seeing as we utilized $n = 250$ points, one can argue that this is the reason why the two methods seem to not be much different from each other (we undergo a sensitivity analysis of our profit objective against varying n to study this further).

Both GAs eventually converge to similar profit objectives of $\approx \$222,600$ with similar solutions after genetic convergence displayed in Figures 27a and 27b for uniform crossover and Figures 28a and 28b for n-point crossover.

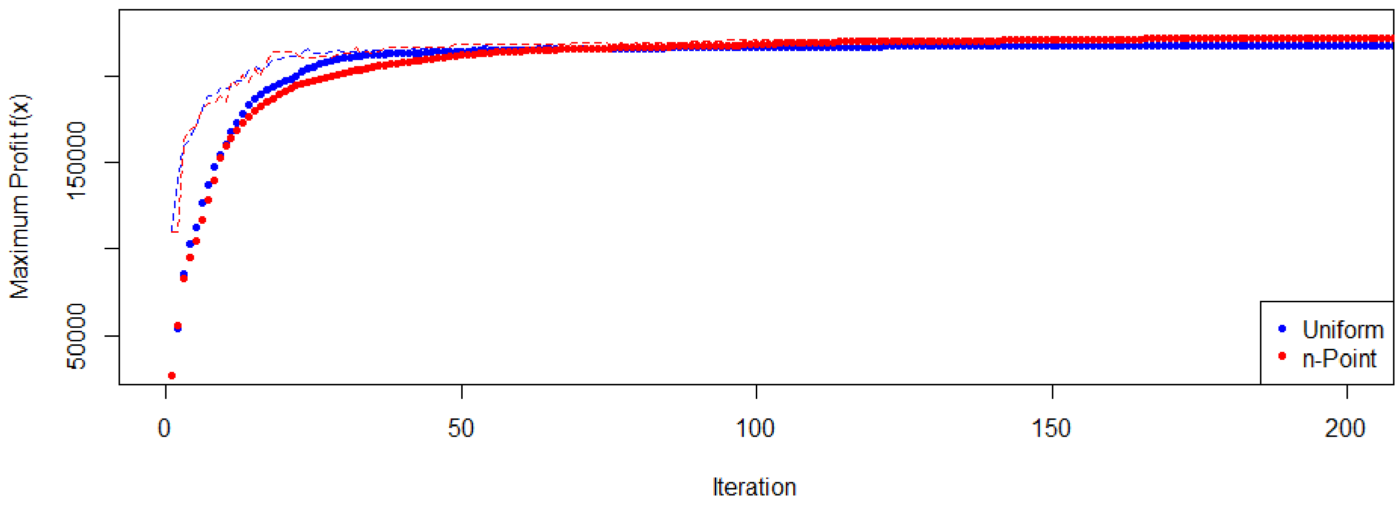
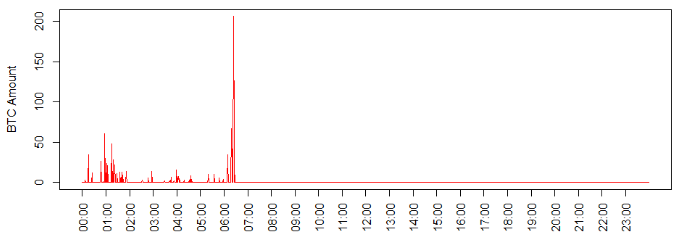
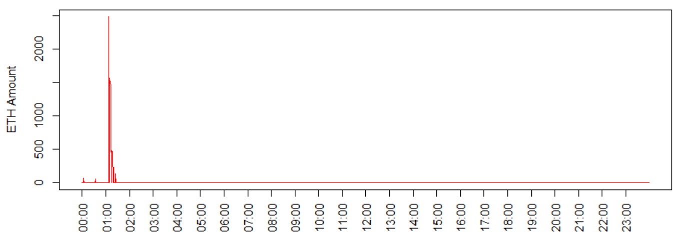


Figure 26: Sensitivity Analysis between Uniform and n-Point Crossover, with $n = 250$ (Maximums of Population are Dotted Lines, with Corresponding Means being Circular Points)

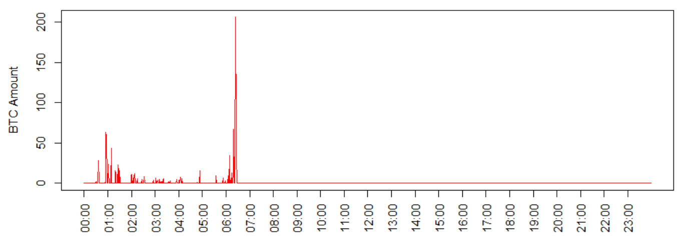


(a) Amount of BTC Arbitraged

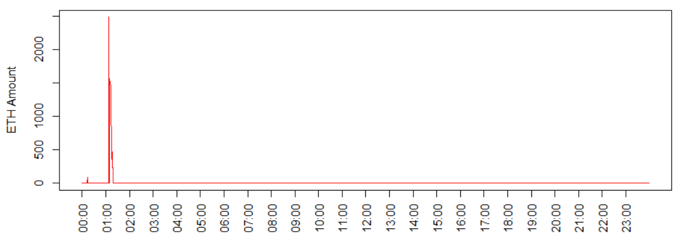


(b) Amount of ETH Arbitraged

Figure 27: Solution of GA at Convergence with Uniform Crossover



(a) Amount of BTC Arbitraged

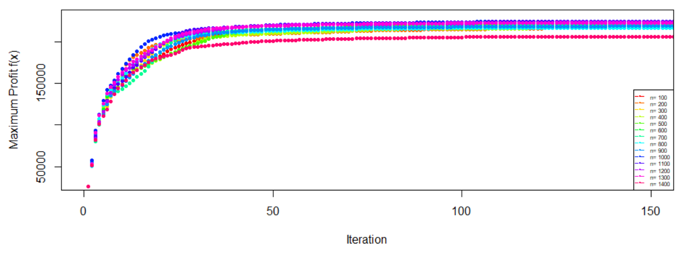


(b) Amount of ETH Arbitraged

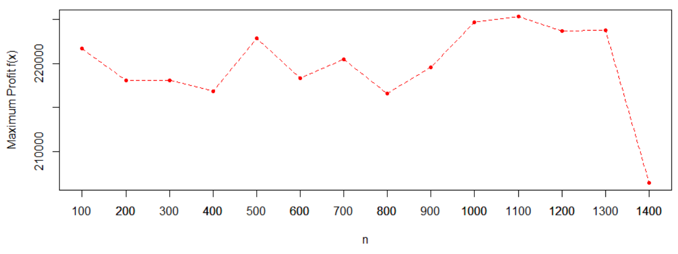
Figure 28: Solution of GA at Convergence with n-Point Crossover

7.2.2.2 Varying n in n-Point Crossover

This section investigates the influence of changing the number of points n where the two parent individuals' genes are split for n-point crossover. Figure 29a and Figure 29b shows differences in both the convergence rates and the final objectives reached at convergence respectively. There is no clear indication of an optimal value for n, as the solutions appear to be heavily influenced by the stochastic nature of n-point crossover. While one could argue that the ideal value of n is $n \in [1000, 1300]$ based on Figure 29b, we propose that this behavior is largely random. Furthermore, we can say that when n is large it approximates uniform crossover.



(a) Sensitivity Analysis of Varying n in n-Point Crossover



(b) Maximum Profit Achieved at Convergence of n-Point GA for Varying n

Figure 29: Sensitivity Analyses of n-Point Crossover by Varying n

7.2.3 Mutation

7.2.3.1 Comparing Scramble with Swap Mutation

We compare the two mutation operators: scramble and swap mutation, both with a mutation rate of 0.05 in Figure 30 (both GAs use selection proportional to fitness, uniform crossover and steady-state replacement). We notice that both GAs look very similar in nature, although slightly faster convergence occurs with scramble mutation. Additionally, both GAs converge to a final profit objective of $\approx \$221,700$, with similar solutions given in Figures 31a and 31b for scramble mutation and Figures 32a and 32b for swap mutation.

Now in theory, scramble mutation introduces a larger variation in the chromosome, whereas swap mutation introduces a more localized, subtle variation. Scramble mutation promotes more exploration (diverse solutions) because it modifies multiple genes at once. This might slow down convergence in the early stages but can help avoid getting trapped in poor local optima - yet not applicable in our problem.

Swap mutation favors exploitation (refining good solutions) due to its more subtle changes, which might lead to faster convergence if the algorithm is already close to an optimal solution but can be slower or less effective early in the search when broader exploration is needed. Considering our optimisation problem however, it seems there is not much difference between the solutions produced by scramble and swap mutation.

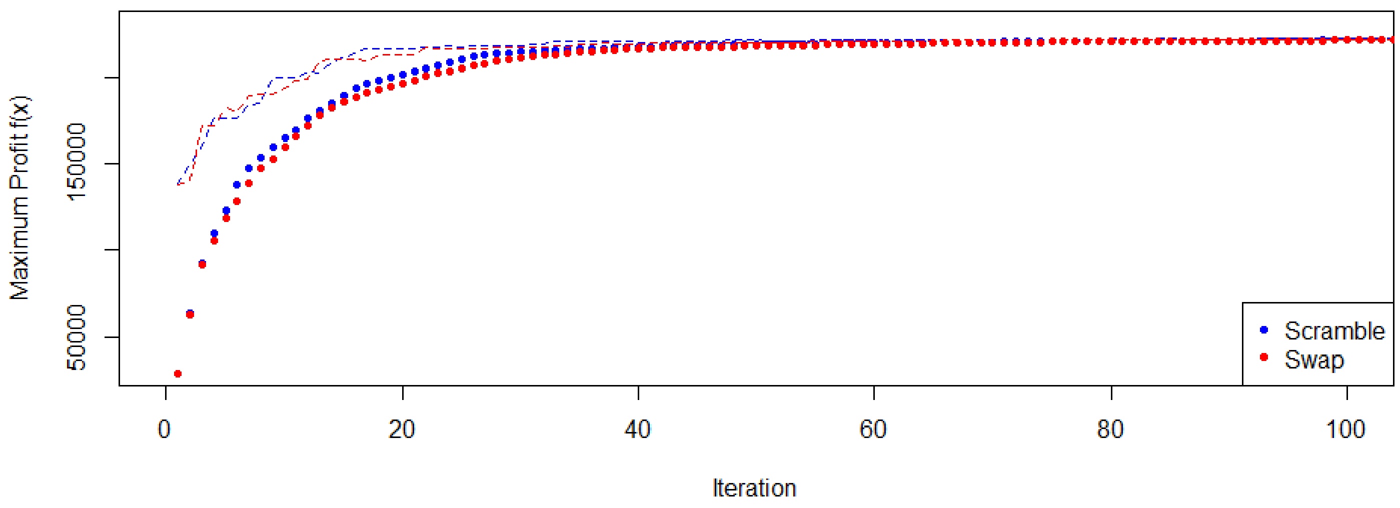
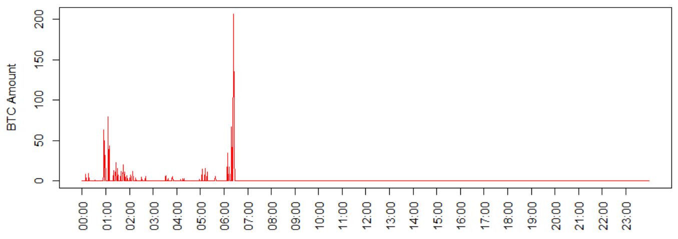
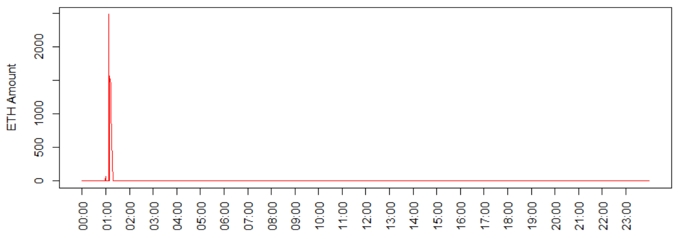


Figure 30: Sensitivity Analysis between Scramble and Swap Mutation, with 0.05 Mutation Rate (Maximums of Population are Dotted Lines, with Corresponding Means being Circular Points)

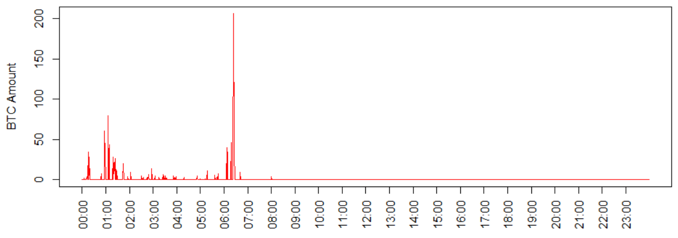


(a) Amount of BTC Arbitraged

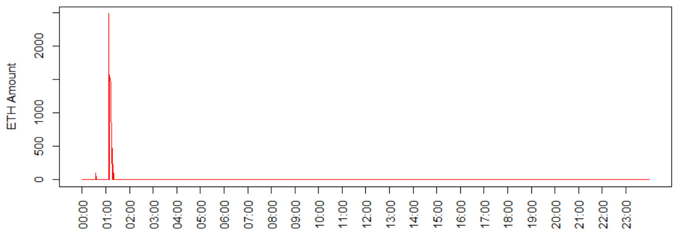


(b) Amount of ETH Arbitraged

Figure 31: Solution of GA at Convergence with Scramble Mutation



(a) Amount of BTC Arbitraged



(b) Amount of ETH Arbitraged

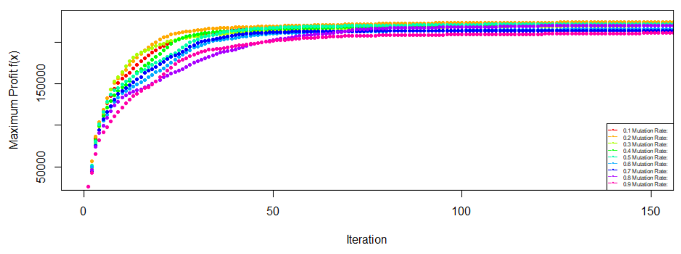
Figure 32: Solution of GA at Convergence with Swap Mutation

7.2.3.2 Varying Mutation Rate in Scramble Mutation

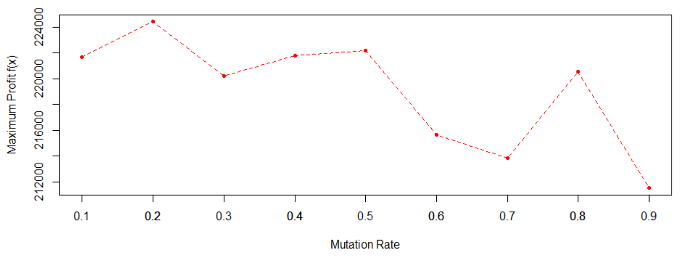
As stated earlier, it is worthwile to investigate the effect of the mutation rate on the genetic algorithm (we solely focus on utilising scramble mutation here, and posit that similar behaviour would apply for swap mutation). Now Figure 33a shows that a higher mutation rate actually gives rise to slower convergence, whereas Figure 33b suggests that higher mutation rates give rise to poorer convergence.

Now we know that a higher mutation rate encourages more exploration of the search space by introducing more random changes in the offspring, and thus preventing one from getting stuck in local optima. However, while a high mutation rate may prevent premature convergence to suboptimal solutions, it can also slow down convergence. This is because the randomness introduced by mutation may disrupt well-adapted solutions.

One the contrary, if the mutation rate is too low, the population may lose diversity too quickly, leading to premature convergence - although this seems not to be an issue with our optimisation problem. We posit that the poor convergence at high mutation rates is due to the excessive randomness: that is, the GA started to behave more like a random search - losing the benefits of the evolutionary process, and this resulted in erratic behavior and poorer convergence.



(a) Sensitivity Analysis of Varying Mutation Rate in Scramble Mutation



(b) Maximum Profit Achieved at Convergence of Scramble Mutation GA for Varying Mutation Rate

Figure 33: Sensitivity Analyses of Scramble Mutation by Varying Mutation Rate

7.2.4 Replacement

When comparing generational and steady-state replacement in Figure 34, we notice how the GA with generational replacement significantly under-performs the GA using steady-state replacement: the former converges to $\approx \$147,400$ and the latter at $\approx \$223,800$, with significantly different solutions given in Figures 35a and 35b for the GA with generational replacement and Figures 36a and 36b for the GA with stead-state replacement.

Additionally, since steady-state replacement makes incremental changes to the population, it can more quickly converge to a solution - which what oocurs in our problem seen in Figure 34. Now there is a risk of premature convergence: that is, if the population lacks diversity, steady-state replacement can cause the algorithm to converge to local optima quickly, making it harder to escape poor-quality solutions. This may also lead to less exploration. This does not seem to occur in our optimization problem however - fast convergence seems to benefit us in this case when compared to generational replacement.

Now contrarily, with generational replacement, there is an increased exploration phase and a slower convergence seeing as the best individuals in one generation may be lost in the next generation - in theory the algorithm generally rediscovers these good solutions repeatedly, yet this seems not to occur in our case. We can see this in Figure 34 around the 20^{th} iteration (dotted blue line) - where individuals with the largest evaluations are being discarded in the next generation and are never rediscovered. This is the reason why the dotted blue line (representing the maximum evaluated individual of the population) decreases after this iteration to eventually converge to the mean of the population.

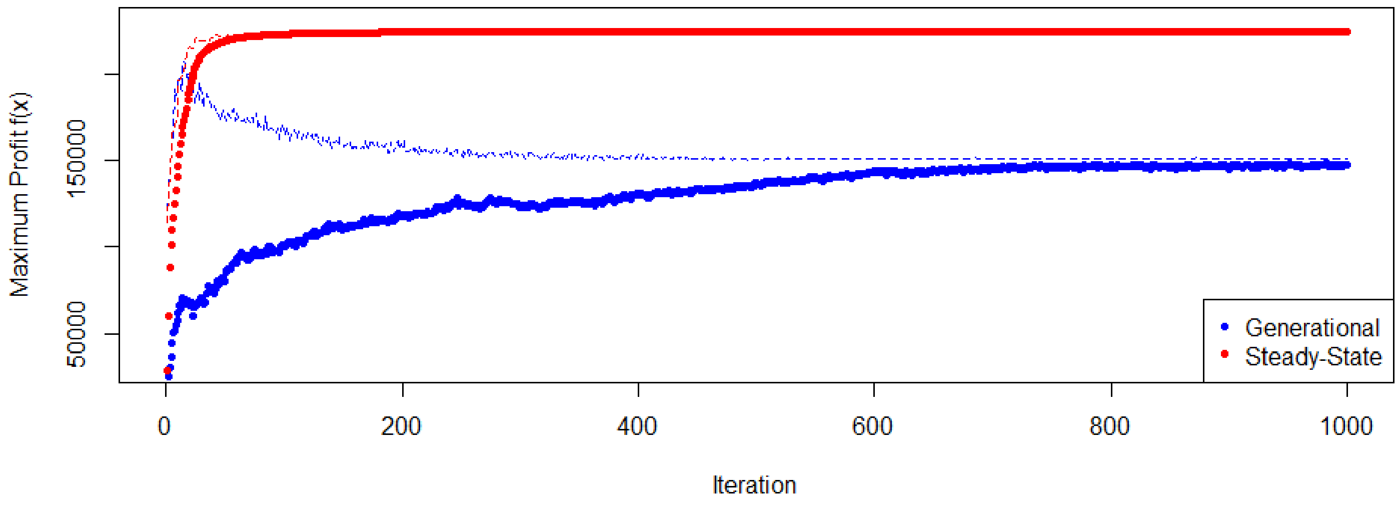
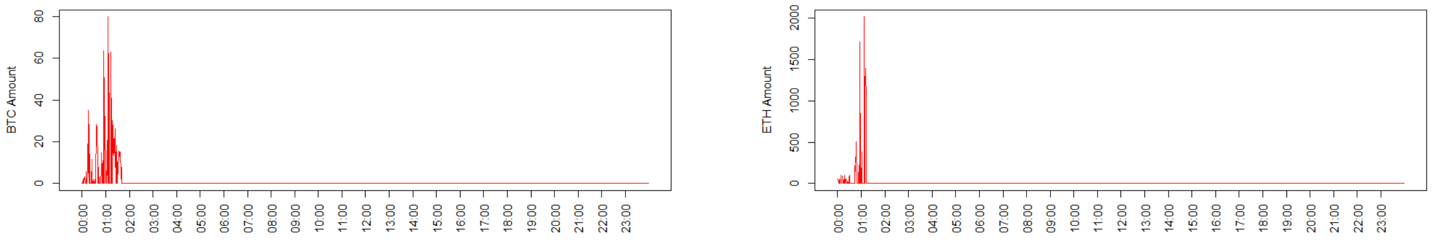
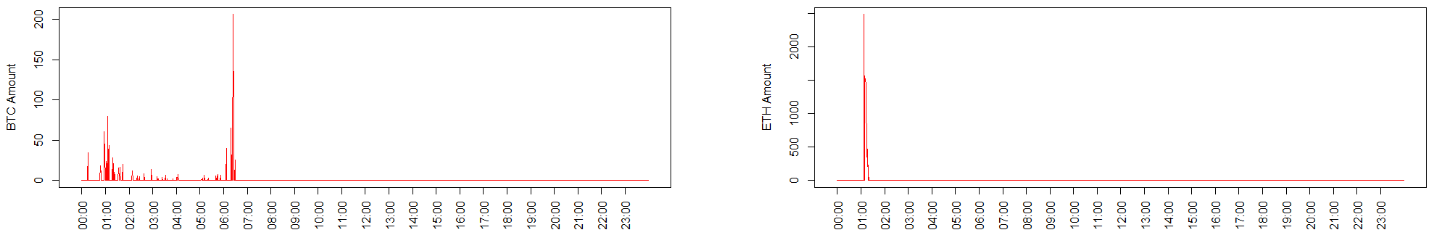


Figure 34: Sensitivity Analysis between Generational and Steady-State Replacement (Maximums of Population are Dotted Lines, with Corresponding Means being Circular Points)



(a) Amount of BTC Arbitraged (b) Amount of ETH Arbitraged

Figure 35: Solution of GA at Convergence with Generational Replacement



(a) Amount of BTC Arbitraged (b) Amount of ETH Arbitraged

Figure 36: Solution of GA at Convergence with Steady-State Replacement

7.2.5 Varying Population Size

Now we know that varying the population size will also have an influence on the convergence and optimum solution of the GA. Hence, for completion sake, we find it worthwhile to run a series of GAs (with the same genetic operations: selection proportional to fitness, 250 point crossover, scramble mutation and steady-state replacement) with varying population sizes.

Now Figure 37 displays these series of GAs for varying population sizes, where we note that only when the population size is 100 individuals does the GA lead to extremely quick convergence where it seems to get 'stuck' in a local maxima due to a lack of exploration. All other population sizes > 100 give rise to GAs which seem to be somewhat indistinguishable.

From theory we know that a larger population size allows for more exploration of the solution space, increasing the likelihood of finding global optima. It helps maintain genetic diversity, preventing premature convergence to suboptimal solutions. A smaller population size can lead to quicker convergence but may cause the algorithm to get stuck in local optima due to insufficient exploration - which is what seemed to occur in our case.

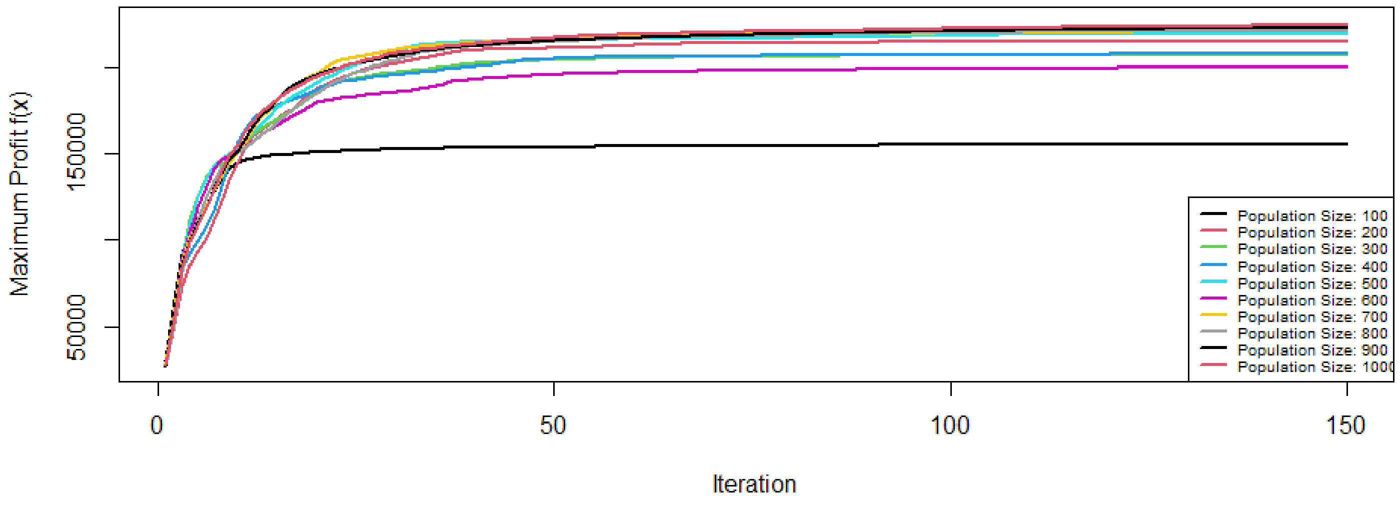


Figure 37: Sensitivity Analysis of GAs with Different Population Sizes

8 Analysis

We assess the overall quality of solutions between the three optimisation techniques (LP, SA and GA) used throughout the study, for our example arbitrageur with the BTC daily volume restriction being $V_1 = 1000$ and the ETH daily volume restriction $V_2 = 10000$. Additionally, we re-emphasize what was posited in the sensitivity analyses, including what was concluded with the goal programming.

8.1 Comparisons and Drawbacks

Now it is clear that for the daily volume constraints set at 1000 units of BTC and 10000 units of ETH, our LP gave rise to the highest maximum profit objective of $\$234,612.90$ out of all three optimisation methods utilised. Neither heuristic optimisation method, SA nor GA, was able to achieve a maximum profit greater than this (although we did argue earlier than for more iterations, the SA could have potentially given us a more competitive solution).

All optimisation methods suggest ETH to be traded mostly between `01:00-02:00` as in Figure 1b, Figure 11b or any of the ETH solutions in Section 7.2. BTC solutions differ somewhat between the optimisation methods: LP gives the majority of BTC arbitraging to be done between `01:00-02:00`, `06:00-07:00` and at `14:00` as in Figure 1a, with a somewhat similar solution in Figure 11a for SA. GA, suggests however, judging from the BTC solution plots in Section 7.2, the majority of BTC arbitraging to occur between `01:00 - 07:00`.

Now we know that LP is highly efficient at solving problems that are linear and convex, meaning that it can explore the entire feasible region and converge on the global optimum quickly. Since our arbitrage problem was well-structured and had a linear objective function and constraints, LP naturally performed better.

Additionally, the 'RGLPK' package utilizes the simplex algorithm [25], which guaranteed an optimal solution within the given constraints, whereas SA and GA are heuristic methods that gave rise to approximate solutions.

Furthermore, we learnt that LP explores the solution space systematically, using mathematical techniques to guarantee finding the global optimum. In contrast, SA and GA use probabilistic approaches, which are better for finding good solutions in complex landscapes but are not always guaranteed to find the absolute best solution, especially in a finite number of iterations.

Additionally, for our arbitrage optimisation problem: LP converged much faster to the best solution. SA and GA required more computational effort, as they explored the solution space stochastically. SA required more than 10000 iterations to obtain a maximum profit which was close to what was achieved with the LP. GA with steady-state replacement however, merely took less than 100 iterations to reach a maximum profit objective which was competitive to the LP maximum profit.

8.2 Sensitivity Analyses: SA and GA

Now we know that our heuristic methods: SA and GA needed careful tuning of parameters - and said parameters not being optimally tuned, might have had some part in SA and GA's underperformance relative to LP.

We saw in Section 6.3 that although the cooling schedules and associated T_0 and α did not vary the results of the SA much, we noted that varying n in the perturbation of our solution did (recall n in the perturbation algorithm was the amount of random indices $t \in [1, 1440]$ such that we set $x_{i,t} = 0$ initially). Furthermore, we hypothesized that the choice of cooling schedule, the cooling factor α and initial temperature T_0 had somewhat of a negligible effect on improving the profit objective, and this was likely due to the highly stochastic nature of the SA algorithm.

In Section 7.2, we noticed there not be any large discrepancies between solutions for GAs which utilized different selection, recombination or mutation operators. We concluded that a large n in n-Point crossover approximated uniform crossover, and also posited that large mutation rates lead to excessive randomness giving rise to poorer convergence. A significant disparity in the quality of solutions obtained by the GA was attributed to the choice of replacement operator: steady-state replacement gave rise to superior solutions relative to generational replacement. We concluded that generational replacement was discarding good solutions and never rediscovering them again after getting stuck in a local optimum. Additionally, we concluded that, for our arbitrage problem, a population size > 100 individuals was sufficient to enable adequate exploration of our solution space.

8.3 Goal Programming

Now Section 5 elucidated that, within the Archimedean framework, regardless of intensity of weight associated with our profit goal, w_1, the LP was always solved such that $d_1^- = 0$: implying that our profit goal was always achieved (deviations from our daily volume goals, d_2^+ and d_3^+, were thus greater as a result). Furthermore, we posited that it was more prudent to 'break' the ETH daily volume restriction g_2 than it was to 'break' the BTC daily volume restriction g_3 if one wanted to achieve the profit goal of g_1 - undergone by choosing a strict w_2. This being due to the fact that one would need to exceed the BTC volume restriction g_2 by almost $10\times$ if one were to perfectly achieve the profit goal g_1 and the ETH daily volume goal g_3. Yet, only exceed the ETH volume restriction g_3 by double if one were to perfectly achieve the profit goal g_1 and the BTC daily volume goal g_2.

Contrarily, Tchebychev goal programming resulted in all three deviational variables changing dependent on the intensity of w_1, w_2 and w_3. It was concluded that for Δ to be minimised - which represents the maximum deviation among all the goals - it would be prudent to utilize a strict w_1, with non-strict w_2 (and hence non-strict w_3 since $\sum_{k=1}^{3} w_k = 1$). This implies a greater desire to satisfy our profit goal g_1 with a lesser emphasis on satisfying both daily volume goals g_2 and g_3. Of course, if one were to prioritise other goals, different weights would be need to allocated accordingly with the assistance of Section 5.2.2.

Furthermore, we concluded that since with Archimedean GP, our profit goal g_1 was always met regardless of w_1, it would be more prudent to utilise Tchebychev GP. Seeing as there might be cases where one would want to prioritise daily volume goals and not the maximum profit goal, and Tchebychev GP would provide greater insight as to how to go about doing this.

9 Conclusion

In light of the findings from this study, we can assert with confidence that for real-world applications, cryptocurrency arbitrage optimization is more prudently executed using linear programming rather than heuristic methods like simulated annealing or genetic algorithms. This conclusion is particularly robust when the arbitrage problem is well-structured, exhibiting a linear objective function and linear constraints. While heuristic approaches produced approximate solutions that were within the same order of magnitude as those derived from LP, they consistently fell short of LP's precise maximization of profit. Moreover, LP achieved the optimal profit objective in a significantly shorter timeframe.

The heuristic methods' inability to outperform LP further strengthens the inference that the LP solution, as presented in Figures 1a and 1b, is indeed the global optimum, securing a maximum profit of $\$234,621.90$. Additionally, it is noteworthy that none of the optimization methods, with the exception of genetic algorithms with small population sizes and generational replacement, succumbed to the challenge of becoming trapped in a suboptimal local optimum far removed from LP's solution. This observation suggests that the optimization landscape of our problem was not heavily multi-modal.

In light of these insights, future investigations employing heuristic optimization techniques should prioritize parameter tuning that emphasizes exploitation over exploration of the solution space, given the relatively smooth optimization landscape encountered in this study. Such a strategy would likely yield more competitive solutions in line with the rigor and efficiency demonstrated by LP.

Appendices

Please find the attached '.rmd' file for all the code employed in the study, with the associated '.csv' files.

References

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Wiley, Chichester, England, 1989.

[2] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, 1996.

[3] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows.* Wiley, New Jersey, 4th edition, 2013.

[4] A. Charnes and W. W. Cooper. Goal programming and multiple objective optimizations. *European Journal of Operational Research*, 1(1):39–54, 1977.

[5] Marco Civitarese and Massimo Repetto. High-frequency trading and arbitrage opportunities in cryptocurrency markets. *Quantitative Finance*, 21(3):295–312, 2021.

[6] D. T. Connolly. General purpose simulated annealing algorithm. *Journal of Optimization Theory and Applications*, 63(3):379–397, 1990.

[7] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1998.

[8] Kenneth A. De Jong. *Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975.

[9] R. W. Eglese. Simulated annealing: A tool for operational research. *European Journal of Operational Research*, 46(3):271–281, 1990.

[10] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer, 2015.

[11] Matteo Fischetti and Matteo Stringher. Embedding simulated annealing within stochastic gradient descent. In Belen M. Dorronsoro, Lionel Amodeo, Mario Pavone, and Pablo Ruiz, editors, *Optimization and Learning. OLA 2021*, volume 1443 of *Communications in Computer and Information Science*, pages 3–17. Springer, Cham, 2021.

[12] Neil Gandal and Hanna Halaburda. Competition in the cryptocurrency market. *Journal of Financial Stability*, 13:47–61, 2014.

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[14] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.

[15] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[16] Lester Ingber. Simulated annealing: Practice versus theory. *Mathematical and Computer Modelling*, 18(11):29–57, 1993.

[17] Jung Yeon Kim, Seok Kim, and Jinho Hong. Arbitrage opportunities in cryptocurrency markets: A simulation-based study. *Journal of Computational Finance*, 22(4):47–64, 2019.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[19] Shan Liu, Yuheng Lu, and Xin Zhang. Reinforcement learning in cryptocurrency trading: A framework for real-time arbitrage optimization. *Quantitative Finance*, 21(5):723–741, 2021.

[20] David G. Luenberger. *Optimization by Vector Space Methods.* John Wiley & Sons, New York, NY, 1997.

[21] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[22] Saralees Nadarajah and Jeffrey Chu. On the inefficiency of bitcoin. *Economics Letters*, 150:6–9, 2017.

[23] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, NY, 1988.

[24] C. Romero. *Handbook of critical issues in goal programming.* Pergamon Press, 1991.

[25] Stefan Theussl and Kurt Hornik. *Rglpk: R Interface to the GNU Linear Programming Kit*, 2020. R package version 0.6-4, Available at `https://CRAN.R-project.org/package=Rglpk`.

[26] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[27] M. Zeleny. *Multiple criteria decision making.* McGraw-Hill, 1982.