Arbitrage Cycle Search Heuristic

Project Tycho

September 24, 2025

Contents

1 Overview

The Tycho Searcher processes real-time blockchain data and identifies arbitrage opportunities. It receives block updates from a data feed, maintains an up-to-date graph of token exchanges, and applies a modified Bellman-Ford algorithm to detect negative cycles, which correspond to profitable arbitrage paths.

2 System Architecture

– **Tycho Feed:** An asynchronous task receives block updates and token data from an external source, forwarding them to the searcher.

– **Searcher:** Maintains the exchange graph, updates it with new data, and runs the arbitrage detection algorithm.

The code is written in rust. It uses asynchronous channels (`tokio::sync::mpsc`) to communicate between the feed and the searcher.

3 Searcher Workflow

1. **Initialization:** The searcher initializes its graph structures and awaits block updates.
2. **Block Update Handling:** On receiving a new block, the searcher updates the graph with the latest exchange rates and liquidity data.
3. **Arbitrage Detection:** The searcher runs the modified Bellman-Ford algorithm to find negative cycles, which indicate arbitrage opportunities.
4. **Result Export:** Detected opportunities are processed and can be exported or logged for further action.

4 Modified Bellman-Ford Algorithm

4.1 Purpose

The Bellman-Ford algorithm can detect negative cycles in a weighted graph. We have adapted it for arbitrage cycle searching, where the task is to find a cycle with positive profit in the exchange graph. We also consider that the internal state of an Automated Market Maker (AMM) changes after each exchange; therefore, we restrict arbitrage cycles to traverse each AMM at most once.

4.2 Data Structures

– `Graph`: A directed graph where nodes represent tokens and each edge corresponds to an AMM and represents the exchange rates between the corresponding tokens.

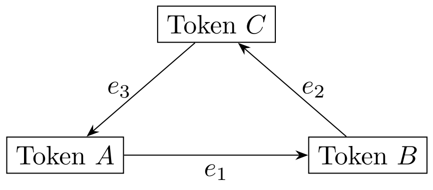
– `Edge Weights`: Each edge e is associated with two functions: $w_e(x_{\text{in}})$ and $g_e(x_{\text{in}})$. Here, $w_e(x_{\text{in}})$ returns the amount of tokens received after the swap, y_{out}, while $g_e(x_{\text{in}})$ returns the gas used, which is typically a constant function, since the gas cost is usually independent of x_{in}.

We assume that $w_e(x_{\text{in}})$ is a monotone non-decreasing function; that is, increasing the input amount x_{in} does not decrease the output amount y_{out}. Moreover, the exchange rate $\frac{y_{\text{out}}}{x_{\text{in}}}$ is a monotone non-increasing function of x_{in}.

Each query to $e()$ incurs a computational cost — in practice, this is implemented via a Rust call to `get_amount_out`, which emulates the swap. Both x_{in} and y_{out} are stored as large integer values.

Fig. 1 illustrates an example of an arbitrage cycle. An arbitrage is defined by a closed path in the (exchange) graph and an input amount x_{in}, which is denominated in the start token. The cycle begins and ends at the same token, referred to as the *start node* or *start token*.

The arbitrage cycle has two key performance metrics:



If the arbitrage cycle traverses tokens $A \to B \to C \to A$ then it is profitable if and only if

$$w_3(w_2(w_1(x_{\text{in}}))) - x_{\text{in}} > p_A \cdot (g_1(x_{\text{in}}) + g_2(x_{\text{in}}) + g_3(x_{\text{in}}))$$

Figure 1: An illustration of arbitrage cycle. p_A is the exchange rate for Token A in ether.

1. Gas cost – the sum of the gas used by each pool along the arbitrage path, including inter-pool transfers,
2. Profit – the net gain in the amount of the start token.

Since gas is paid in Ether, determining whether an arbitrage is profitable requires knowing the price p of the start token in ETH. If the start token is WETH, then $p = 1$.

It is important to note that the function $e()$ changes after each swap; therefore, Fig. 1 only holds if the three edges belong to different AMMs. Therefore, for each edge we explicitly store the pool to which it belongs.

– `Pools`: For each edge e we store the associated pool $p(e)$.

We restrict the search space to cycles in which every edge belongs to a different pool; that is, for any cycle C and any distinct edges $e, f \in C$, we require $p(e) \neq p(f)$. Although some studies do not impose this restriction on the problem space [1], we found that in practice it has only a minimal impact on solution quality.

4.3 Precomupations

As a first step, we decompose the graph into connected components so that cycles can be searched independently within each component. This has several advantages: upon an update, only the affected components need to be recomputed, and the search is also easily parallelizable (though we did not implement parallelization in our prototype).

We select a starting node from which the search begins, typically the WETH token. As a first step, we consider the 2-edge-connected component of the graph that contains the start node. Tokens that are either unreachable from the start node through AMMs, or reachable only via a single AMM, are irrelevant for the purpose of cycle detection. Next, we remove this node from the graph and compute the connected components. By adding the starting node back to the graph, we obtain subgraphs that are mutually disjoint apart from the starting node. It is easy to see that cycles can be searched independently in these subgraphs, because any cycle that would traverse multiple components

Table 1: Performance of the algorithm for pools of various minimum TVL values.

AMMs types	TVL	full graph #nodes	full graph #edges	#components	largest component #nodes	largest component #edges	runtime
Uniswap v2,3,4 Ekubo Balancer curve	1000	111	322	54	47	190	4.2 s
	100	623	1662	435	166	574	12 s
	10	2467	6164	1980	462	1924	21 s
Uniswap v2, v3	1000	78	202	42	26	90	40 ms
	100	563	1324	420	126	398	271 ms
	10	2375	5470	1973	367	1258	359 ms

could only cross via the starting node. In such a case, however, the cycle can be split into smaller cycles that each remain within a single component.

Table 1 reports the graph size, the number of connected components, and the size of the largest component, as a function of the set of pools under consideration. The pools include Uniswap v2, v3, and v4, Ekubo v2, Balancer v2, and Curve. We further filter pools by their total value locked (TVL), measured in ether. The table shows that this step yields a significant speedup. For instance, in a graph with 2 375 nodes, the largest component contains only 367 nodes, illustrating how special the structure of the graph is.

See also Fig. 2 showing the graph skeleton of the largest component, which is a simplified representation produced by first removing self-loops and iteratively pruning leaves (degree-1 nodes), then collapsing degree-2 chains into single edges marked as skeleton (and mark these edges as dashed lines). These transfromations preserves the graph for cycle search.

4.4 Modified Belmann-Ford

Next we explain our most important subrutine. In this subrutine we assume x_{in} is given, and a subgraph $G = (V, E)$. The task is to find candidate cycles such that are profitable, i.e. $w_3(w_2(w_1(x_{\text{in}}))) > x_{\text{in}}$ on Fig. 1.

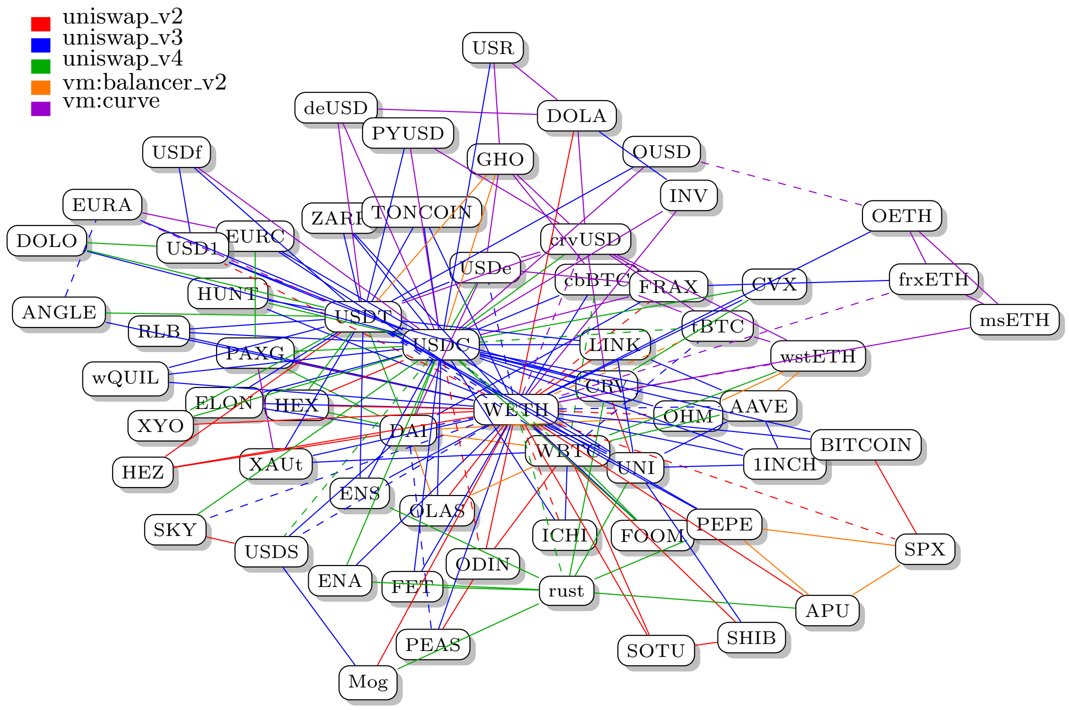
We define the following data structure

– `Distance_without_loop`: For each node v, we store the maximum attainable output amount y_{out} (denoted as d_v) along a path that does not traverse the same node (token) more than one.

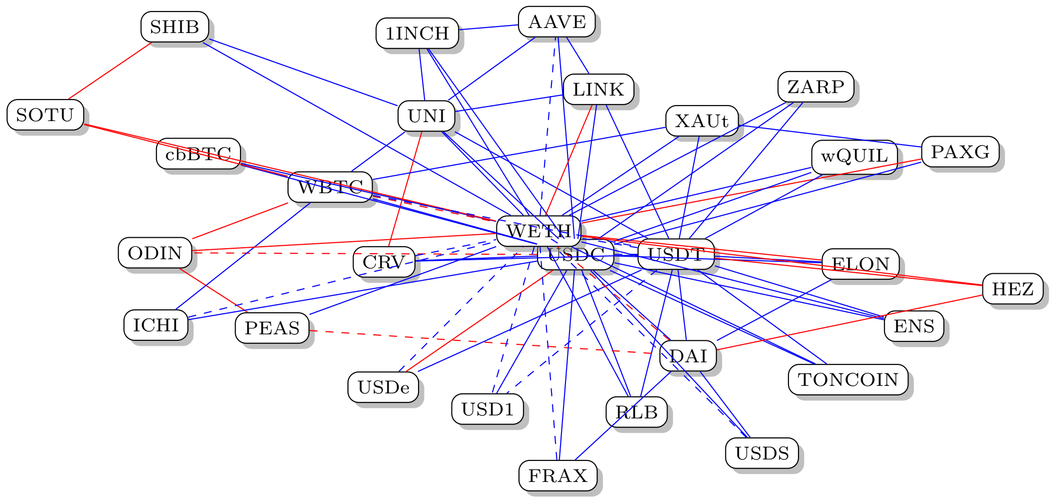
– `Path_without_loop`: For each node v, we store the path P_v towards the start node corresponding to d_v.

– `Distance_with_a_single_loop`: For each node v, we store a second value, the maximum attainable output amount y_{out} (denoted as d_v^1) along a path that traverses one node (token) twice, all the others only once.

– `Path_with_a_single_loop`: For each node v, the corresponding path P_v^1 from the start node.



(a) All pools



(b) Uniswap v2 and v4 pools

Figure 2: The sceleton of the token graphs for pools with TVL≥ 10 Ether, at 22nd of September 2025. Graph skeleton: the core network obtained by removing self-loops and peripheral leaves and collapsing chains of degree-2 nodes into single (dashed) edges, so the backbone connectivity between hubs is emphasized.

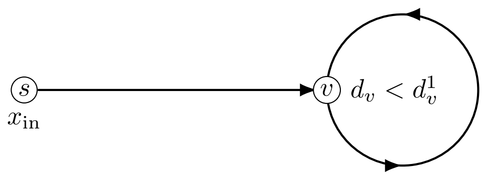


Figure 3: Illustration of an arbitrage path saved in `save_path`.

Algorithm 1 describes the pseudocode of the algorithm. It have the following steps:

1. Initialize distances from the source node to all other nodes as infinity, except the source itself (zero and empty path).
2. For each edge $u \to v$, if node u has a finite distance (i.e., $d_u \neq \infty$), attempt to extend the path P_u by adding the edge. See function `extend_path` for details. In our approach, this is allowed only if the pool of $e_{u\to v}$ has not previously been used in the path. If this results in more tokens than currently recorded at d_v, then we need to update our data structures as follows: If the path P_u dos not traverse token v, we update d_v and P_v, otherwise we update d_v^1 and P_v^1. In the later case we have found a path from s and with a loop through v, which we save as a candidate. Finally, if node $v = s$, we have found a loop.
3. We perform a similar check if node u has a finite distance with loop (i.e., $d_u \neq \infty$). In this case we don deal witht he case of having more loops.
4. Repeat the previous step for k iterations, where k is the maximum allowed path length.

The function `extend_path`$(d_u, g_u, P_u, e_{u\to v})$ returns the token amount and gas usage resulting from extending path P_u with edge $e_{u\to v}$. Note that, it could handle the case where the same AMM is traversed multiple times along $P_u + e_{u\to v}$, updating its internal state after each traversal.

The function `save_cycle`(x_{in}, P) saves a profitable arbitrage cycle with a given amount x_{in}. While the function `save_path`(x_{in}, P) saves a profitable path to a token, from which there is an arbitrage cycle. It also stores an amount x_{in} from the start node. See Fig. 3 for illustration.

4.5 Arbitrage search

Next, we discuss how to search for the optimal input amount x_{in}. We begin by running Algorithm 1 with $p = 0$ and a small initial input, e.g., $x_{\text{in}} = 0.001$, which typically yields multiple arbitrage cycles. For each identified cycle, we apply a black-box optimization method—Golden Section Search (GSS)—to find the optimal x_{in}.

GSS is a derivative-free optimization technique for finding the maximum (or minimum) of a unimodal function over a bounded interval. It is particularly suitable when the function's analytic form is unknown or too expensive to compute, but function evaluations are available. GSS progressively narrows the interval of interest using the golden ratio, requiring only function values at strategically chosen points. In our setting, GSS is used to identify the input amount x_{in} that maximizes arbitrage profit along a given cycle.

4.6 Optimizations

To reduce computation time, we divide the graph into subgraphs and solve the problem independently for each. To compute the subgraphs, we remove the start node s and identify the resulting connected components. Each subgraph is then formed by reintroducing the start node s and connecting it to a component. The algorithm is executed independently within each subgraph.

To further reduce runtime, we implemented the following memory management strategy.

– **Reference Counting:** By using `Rc<Vec<EdgeIndex>>`, the algorithm avoids deep cloning of cycle paths, which is especially beneficial when many cycles are detected.

– **Efficient Graph Updates:** Only relevant subgraphs are updated per block, reducing recomputation.

– **BigUint Arithmetic:** Ensures that all calculations are safe from overflow, which is critical for financial computations.

4.7 Summary:

We have presented a Bellman-Ford algorithm with the following modifications:

- Cycles are only considered if they start and end at the designated start token (e.g., WETH).
- Reference-counted vectors (`Rc<Vec<EdgeIndex>>`) are used to avoid unnecessary cloning.
- The algorithm skips cycles that revisit nodes to avoid infinite loops.
- Profit calculation includes gas costs, and only cycles with net positive profit are exported.
- The golden section search adapts the input range if invalid points are encountered.

References

[1] Diamandis, T., Angeris, G., Edelman, A.: Convex network flows. arXiv preprint arXiv:2404.00765 (2024)

Algorithm 1: Modified Belmman-Ford

```
   Input: Directed graph $G$, strat node $s$, amount $x_{\text{in}}$, price $p$, max
          iteration $k_{\max}$
1  $d_s = x_{\text{in}}$; $g_s = 0$; $d_v = \infty$ and $g_v = 0$ for $v \in V \setminus \{s\}$
2  for $j \in \{1, \ldots, k_{\max}\}$ do
3      for $e_{u \to v} \in E$ do
4          if $d_u \neq \infty$ and pool of $e_{u \to v}$ is not in $P_u$ then
5              $d', g'$ = extend_path($d_u, g_u, P_u, e_{u \to v}$);
6              if $v \neq s$ then
7                  if token $v \in P_u$ then
8                      if $d_v^1 + p \cdot g_v^1 > d' + p \cdot g'$ then
9                          $d_v^1 = d'$; $g_v^1 = g'$; $P_v^1 = P_u + e_{u \to v}$;
10                         save_path($x_{\text{in}}, P_u + e_{u \to v}$);
                   else
11                     if $d_v + p \cdot g_v > d' + p \cdot g'$ then
12                         $d_v = d'$; $g_v = g'$; $P_v = P_u + e_{u \to v}$;
               else
13                 if $d' > x_{in} + p \cdot g'$ then
14                     save_cycle($x_{\text{in}}, P_u + e_{u \to v}$);
15         if $d_u^1 \neq \infty$ and pool of $e_{u \to v}$ is not in $P_u^1$ then
16             $d', g'$ = extend_path($d_u^1, g_u^1, P_u^1, e_{u \to v}$);
17             if $v \neq s$ then
18                 if token $v \notin P_u^1$ then
19                     if $d_v^1 + p \cdot g_v^1 > d' + p \cdot g'$ then
20                         $d_v^1 = d'$; $g_v^1 = g'$; $P_v^1 = P_u^1 + e_{u \to v}$;
               else
21                 if $d' > x_{in} + p \cdot g'$ then
22                     save_cycle($x_{\text{in}}, P_u^1 + e_{u \to v}$);
```