



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

How to build an arbitrage bot

DISSECTING THE COMPONENTS
TO BUILD A MEV ARBITRAGE BOT

Authors:
Marco Fasanella
Elia Maggioni
Lucas Manini

Student IDs: 10617541 - 10610008 - 10625965

Advisor: Prof. Andrew Damian Tamburri, Giovanni Quattrocchi

Co-advisors:

Academic Year: 2021-22

Abstract

The rise in popularity of blockchain systems, and in particular in smart contract platforms such as Ethereum, has driven a large amount of engineers and developers towards working on the challenges and opportunities presented by this new computation paradigm. Among the most popular categories of applications built in web3 are financial applications, which have taken the crown as the most utilized by every-day users.

Ever since the existence of trade, many keen-eye market participants have searched closely for opportunities to generate profit. One such kind of participants are arbitrageurs, who exploit mispricings of the same asset between two different markets, in order to gain a small margin, effectively reducing such market inefficiencies.

The aim of this paper is to use a systematic approach to provide fundamental knowledge on automatic arbitrage trading between blockchain based exchanges. We will provide an understanding of how these systems work, the laws behind the most common implementations and a model of the problem at hand.

We will also propose an architectural pattern of which we will analyze in our own implementation, providing benchmarks, spatial and temporal complexities of the overall system.

Keywords: MEV, ETH, Financial Arbitrage, Blockchain, Golang

Contents

Abstract	i
Contents	iii
1 Introduction	1
1.1 Blockchain	1
1.1.1 Ethereum blockchain	1
1.2 Decentralized Finance	1
1.2.1 Financial arbitrage	1
1.3 AAM - Automated Market Maker	2
1.3.1 CPAAM	2
1.3.2 WCPAMM - Weighted Constant Product Automated Market Maker	3
1.4 MEV	4
1.5 External tools used	5
1.5.1 Liquidity Pools (in Balancer)	5
1.5.2 TheGraph	5
1.5.3 Abigen	6
1.6 Ethics	6
2 Problem Description and Model	7
2.1 Problem Description	7
2.2 Model	8
3 Architecture	11
3.1 Generic software architecture	11
3.2 Problem Constraints	12
3.3 Solutions: Divide et impera	12
3.4 Startup-time module	14
3.5 Runtime module	15

3.6	Concurrency	16
4	Benchmark Implementation	17
4.1	Language and hardware requirements	17
4.2	High-level description of the algorithms	17
4.2.1	Startup-time module	17
4.2.2	Runtime module	19
5	Performance Evaluation	21
5.1	Startup-time performance (space complexity)	21
5.1.1	Dataset Creation	21
5.1.2	Benchmark MaxSteps	22
5.1.3	Benchmark GraphSpace	23
5.2	Runtime performance (time complexity)	23
5.2.1	Benchmark CheckArbitrage	25
5.2.2	Benchmark Sync	25
6	Conclusion and future works	27
	Bibliography	29

1 | Introduction

In this first section, we provide minimal knowledge regarding the building blocks on which our work is based on.

1.1. Blockchain

1.1.1. Ethereum blockchain

Ethereum is a permissionless peer-to-peer network, powered by an underlying blockchain and its native virtual machine. It is commonly known for its native cryptocurrency ether, or ETH, which is used to pay for the fees involved in sending transactions to its blockchain. Apart from storing and sending value it may be used to create any kind of program, commonly referred to as 'smart contract'; leveraging the Ethereum Virtual Machine's instruction set.

1.2. Decentralized Finance

A category of smart contracts which implement financial services, such as money markets where users are able to take out loans or lend their tokens, or exchanges where users may trade.

1.2.1. Financial arbitrage

Arbitrage trading is a sophisticated high frequency trading practice which can be used whenever any asset listed on financial markets has a small discrepancy in the listed prices between any two given markets.

Essentially, the strategy consists in purchasing and selling simultaneously the same asset in two distinct markets, in order to profit from temporary market inefficiencies, which occur naturally between financial markets. This is usually known as binary arbitrage, as it involves buying or selling in two distinct markets.

Later in the paper, we propose a way to model these assets and the markets they are

traded on, alongside a model to represent not only binary arbitrages, but n-ary arbitrages as well.

1.3. AAM - Automated Market Maker

The rise in popularity of smart contract platforms like Ethereum has pushed developers and engineers worldwide to build (and often, rebuild) systems which mimic already existing ones, especially financial systems.

This sub-class of smart contract based systems is broadly referred to as Decentralized Finance (or DeFi, for short), highlighting the fact that these new systems are operated by code, which is executed and verified by every single node of the blockchain's peer network. Among the most popular types of DeFi applications, first ones to be born were an Automated Market Maker based Decentralized Exchanges (or AMM based DEXes, for short), which have been engineered with the objective of escaping the common orderbook structure for today's financial exchange markets and grant users immediate execution on the trade they wish to take, with no need for the presence of a counterpart willing to take the opposite side of the trade at the same price.

Because they do not use an orderbook to determine at when and at what price trades are executed, AMMs make use of specific mathematical equations to determine the price at which a trader is able to swap a token X for another token Y.

The first widely popular DEX which is based on an AMM is Uniswap [1] which in its v1.0, implemented what is now known as a **Constant Product Automated Market Maker** (CPAMM, for short), whose architecture was first conceptualized by Vitalik Buterin in a blog post [2] in 2017.

1.3.1. CPAAM

Constant Product Automated Market Makers are named after the characteristic equation they use to determine the execution price of a single trade. This kind of AMMs allow trade between two given tokens and are composed of 2 pools of such tokens, these commonly referred to as liquidity pools.

Let the amounts of tokens in the liquidity pools be x and y, thus the CPAMM's characteristic equation is defined as:

$$x * y = k$$

where k is a constant value set when the AMM is created and can take any positive value.

Note that the law above holds at all times for every CPAMM market. Also, using this law we can uniquely define the relative price of tokens involved in the pools, given by:

$$P = \frac{dy}{dx}$$

where dy and dx are differential amounts which enter and exit the pool. These two quantities may be defined one with respect with the other, depending on whether we wish to fix the amount of tokens entering or exiting the tokens.

For brevity, we report the amount of tokens exiting a pool, given a fixex amount of tokens entering the pools, dx :

$$x * y = (x + dx) * (y - dy) = k$$

$$dy = y - \frac{x * y}{x + dx}$$

$$dy = y(1 - \frac{x}{x + dx})$$

Note that the calculations above exclude fees the CPAMM may charge traders.

In that case, the entering amount of tokens dx is multiplied by a constant $f = 1 - fee$ (e.g: a 0.3% CPAMM fee results in $f = 1 - 0.003 = 0.997$).

1.3.2. WCPAMM - Weighted Constant Product Automated Market Maker

The model popularized by Uniswap was a true breakthrough for the Ethereum community, finally bringing to life an architecture which many thought would not be able to function properly for an extended period of time. It may thus come as no surprise, that other engineers and developers decided to improve and expand this architecture.

A generalization of the CPAMM model is the Weighted Constant Product Automated Market Maker (WCPAMM for short), popularized by the Balancer protocol.

This type of AMMs allows to create markets with a larger amount of tokens in them, thus bringing together tokens to a single pool.

Specifically, let n be the number of tokens within a generic pool, then we can calculate

the number of distinct pairs of tokens users can trade using the pool as the binomial coefficient of n over 2:

$$numberdistinctpairs = \frac{n!}{(n-2)! * 2!}$$

WCPAMMs are defined by the following equation:

$$x^{w_x} * y^{w_y} * z^{w_z} * \dots = k$$

where x , y and z represent the amounts of tokens X , Y and Z in the liquidity pool, k is an arbitrary positive constant akin to the one in CPAMMs and all weights w_i are such that:

$$\sum_i w_i = 1$$

Finally, similarly like we did with CPAMMs, we can uniquely define the amount of exiting tokens dy , given an entering amount of tokens dx in a trade from X to Y as:

$$dy = y * \left(1 - \frac{x}{x + dx}^{w_x/w_y}\right)$$

1.4. MEV

«Maximal extractable value (MEV) refers to the maximum value that can be extracted from block production in excess of the standard block reward and gas fees by including, excluding, and changing the order of transactions in a block.»[4]

This concept was firstly applied to **proof-of-work**, a procedure to allow the nodes of the network to agree on the state of all information recorded on the blockchain, and concerned only the miners.

The term MEV (formerly Miner, now Maximal, Extractable Value) was first introduced in the Flashboys2.0 paper [6], with which the authors identify *all the value blockchain miners are able to extract for their own profit by reordering, copying, stealing and/or censoring transactions*.

Given their privileged position as block builders and proposers, and given the fact that miners always must simulate transactions before including them in a block, some miners noticed they were in prime position to censor and copy transactions that would generate

a profit for the sender (e.g. a transaction which arbitrages the price discrepancy of an asset in two different markets).

Today, MEV has been rebranded to Maximal Extractable Value, thanks to Flashbots and Flashbots Auction, as access to capture such value has been democratized and made accessible to other ecosystem participants, namely "searchers", via a sealed-bid auction mechanism in which searchers offer direct payment to miners in exchange for being inserted at the top of next block, thus capturing the MEV opportunity.

Some MEV examples are:

- Liquidations
- Sandwich trading
- NFT MEV
- DEX arbitrage

Among all these techniques we will consider the DEX (decentralized exchange) arbitrage, which is the most well-know MEV opportunity, but also the most competitive one.

1.5. External tools used

1.5.1. Liquidity Pools (in Balancer)

Balancer Pools contains two or more tokens that traders can swap between. The main components of a liquidity pool are:

- Liquidity Providers: a market participant that provides liquidity, put their tokens in the pools in order to collect swap fees.
- Liquidity Taker: a market participant that buys or sells one asset in exchange for another asset, by taking the liquidity offered by a liquidity provider

Balancer is a community-driven protocol, automated portfolio manager, liquidity provider, and price sensor that empowers decentralized exchange and the automated portfolio management of tokens on the Ethereum blockchain and other EVM compatible systems.

1.5.2. TheGraph

This is a tool that allows to query the blockchain as a database: it's datasets are called subgraphs. A subgraph is a GraphQL database in which entities are recorded as defined

by an implementation (the mapping), and a schema.

A subgraph can be deployed on-premise or used on the hosted service provided by TheGraph.

Essentially, in a TheGraph node there aren't many parts at play, the base layer is a Postgress DB on which entities are defined as prescribed in the mapping file.

In the mapping file, developers can specify which events, logged by specified smart contracts, the TheGraph's nodes are supposed to listen for and in what way the underlying DB (the entities) should be modified, using the event's logged data.

1.5.3. Abigen

Given that our implementation was carried out in a statically typed language such as Golang, we needed a way to interact with the Ethereum blockchain and the smart contracts deployed on it in a compile-time type-safe manner. This is achieved thanks to GoEthereum's (geth, for short) abigen CLI tool, a source code generator that can convert Ethereum smart contract ABI definitions into type-safe Golang packages. Once we've created an instance of the generated package (if you're not familiar with Golang packages, read: `class`) by providing it an Ethereum address and a connection to an Ethereum node, we're able to generate, sign and send transactions calling any external method of the target smart contract with a single method call on the package's instance. We use this extensively so as to be able to execute our arbitrage trades between Balancer's different pools.

1.6. Ethics

A last part to consider, is the impact that these kinds of techniques have on the Crypto Ecosystem. MEV is not bad, for instance in the case that we are considering in this paper the constant research and fix of inefficiencies in DEX arbitrage, ensures that users get the best, most correct prices for their tokens.

However in some cases like sandwich trading, it will result in an unequivocally bad experience for users. Users who are sandwiched face increased slippage and worse execution on their trades.

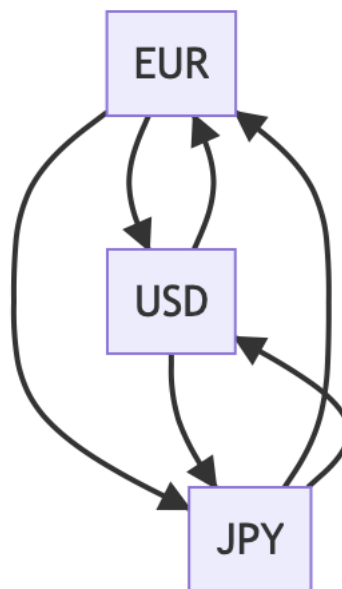
As Bitcoin's block reward halves and transaction fees make up a greater and greater portion of the block reward, miners tend to give up the next block's reward and instead remine past blocks with higher fees.

2 | Problem Description and Model

2.1. Problem Description

Every market, on-chain or off-chain, can be modeled as a complete graph $G = (N, E)$, in which the set of nodes N is the set of assets tradable in such market, and the set of edges E is all the possible edges between the nodes in N .

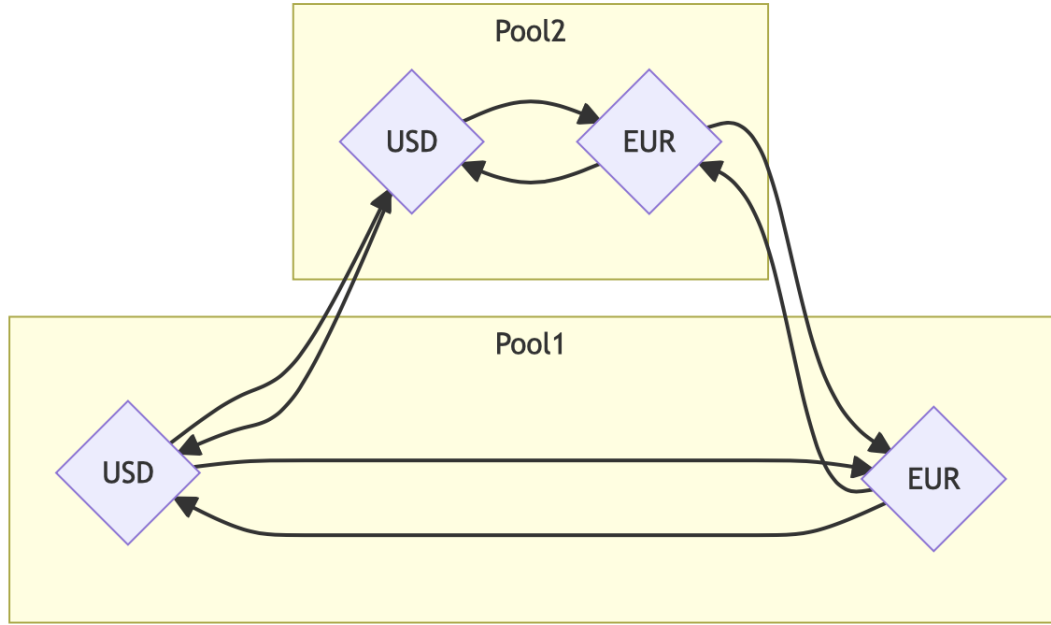
As an example, let's consider a simplified version of a foreign exchange in which the tradable assets are Euros *EUR*, United States Dollars *USD* and Japanese Yen *JPY*, it's graph model would be:



Given that, in DeFi, a smart contract which allows the trading of different assets is often referred to as a liquidity pool, from here forth we will refer to such a market, or graph, as a *Pool*.

In a *Pool* the laws governing exchanges are very variegated and in some cases complex: for the purpose of this paper we will consider only **Balancer's Weighted Pools** which are governed by their **Weighted Constant Product invariant** and its generalization in *Pools* holding more than 2 tokens.

We will refer to a set of *Pools* as a *Market*



2.2. Model

The fundamental entities in any program that aspires to capture arbitrage opportunities are the Token and the Pool entities.

We represent a *Pool* as tuple containing, among some metadata, an array of *Token* which will record each token's balance, allowing us to compute the exchange rates provided by such pool, and the *Pool's* Ethereum address.

Given a *Market* composed of n distinct pools, we can represent all the pools within a single consolidated graph by putting as many edges between nodes as there are pools that allow for the two involved tokens to be traded.

In order to identify viable arbitrage routes, we will then compute all possible, non zero-length, **cyclic paths** on the consolidated market graph, taking us from a selected **root token** to a **destination token** such that $roottoken = destinationtoken$, for each token in the market.

The underlying idea is that we will be able to exchange our tokens in different Pools and

we will be able to make a profit transitioning from one pool to another, by exploiting the implied price discrepancies among the traversed pools.

Let's understand this with an example and some simple numbers:

Given 2 identical pools $Pool_A$ and $Pool_B$, containing each 2 types of token *CANDY* and *EURO*. In $Pool_A$ we have an exchange rate governed by a simple fixed law $CANDY = K * EURO$, with K different for each $Pool$: $Ka = 1$ and $Kb = 2$.

It's easy to understand that this market is not efficient and there is an arbitrage opportunity to make a profit.

We will obtain a profit by exchanging x EURO for x CANDY in $Pool_A$ through a 1 to 1 exchange rate, then exchange x CANDY for $2x$ EURO in $Pool_B$, doubling the amount of *EURO* we started with.

We can see how this generates an infinite profit: this is because of the very simple nature of the exchange law.

Note that in real-world financial markets this is never the case, as capturing an arbitrage opportunity causes the exchange rates to converge towards an equilibrium in which the assets are priced so similarly that the arbitrage opportunity is no longer profitable to extract, often because of exchange fees, and thus ceases to exist.

This opportunity will be represented in our model as a cyclic path:

$$EURO \Rightarrow CANDY \text{ in } Pool_A, CANDY \Rightarrow EURO \text{ in } Pool_B$$

3 | Architecture

In this section, we will describe the main challenges faced during the design and how the architecture as a whole evolved around them to accommodate very complex and time-consuming algorithm.

3.1. Generic software architecture

Any system which focuses on arbitraging DEXes, must be made of the following components:

- A *backend system*, implemented in an efficient and "fast" programming language: used to carry out all critical computation and execute on detected opportunities.
- A *smart contract deployed on the blockchain*: this acts as the sole entry point for all of the bot's trades, as it's thanks to the unique properties these programs guarantee that our system is able to carry out an atomic series of trades which must generate a profit for the sender.
- An *Ethereum client*, which acts as a bridge between the two components listed above, allowing the backend to see and inspect the blockchain's latest state.

It's important to note that today's sophisticated setups will employ multiple gateways to a subset of the blockchain miners, in order to be able to submit the arbitrage-capturing transaction directly to the party who will include in the next block and (hopefully) be lucky enough to be the first to find the next valid one.

Such gateways today might be:

- Flashbots Auction
- Eden Network

3.2. Problem Constraints

When designing the system's architecture we had 2 main constraints: one external and one internal. The former is intrinsic to the problem: we want to capitalize on market-wide inefficiencies, meaning we have to operate as close as possible to very large transactions that left one or more Pools with an exchange rate significantly different from one of another set of Pools. Assuming such a large trade is included at the latest block X of the blockchain, as soon as that block is published, we can safely assume that information regarding the existing of such a profitable opportunity is common to all participants looking for such opportunities. Thus, we must compete with all others to correctly identify and execute on such an opportunity before block X+1 comes along, as by the time it's published, the arbitrage may be gone.

«Given that we must be able to compete for extracting the arbitrage in the next block, this leaves us with a time window of approximately 12s.»

The latter is space and it's a limitation that is not embedded in the problem itself but is one that will arise in your implementation. As we will see later, we will be dealing with hundreds of Pools, each containing 2 to 8 Tokens.

«If we tried to generate all possible graphs using all identified pools, we would need hundreds of GB of RAM.»

This might be fine if you're working in a data center but will take some consideration while building your dataset on your PC. As it often happens we will have to compromise and sacrifice a bit of execution speed to deal with our space constraints.

3.3. Solutions: Divide et impera

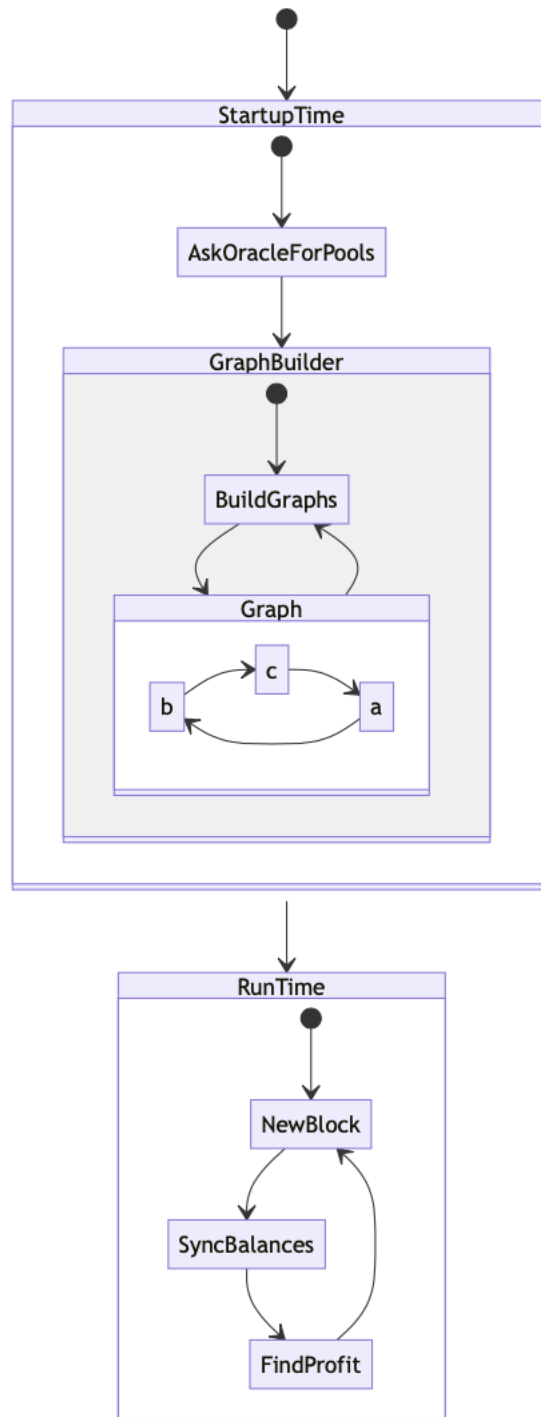
The core idea of the architecture is the separation between a runtime module and a startup-time module.

This is possible because the *Pools* balances are rapidly changing but their composition is not. Given that new Pools are created very rarely, in a time scale very different from what we're dealing with, we can ignore that or run the startup module every so often.

Thus, we present our architecture's logical module separation:

- A **startup-time** module that will handle the consolidated market graph, building it and providing the extracted dataset in a data structure optimized to be read and updated rapidly.

- A **runtime** module that will listen for new blocks and keep an updated balance of each Token in each Pool, compute if there exists a cycle that is generating a profit and then submit that transaction to the blockchain.



3.4. Startup-time module

As cited above, the main purpose of this component is to provide a dataset containing all the necessary information for the **Runtime module**, created in such a way to be very efficiently read and updated. To minimize many of the constants in the Runtime module time complexity, we will have to increase the size of our dataset by creating secondary access keys.

«There is a compromise between the size of the dataset and its efficiency»

In order to understand why we need secondary accesses we need to state what are this module's inputs and outputs. In input we have an array of Pool, each containing an array of Token. In output we want a dataset containing a step-by-step description of every possible path leading from Token X back to Token X, which are the possible arbitrage paths. The number of cyclic paths grows exponentially with the number of Pools (assuming that the number of Pools is comparable to the number of Tokens and that the Tokens are uniformly distributed among them, which is reasonable to assume). Given n Pool containing each k Token randomly selected from a set of m , we have a probability:

$$P(x) = \frac{\binom{m-1}{k-1}}{\binom{m}{k}} = \frac{k}{m}$$

of Token x being present in Pool y . The number of times x will appear in our market is:

$$N(x) = n * P_k(x) = \frac{n * k}{m}; m \geq k$$

Each token $xN(x)^2$ interconnection graphs for each token. These are the building blocks of our graphs, but we still need to factor in the token conversion.

Now let's focus on a single Token in a single Pool. This Pool will have $xN(x)^2$ connected Pools, each with a probability $P_k(x)$ to have x as a Token.

We repeat this process s times, where s is the maximum number of steps.

So for 1 step we have

$$(k - 1) * N(x)(P_{k-1}(x))$$

For 2 steps we have

$$(k - 1) * N(x)(P_{k-1}(x) + (k - 1)N(x))(P_{k-1}(x))$$

$$(k-1) * N(x)(P_{k-1}(x) + (k-1)N(x))(P_{k-1}(x) + (k-1)N(x)(P_{k-1}(x)))$$
$$P_{k-1}(x) \sum_{i=0}^{s-1} ((k-1) N(x))^{s-i}$$

3.5. Runtime module

This is because otherwise we could potentially run on outdated balances, effectively doing useless work.

For the syncing of new blocks we can rely on external services or a local Ethereum node, using whichever grants the least amount of latency, in order to maximize the time we have available to traverse the arbitrage path, therefore using a solution such as TheGraph here is unadvisable, as TheGraph must process all received blocks and update it's internal DB, before making these changes available to connected clients.

Now we come to the most delicate part of the application: if all was done right until here we can expect to have in the order of 10^1 seconds to find a profitable path in our cycles. This can be done in several ways: a trivial and simplistic approach could be to blindly simulate trades on each cycle in a local fork of the blockchain, discarding those who don't capture an arbitrage.

A more sophisticated and efficient approach would be re-implementing in native code the math library used in the market smart-contract and using the custom math library to analytically determine if an arbitrage opportunity is present in a certain cycle.

3.6. Concurrency

Many of the tasks above can be efficiently parallelized, with some caution. In order to create an application that is able to run without an extraordinary amount of memory, specific caution must be taken when trying to create or read a graph with multiple threads. For example if we run Deep First Search DFS (iterative or recursive) for building our graph in n threads (for instance running it on n different branches), in the worst case we will have n times the maximum occupied amount of RAM for the same graph.

It's important to understand, before allocating 10TB of swap memory on your hard-drive, that it's useless and often harmless to go beyond a certain amount of threads running in parallel on your system.

This number is defined by your CPU architecture and can be obtained at runtime using a library in most cases.

Also, keep in mind the usual best practices like not starving your threads and don't cross-access memory or if you do so, make sure to properly synchronize your code.

4 | Benchmark Implementation

In this section we will illustrate the implementation created to validate our architectural design patterns and to verify our theoretical results.

4.1. Language and hardware requirements

In order to increase performances we advise to use a programming language that was running on hardware and not in a virtual machine. We selected from the wide range of possibilities the language Go, for a couple of reasons.

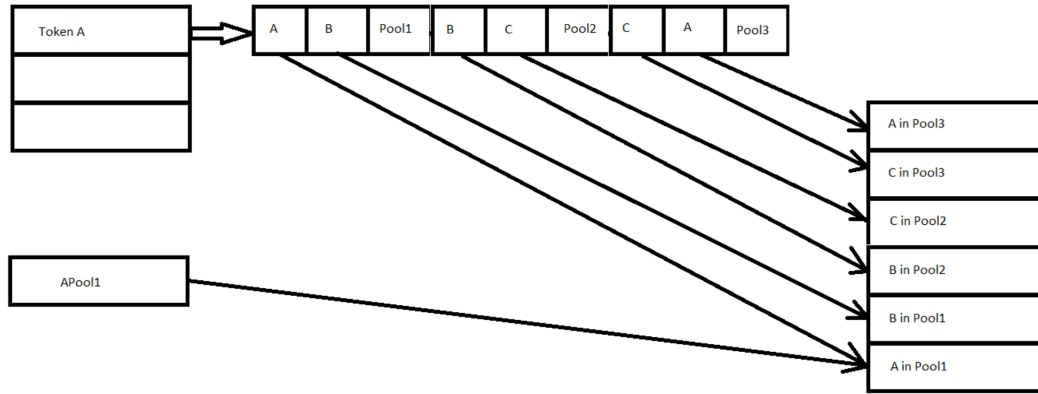
It's C-like so everybody was conformable with it, but it also provided very good libraries for multithreading and in general was a really strong candidate providing good performances (not best, it loses a little bit against hyper optimized programming languages like Rust) and ease of use. Also, there is a very good Ethereum library which helped this choice.

4.2. High-level description of the algorithms

As previously designed, we divided the implementation in 2 parts, the *setup-time module* and the *runtime module*.

4.2.1. Startup-time module

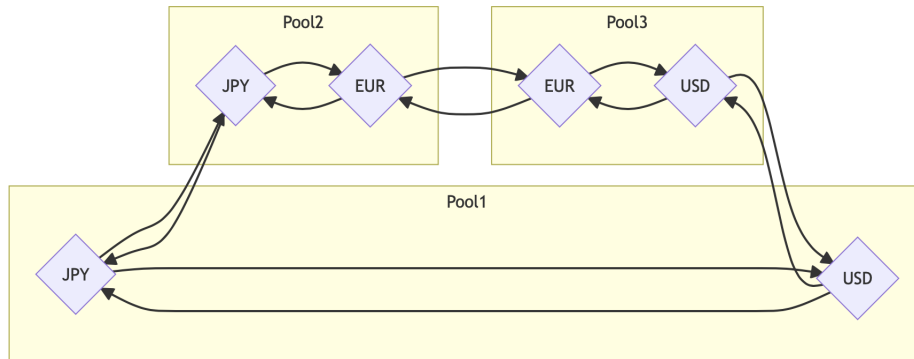
To obtain the theoretical optima described before, we elected to use a dual access dataset and to implement it as a pointer structure. For the primary access method, we used a simple hashmap from Token to an array of Paths. Each Path is a struct composed of a Starting Token, an Arriving Token and a Pool. The tokens are provided by reference, meaning they are pointers to Token. This allows us to create a secondary access structure, also in this case an hashmap, mapping from the string concatenation of *Pool.Address* and *Token.Address*. This is what the resulting dataset looks like:



To generate the dataset we use recursive DFS (deep first search) on an initial array of Pools provided by TheGraph.

As stated before, it's very important to filter out any possible duplicate to avoid repeated paths. To do this we implemented a graph reordering technique that works by letting the DFS proceed with a cycle only if certain conditions are met. Let's tackle this with an example.

Suppose to have again the market:



Let's build the paths starting from *USD*:

- USD -> JPY [Pool1]
- JPY -> EUR [Pool2]
- EUR -> USD [Pool3]

and

- USD -> EUR [Pool3]
- EUR -> JPY [Pool2]
- JPY -> USD [Pool1]

we want to select only one of these 2 paths. To do this we use a temporary hashmap mapping from Token to Token, containing the next iteration for every token we visit.

For example, in path 1 $Map[USD] = JPY$, $Map[JPY] = EUR$ and so on.

Before adding a new token to our path, we check if $oldToken = Map[newToken]$ and we add only if $oldToken > newToken$. In our example, the procedure would be:

- USD $\rightarrow$ JPY: $OldToken = Map[JPY] = null$ ok $Map[USD] = JPY$
- JPY $\rightarrow$ EUR: $OldToken = Map[EUR] = null$ ok $Map[JPY] = EUR$
- EUR $\rightarrow$ USD: $OldToken = Map[USD] = JPY$, $JPY > EUR$? *continue : fail*

This must fail in one of the two cases leaving us with cycles only going in 1 direction. This also works for nested repetition, not only on the root token.

There is a small issue with this procedure, if we have a symmetric cycle like $A \rightarrow B$, $B \rightarrow A$ this cannot work. In this case we split the equality on the pool address, meaning that if at the last step $JPY == EUR$, then we only take the cycle starting from Pool1 and ending with Pool3, instead of the reverse.

The last step is therefore modified as such: $JPY == EUR ? splitOnPool() : (JPY > EUR ? continue : fail)$.

The DFS implementation is heavily parallelized, we launch a subroutine with each token as Root of the tree and we merge everything into a single dataset. Our implementation controls that only the maximum optimal amount of threads is running before starting a new one.

4.2.2. Runtime module

Given the architectural premises stated before, this part is relatively straightforward. We have 3 main components here, the Sync module, the Math library and the block watcher.

Let's start from the simplest, the block watcher is a very simple thread that using a Go Channel and a pointer to a variable, keeps that variable always updated with the latest block. The variable allows us to check if we are still current whenever we want in the program with a simple (synchronized) memory access. The channel is utilized by the main loop to wait for a new block to appear when the computation of the paths is over.

The Sync Module, as the name suggests, is triggered whenever a new main loop is initialized and synchronizes from the last seen block to the newest. It does so in the most

efficient way possible directly reading the logs from the newly arrived block and updating the internal balances in the dataset using the $O(1)$ access provided by the *Token+Pool* *hashmap*.

At the core of the main loop we have a producer-consumer pattern utilizing a reader thread that distributes the computation across an optimal amount of threads.

The reader thread sequentially reads the dataset and fills a buffered channel with the graph's paths. The consumer threads read from the channel and using our implementation of the smart contract's math library check if there's a profit to be made.

If they find it, they produce and sign a transaction which is given back in console. Whenever they finish, they get back to the channel to start a new iteration. Every new iteration the `currentBlock` variable is checked and if we are outdated the channel is closed and the loop is repeated.

5 | Performance Evaluation

In this chapter, we analyze the results we've achieved with our implementation of the presented architecture, dividing our analysis in the two distinct runtime phases we've established.

For each part, we analyze it's most critical aspect, namely: for the startup-time module, we analyze it's space complexity, whilst for the runtime module we analyze it's time complexity.

5.1. Startup-time performance (space complexity)

From the formula derived of the number of graphs derived in the Architectural Section, we obtain this complexity

$$O\left(P_{k-1}(x)(N(x)(k-1))^s\right) = O\left(\left(\frac{k-1}{m}\right)\left(\frac{nk}{m}(k-1)\right)^s\right) \approx O\left(\left(\frac{k}{m}\right)\left(\frac{nk^2}{m}\right)^s\right)$$

with $k \gg 1$

Where k is the number of tokens in a Pool, n is the number of Pools and m is the total unique tokens.

This complexity has been empirically verified via benchmarks of all principal components of our system.

5.1.1. Dataset Creation

Benchmarks were performed on three different *datasets*:

1. Red, CPU: Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
2. Blue, CPU: Intel(R) Xeon(R) Platinum 8272CL CPU @ (Github Cloud)
3. Black, CPU: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz

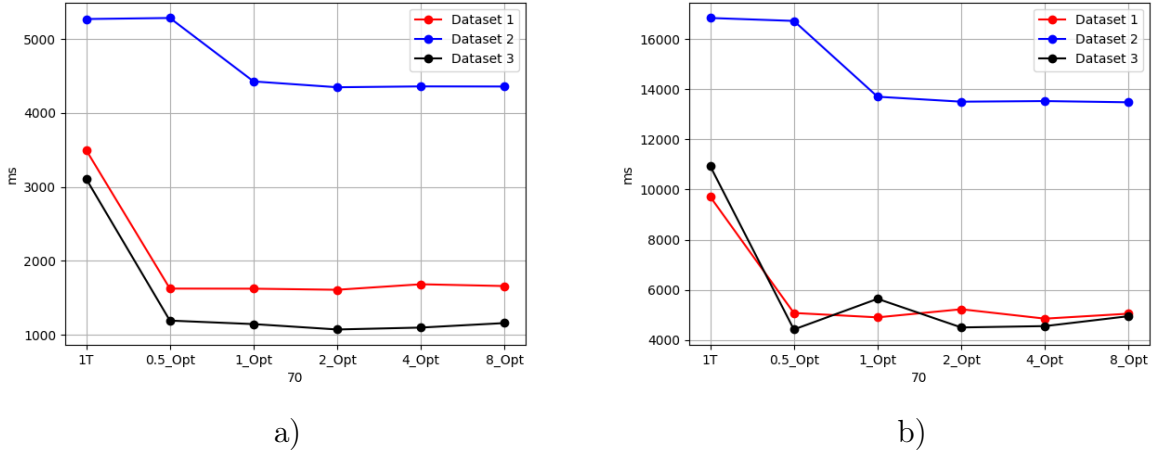


Figure 5.1: Shows the amount of time spent creating the market graph, starting from a variable number of the top Balancer pools ordered in descending order of USD held.

- (a) Shows the amount of time spent creating the market graph using a fixed number (70) of pools, with respect to the number of parallel go routines employed to carry out the computation.
- (b) Shows the amount of time spent creating the market graph using a fixed number (80) of pools, with respect to the number of parallel go routines employed to carry out the computation.

Notice that in the 2 last benchmarks, the results may be influenced by the machine's memory performance, thus we show these 2 so as to have a comparison with equal RAM performance.

5.1.2. Benchmark MaxSteps

Shows the amount of cyclic paths we find, based on the maximum length allowed for such paths.

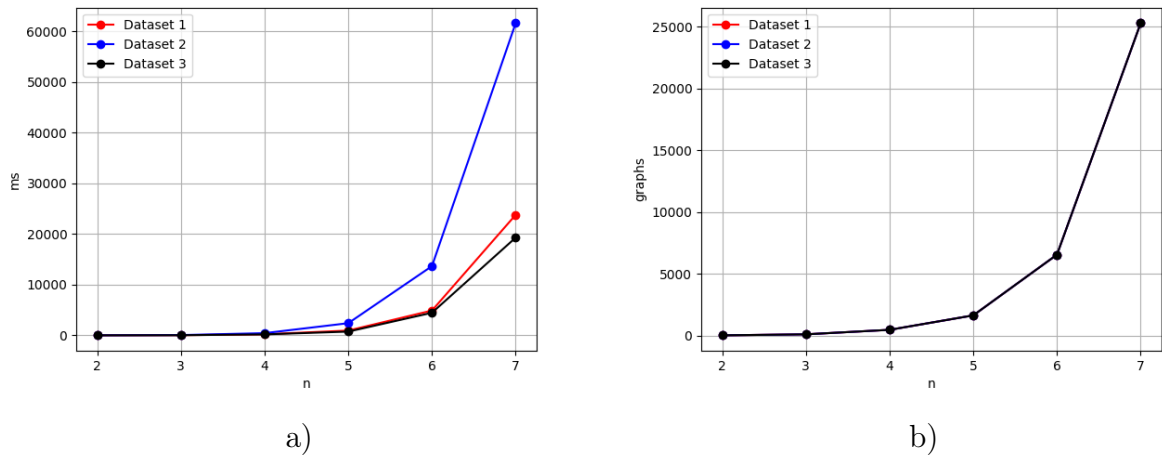


Figure 5.2: a) Time(ms) and b) Space(Graphs)

5.1.3. Benchmark GraphSpace

Shows the amount of time spent creating the market graph, starting from a variable number of the top Balancer pools ordered in descending order of USD held.

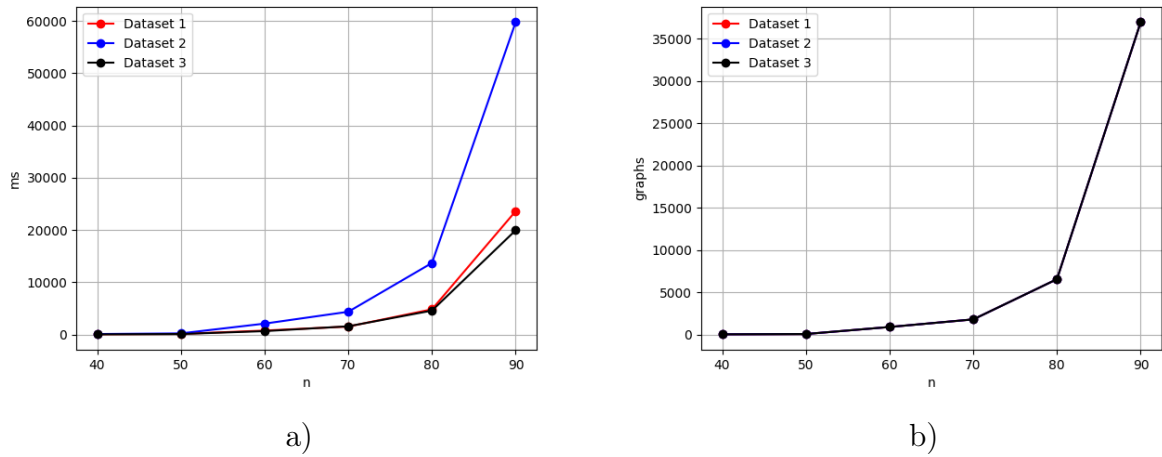


Figure 5.3: a) Time(ms) and b) Space(Graphs)

5.2. Runtime performance (time complexity)

The following decisions were made to best respond to the following questions/problems:

- When do we start looking for new arbitrage opportunities?

- What's the best way to keep up with our monitored pools states?
- How should we search our consolidated market graph?

As previously described, our arbitrage-checking algorithm is a continuous loop, whose execution is triggered as soon as the Ethereum client we're connected to, notifies us that a new block (and hence, a new state) has been mined and pushed to the blockchain. Because of the necessity of being notified when a new block is mined, we've elected to always maintain a connection to our node via a websocket-based subscription.

As soon as the new block message has been received through our pipeline, the first thing to take care of is the pools' internal token balances. For this purpose, we exploit the event logs emitted by Balancer's smart contracts, from which we are able to extract all token movements in every pool we track and apply such deltas to the in-memory graph. This step of the algorithm has been optimized so as to run linearly in the number of *emitted logs*, which ensures us that we aren't doing any excess computation to keep our balances updated.

Keeping balances in the bot memory and applying the bare minimal updates is the best compromise between speed and ease of deployment which fits our setup, given that we were working with a *remote* Ethereum node.

Now that we're sure to be working on the latest observed state, we can start the search for a sequence of trades which, at the end, return more tokens than the amount given, while also accounting for all fees incurred. To optimize this path, the best approach we've found was to implement from scratch all the relevant functions from Balancer's Weighted Math library[7], which is the library it's smart contracts use. In order to make the best of Golang's concurrency, our implementation properly divides the consolidated market graph into non-overlapping sections: each section is then fed to a go routine which can independently carry out all the weighted math computation to determine whether a given set of trades contains an arbitrage or not.

5.2.1. Benchmark CheckArbitrage

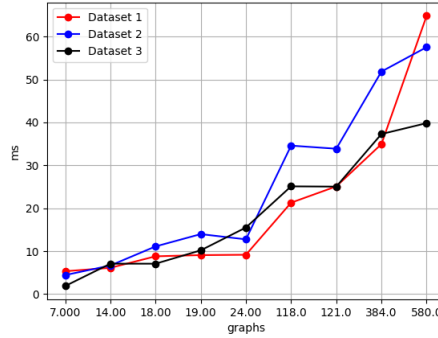


Figure 5.4: Shows the amount of time spent to check a number of cyclic paths searching for an arbitrage opportunity, using a fixed number of parallel processes.

5.2.2. Benchmark Sync

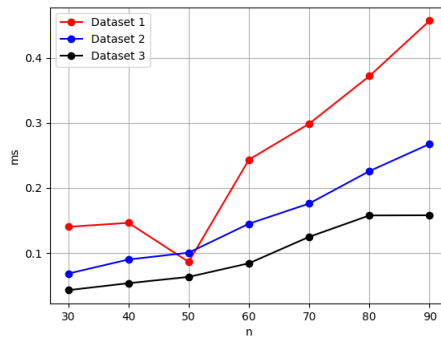


Figure 5.5: Shows the amount of time spent to populate every pool with the initial amount of tokens within, which we expect to be linear in the amount of pools by tokens within each pool.

6 | Conclusion and future works

The main scientific contributions in this paper are:

- A model built upon the standard to represent financial markets, expanding it to capture the fact that any two assets may be tradeable in more than one venue. This is represented by the concept of cyclic graphs on which we expanded in previous sections.
- A formalized generic architecture for these kind of systems, remarking what fundamental aspects should be considered when building your own. From this we designed the separation of a startup module to deal with the heavy graphing computation and a runtime core actively listening for new blocks, updating a local state and analyze potential profits.
- Analyzed our theoretical results by building an arbitrage-searching implementation as benchmark to measure our architectural pattern against real world constraints and challenges.

Using this paper as a starting point, some possible development on top of it could be:

- Uncle node detection and correction, based on a rollback of the inconsistent changes made from reading the uncled block's logs
- Supporting different connections to multiple, geo-distributed nodes, so as to minimize latency of viewing new blocks
- Efficient search for the optimal amount to begin an arbitrage with

Bibliography

- [1] : <https://uniswap.org/>
- [2] : <https://vitalik.ca/general/2017/06/22/marketmakers.html>
- [3] : <https://www.investopedia.com/terms/t/triangulararbitrage.asp>
- [4] : <https://ethereum.org/en/developers/docs/mev/>
- [5] : [https://thegraph.com/hosted-service/subgraph/balancer-labs/
balancer-v2](https://thegraph.com/hosted-service/subgraph/balancer-labs/balancer-v2)
- [6] : <https://arxiv.org/abs/1904.05234>
- [7] : [https://github.com/balancer-labs/balancer-v2-monorepo/blob/master/
pkg/pool-weighted/contracts/WeightedMath.sol](https://github.com/balancer-labs/balancer-v2-monorepo/blob/master/pkg/pool-weighted/contracts/WeightedMath.sol)

