Discovering Arbitrage Opportunities

Georgi Yovchev

June 29, 2022

1 Introduction

Arbitrage is an action where a trader buys an asset and immediately sells it for a higher value. For an arbitrage opportunity to exist, an asset must be sold on at least two unrelated markets. When someone announces that they are buying an asset for more money than it is being sold for in another market, an arbitrage opportunity arises.

Performing arbitrage in public centralized exchanges is extremely difficult due to the high dependence on intermediaries who always learn the information before you and are not obliged to announce it immediately. In contrast, many decentralized exchanges in the blockchain exist solely because of the presence of arbitrageurs who keep prices uniform across different exchanges. They are based on liquidity pools, which do not require people to declare a price but calculate the price through a mathematical formula, allowing instant liquidation by anyone.

In the event of a particularly large purchase, there may be a discrepancy between two exchanges, and then they rely on an arbitrageur who will solve the problem for personal profit. This document will not describe the process of performing arbitrage in decentralized exchanges, but will focus on the algorithmic part of finding them.

2 Discovering Arbitrage Opportunities

To discover an arbitrage opportunity, we assume that we have the current prices of assets in different exchanges. Let's say we have N assets. Each of these assets can be converted into another asset on one of the exchanges. If we represent this data in a graph where each vertex is an asset and all edges are conversion prices from one asset to another, we will get a complete graph.

In this graph, we will have several edges between each pair of vertices, representing the different trading prices in different exchanges. To discover arbitrage, we need to find a path where, through a sequence of conversions, we obtain more than the initial asset.

For example, in one exchange 1 BTC costs 10 BGN, in another 10 BGN is worth 5 EUR, and in a third exchange 5 EUR is worth 1.01 BTC. By performing these conversions, we will have gained 0.01 BTC. In this case, the price of BGN in BTC is 1/10, the price of EUR in BGN is 10/5, and the price of BTC in EUR is 5/1.01. Multiplying the prices, we get:

$$\frac{1}{10} \cdot \frac{10}{5} \cdot \frac{5}{1.01} = \frac{1}{1.01} < 1$$

To obtain arbitrage, the multiplication of the paths must be less than 1. Computers cannot work well with multiplication, so we need to reduce the process to the addition operation. This is easy because we know that:

$$\ln(x \cdot y) = \ln(x) + \ln(y)$$

This allows us to convert all edges v to $\ln(v)$, and at the end, we need to do the reverse - all edges v should become $\exp(v)$. To obtain arbitrage, the sum of the paths should be less than 0. Thus, we have reduced the problem to finding a negative cycle in a directed graph.

2.1 Finding a Negative Cycle in a Directed Graph

We know many algorithms for finding the shortest path in a graph. For example, the one by the well-known Dutchman - Edsger Dijkstra. It finds the length of the shortest paths from a vertex to all other vertices. It can be easily extended to find the paths themselves. The peculiarity with it is that, being a greedy algorithm, it only works with positive edges, and we need to work with negative ones.

The Bellman-Ford algorithm achieves the same results, working with negative edges, but with different complexity. For comparison - the complexity of Dijkstra's algorithm reaches $O(|E| + |V| \log |V|)$, while the complexity of Bellman-Ford reaches $O(|V| \cdot |E|)$, where V is the number of vertices, and E is the number of edges.

Here's what the pseudocode of an algorithm finding a negative cycle in a directed graph would look like:

```
for each vertex v in vertices:
    if v is source then distance[v] := 0
    else distance[v] := inf
    predecessor[v] := null

for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u

for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        return distance, predecessor

return null, null
```

2.2 Finding Multiple Negative Cycles in a Directed Graph

In such a graph, many negative cycles are expected. We will say that two cycles are interdependent when they have a common edge. With the above algorithm, we can easily discover several non-interdependent cycles and execute them simultaneously, but can we miss profit if we stop immediately after that?

When we execute one route, we have interacted with the market, which will most likely result in a change in prices. Our options for finding more cycles are two:

- Remove the edges we have interacted with and restart Bellman-Ford
- Synchronize the edges we have passed through with the new prices

3 Implementing Arbitrage on the Ethereum Blockchain

While the previous sections focused on the theoretical aspects of discovering arbitrage opportunities, this section will explore how to implement arbitrage on the Ethereum blockchain, specifically using decentralized exchanges (DEXs) like Uniswap.

3.1 Smart Contract for Arbitrage

To perform arbitrage on Ethereum, we need to create a smart contract that can interact with DEXs and execute trades atomically. Here's a basic structure of such a contract:

```
contract Arbitrager {

    address[] private routerAddresses;
    address[] private factoryAddresses;

    constructor() {
        routerAddresses.push(ADDRESS_1);
        // ...

        factoryAddresses.push(ADDRESS_2);
        // ...
    }

    function arbitrage(address[] calldata pairsPath, string[] calldata sym
        IPair[10] memory pairsPathF;
        for(uint i = 0; i < pairsPath.length; i++) {
            pairsPathF[i] = IPair(pairsPath[i]);
        }
        uint256 amountBefore = 0;
        IERC20 from;
        uint256 inAm = amountIn;
        for(uint i = 0; i < pairsPath.length; i++) {
            IRouter router = getRouter(pairsPathF[i]);
            address address0 = pairsPathF[i].token0();
            address address1 = pairsPathF[i].token1();
            IERC20 token0 = IERC20(address0);
            IERC20 token1 = IERC20(address1);
            address[] memory t = new address[](2);
            if (stringsEquals(token0.symbol(),symbolPath[i])) {
                if(amountBefore == uint256(0)) {
                    amountBefore = token1.balanceOf(address(this));
                    from = token1;
                }
                token1.approve(address(router), inAm);
                t[0] = address1;
                t[1] = address0;
                inAm = router.swapExactTokensForTokens(inAm, 0, t, address(
            } else if(stringsEquals(token1.symbol(),symbolPath[i])) {
                if(amountBefore == uint256(0)) {
                    amountBefore = token0.balanceOf(address(this));
                    from = token0;
                }
                token0.approve(address(router), inAm);
                t[0] = address0;
                t[1] = address1;
                inAm = router.swapExactTokensForTokens(inAm, 0, t, address(
            } else {
                require(false, "Invalid symbol input");
            }
        }
        require(from.balanceOf(address(this)) > amountBefore, "We lost mon
    }

    function stringsEquals(string memory s1, string memory s2) private pure
        bytes memory b1 = bytes(s1);
        bytes memory b2 = bytes(s2);
        uint256 l1 = b1.length;
        if (l1 != b2.length) return false;
        for (uint256 i=0; i<l1; i++) {
            if (b1[i] != b2[i]) return false;
        }
        return true;
    }

    function getRouter(IPair pair) view private returns (IRouter) {
        address factoryAddress = pair.factory();
        for (uint i = 0; i < factoryAddresses.length; i++) {
            if (factoryAddress == factoryAddresses[i]) {
                return IRouter(routerAddresses[i]);
            }
        }
        revert("Invalid factory address");
    }
}
```

This contract allows the owner to execute an arbitrage opportunity by specifying a path of token swaps and the initial amount to trade.

3.2 Mempool Monitoring for Arbitrage

One powerful technique for identifying and executing arbitrage opportunities on Ethereum is mempool monitoring. The mempool, short for memory pool, serves as a waiting area for transactions that have been submitted to the network but not yet included in a block. By closely observing the mempool, arbitrageurs can gain a significant advantage by identifying potential price discrepancies before they're reflected on-chain.

Mempool monitoring involves setting up a node to access the Ethereum network and continuously analyzing incoming transactions. This process allows traders to spot potential arbitrage opportunities in real-time, often before they become visible to the broader market.

The key steps in implementing a mempool monitoring system for arbitrage include:

1. Establishing a robust connection to the Ethereum network
2. Efficiently filtering and processing incoming transactions
3. Rapidly analyzing transactions for potential arbitrage opportunities
4. Executing trades swiftly when profitable situations are identified

To effectively monitor the mempool, arbitrageurs typically use specialized software that can process large volumes of transaction data in real-time. This software needs to be highly optimized for speed, as the window of opportunity for many arbitrage trades can be extremely short-lived.

The analysis of mempool transactions involves examining various factors, such as:

- The tokens involved in the transaction
- The size of the trade
- The exchange or protocol being used
- The current market prices across different exchanges

By combining this information, the monitoring system can identify transactions that are likely to create price discrepancies, which can then be exploited for profit.

However, mempool monitoring for arbitrage is not without its challenges. The process requires significant technical expertise and infrastructure. Traders need to ensure they have reliable, low-latency connections to Ethereum nodes to receive mempool data as quickly as possible. Additionally, the analysis of transactions must be extremely fast and accurate to beat competitors who are likely employing similar strategies.

Another consideration is the prevalence of false positives. Not all transactions that appear to create arbitrage opportunities will actually be mined into blocks. Arbitrageurs must account for this uncertainty in their strategies, often using probabilistic models to estimate the likelihood of a transaction being included in the next block.

Ethical considerations also come into play with mempool monitoring. While it can contribute to market efficiency by quickly correcting price discrepancies, there are concerns about the fairness of acting on information before it's publicly available on-chain. Moreover, arbitrageurs must be aware of the risk of their own transactions being front-run by other market participants.

Despite these challenges, mempool monitoring remains a powerful tool in the arsenal of many Ethereum arbitrageurs. When implemented effectively, it can provide a significant edge in identifying and capitalizing on market inefficiencies. However, success in this area requires not only technical prowess but also a deep understanding of the Ethereum ecosystem and market dynamics.

As the DeFi landscape continues to evolve, so too will the strategies and technologies used for mempool monitoring. Arbitrageurs must stay abreast of these developments to maintain their competitive edge in this fast-paced and complex market environment.

References

[1] Uniswap, "Uniswap V2 Whitepaper," 2020. [Online]. Available: `https://uniswap.org/whitepaper.pdf`

[2] Wikipedia contributors, "Graph (discrete mathematics)," Wikipedia, The Free Encyclopedia. [Online]. Available: `https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)`

[3] Wikipedia contributors, "Dijkstra's algorithm," Wikipedia, The Free Encyclopedia. [Online]. Available: `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`

[4] Wikipedia contributors, "Bellman–Ford algorithm," Wikipedia, The Free Encyclopedia. [Online]. Available: `https://en.wikipedia.org/wiki/Bellman-Ford_algorithm`