Hermes: Scalable and Robust Structure-Aware Optimal Routing for Decentralized Exchanges

Soroush Farokhnia, Sergei Novozhilov, Sanaz Safaei, Jiasi Shen

Hermes: Scalable and Robust Structure-Aware Optimal Routing for Decentralized Exchanges

Soroush Farokhnia*, Sergei Novozhilov*, Sanaz Safaei†, Jiasi Shen*
*The Hong Kong University of Science and Technology, Hong Kong
{sfarokhnia, snovozhilov}@connect.ust.hk, sjs@cse.ust.hk
†Digikala, Iran
sanaz.safaei@digikala.com

***Abstract*—Decentralized exchanges (DEXs) have transformed the financial landscape by enabling transparent, permissionless token trading on blockchains. These platforms rely on smart contracts called liquidity pools. Each pool allows for the trading of two tokens, with the exchange price dynamically calculated by an automated algorithm based on the available liquidity. Uniswap, the leading DEX on Ethereum, features over 400,000 pools and tokens, supporting an average daily trading volume of $1.3 billion USD since 2024. However, fragmentation across DEXs and the rapid growth in the number of tokens significantly complicate the search for optimal exchange rates, particularly when no direct trading pair exists. As a result, it often requires a sequence of trades across multiple pools, a challenge known as *routing*. Routing is typically modeled as a shortest path problem on a given graph of tokens. Existing algorithms have significant drawbacks: scalable approaches often lack guarantees for route validity, while robust methods struggle with the scale and dynamic nature of modern decentralized exchanges.**

In this work, we address the problem of optimal routing on DEXs. We demonstrate that by leveraging the structural properties of this graph, in particular its *treewidth*, it is possible to reconcile scalability with robustness. On the theoretical side, we adapt a parameterized algorithm utilizing treewidth to handle the dynamic setting of DEXs, where pools frequently change. We show that our approach achieves improved time complexity over existing methods and additionally provides a formal guarantee on the quality of the computed routes. We present empirical analysis on real Uniswap data to demonstrate the suitability of a parameterized online algorithm. Furthermore, we have implemented this algorithm in a free and open-source tool called Hermes and compared it with existing methods. On small instances where both tools produced results, Hermes reduced the average runtime by four orders of magnitude, from 2.81 seconds to 0.0002 seconds. Notably, Hermes is the only tool capable of computing routes in the presence of 100,000 tokens. It achieves an average runtime of 0.19 seconds, while other approaches fail to complete within the allotted time.

***Index Terms*—Ethereum, Decentralized Exchanges, Uniswap, Routing, Single-Source Shortest Path (SSSP), Parameterized Algorithms, Tree Decomposition**

1. Introduction

Decentralized Finance (DeFi). Smart contracts are blockchain-based programs that manage, transfer, and control digital assets without intermediaries. This significant advancement has led to the use of smart contracts in developing a new ecosystem that enables advanced, composable financial functions beyond basic token transfers on the blockchain, now collectively referred to as *decentralized finance* (DeFi) [1]. DeFi has grown rapidly due to its transparent, trustless, and programmable financial services. Currently, the total value locked (TVL)[1] in DeFi on Ethereum alone exceeds 65.9 billion USD [2], [3].[2]

Decentralized Exchanges (DEX). Decentralized exchanges (DEXs) are a cornerstone of the DeFi ecosystem, enabling the permissionless trading of digital assets. Among the different DEX types, *Automated Market Makers* (AMMs) have emerged as the most widely adopted by nearly every metric [4]. Central to AMMs is the concept of a *liquidity pool*: a smart contract that holds reserves of two (or more) tokens, enabling users to trade between them without intermediaries. The exchange rate between two tokens within a liquidity pool is autonomously determined by a function of the current reserves. More details are provided in Section 3. Uniswap is the leading DEX by TVL on Ethereum. Since 2024, Uniswap has hosted an average daily trading volume of 1.3 billion USD, with an all-time cumulative trading volume reaching 2.26 trillion USD [2].

Routing in Decentralized Exchanges. A fundamental challenge within the DEX ecosystem is discovering optimal exchange rates, a problem necessitated by the decentralized and fragmented liquidity landscape. Often, a direct trading pair between two assets does not exist, requiring a trader to perform a sequence of trades across several liquidity pools, a process known as *routing*. Routing can be modeled as a graph problem on $G = (V, E, w)$, where the vertices

1. Total value locked refers to the aggregate value of all assets deposited in smart contracts.

2. All prices in this paper are reported as of 15-06-2025.

V represent tokens, the edges E correspond to liquidity pools connecting pairs of tokens, and the weights w encode the spot exchange rates available for trading between those tokens. A route is a sequence of tokens whose cumulative edge weights determine the final exchange rate. The goal is to find the route between two tokens that yields the best price. See Section 3 for details.

A well-known special case of routing is *arbitrage*, where a trader earns risk-free profit by exploiting price discrepancies for the same asset across different pools. This problem is commonly modeled as finding a negative-weight cycle in the aforementioned graph [1], [4]. Arbitrages have been studied exhaustively in the literature on DEX due to their risk-free profits [3]. However, the reported arbitrages constitute only a small fraction of trades, accounting for approximately 11.71% of daily volume on Uniswap [2], [5].

Algorithmic Challenges in Routing. While financial arbitrage has been extensively studied in both traditional and decentralized markets [3], [4], research focused on optimal routing within decentralized exchanges remains comparatively limited. The existing body of work reveals two primary algorithmic challenges. First, our study shows that current routing algorithms scale poorly as the number of assets increases. This issue is particularly acute given the operational constraints of modern blockchains, such as Ethereum's 12-second block time, and the dynamic nature of the ecosystem, which features continuously changing prices. Second, the presence of arbitrage opportunities, which manifest as negative-weight cycles in the graph-based problem formulation, renders standard shortest-path algorithms such as Bellman-Ford incapable of finding optimal routes. This situation has led to a dichotomy in existing solutions: they are either general-purpose algorithms that guarantee optimality but are often too slow for practical on-chain execution, or they rely on heuristics that are faster but offer no formal guarantees on the quality of the route.

Our Approach: Parameterized Algorithms. In this work, we bridge this gap by designing a *parameterized algorithm* tailored to the structural properties of DEX transaction graphs [6]. The design of such algorithms follows a paradigm that diverges from traditional runtime analysis. Instead of measuring performance solely against the input size, a parameterized algorithm's efficiency is also analyzed with respect to an additional structural property of the input, known as a *parameter*. For graphs, a common and powerful parameter is *treewidth*, which measures how closely a graph resembles a tree. Our central thesis is that real-world DEX transaction graphs exhibit low treewidth. By exploiting this structural property, we have developed an algorithm that is both *theoretically sound*, due to its formal complexity analysis, and *practically efficient*, a claim we substantiate through extensive experiments on real-world DEX data. This approach strikes a crucial balance, achieving provable optimality without sacrificing the performance required in the demanding DEX environment.

Our Contributions. In this paper, we analyze the structural properties of Uniswap exchange pools. Our main contributions are as follows:

- We conduct, to our knowledge, the first large-scale empirical study of the parameters of real-world DEX graphs. By analyzing data from Uniswap, we demonstrate that these graphs exhibit low treewidth.
- We extend a parameterized algorithm that leverages the low treewidth of DEX graphs to find optimal routes efficiently. The algorithm is specifically tailored for the dynamic, online environment of DEXs, where liquidity pools are constantly changing. Our formal complexity analysis shows that its theoretical runtime outperforms state-of-the-art algorithms.
- We implement the algorithm in an open-source routing tool called Hermes and conduct a comprehensive experimental evaluation using real-world Uniswap transaction data covering $20{,}000$ blocks. The results show that our treewidth-based approach achieves superior practical performance compared to existing methods. Hermes is the only method capable of processing token sets of size $100{,}000$, with an average query time of **0.19** seconds. For instances where both tools yield results, Hermes reduces the average runtime from **2.81** seconds to **0.0002** seconds, an improvement of over four orders of magnitude.

2. Related Work

Arbitrages. The majority of the literature on routing in DEXs focuses on identifying arbitrage opportunities. In such scenarios, trader exploits price discrepancies across trading pools to generate profit. Recent studies systematically analyze historical trade data, uncovering millions of executed arbitrage cases through the detection of cyclic trading patterns [1], [7]. Consequently, a substantial body of literature models the arbitrage problem as a graph and applies graph-based algorithms to identify negative cycles that signal profitable arbitrage opportunities. Researchers have employed algorithms such as Johnson's algorithm [1] and modified variants of Bellman-Ford [4], [8] for this purpose. Beyond graph-based methods, some researchers have approached arbitrage detection as a convex optimization problem [9], while others have utilized artificial intelligence techniques, such as Graph Neural Networks (GNNs), for this purpose [10].

Routing. In contrast to arbitrage detection, there are only two notable studies on routing in DEXs. Diamandis et al. [11] formulate routing as a convex optimization problem and employ decomposition techniques together with commercial solvers. Zhang et al. [4] introduce the Modified Moore-Bellman-Ford algorithm, which adapts the standard

Bellman-Ford algorithm to produce valid routes even in the presence of negative cycles. They further improve their approach by using line graphs to enhance practical performance. While their work mainly targets arbitrage detection, it also finds shorter routes between tokens. However, this method does not scale to thousands of tokens, leaving robust and scalable routing algorithms as an open challenge.

Parameterized Algorithms in Blockchain. Parameterized algorithms have been applied to blockchain problems by exploiting structural properties of underlying graphs. For example, NP-hard problems like optimal block production become tractable on instances with low pathwidth or treedepth [12], [13]. Similarly, smart contract analysis benefits from the low treewidth of control flow graphs [14]. To our knowledge, this is the first work to apply such techniques to decentralized exchanges.

3. Preliminaries

Automated Market Makers. An *Automated Market Maker* (AMM) is a protocol for decentralized trading that replaces traditional order books with deterministic pricing algorithms [1]. In an AMM, prices are set by a mathematical function of the current token reserves in each liquidity pool. The most widely used class of AMMs is the *Constant Function Market Maker* (CFMM), of which Uniswap is a prominent example. In Uniswap V2, the CFMM uses the constant product formula, which maintains a constant product of the reserves of two tokens u and v in a liquidity pool: $R_u \cdot R_v = k$, where k is a fixed constant. Here, R_u and R_v represent the current quantities (reserves) of tokens u and v held in the pool, respectively. This means that any trade will adjust the reserves of both tokens while keeping their product constant. The spot price from token u to token v is determined by the ratio of their reserves, specifically $\text{price}(u \to v) = \frac{R_v}{R_u}$. Uniswap V3 allows liquidity providers to concentrate capital within custom price ranges, increasing efficiency. Uniswap V4 introduces *hooks* for customizable trading strategies and fee structures [15]. All three versions are active on Ethereum.

For the scope of this paper, these versions share two pricing notions. The first is the *spot price*, which depends on the current reserves and indicates the current market price. The second is the *trade price*, which also accounts for the size of the trade. The applications of both concepts have been thoroughly studied in the literature [3]. In this work, we focus on spot prices.

Modeling the DEX Ecosystem as a Graph. To analyze and optimize trade routing across a decentralized exchange ecosystem, we employ the abstraction of a weighted directed graph $G = (V, E, w)$. In this model, the set of vertices V represents the universe of available tokens. A directed edge $(u, v) \in E$ exists if there is a liquidity pool capable of direct trading from token u to token v. A path in this graph, which is a sequence of connected vertices, therefore corresponds to a multi-hop trade that converts a source token into a destination token through one or more intermediaries.

Shortest Path Formulation. The objective in trade routing is to find a path that maximizes the product of exchange rates[3]. Since standard shortest path algorithms are designed to minimize a sum of weights, we align our problem by defining the weight of an edge (u, v) as the negative logarithm of the exchange rate: $w(u, v) = -\log(\text{price}(u \to v))$. In the case of multiple liquidity pools between two tokens, we choose one pool maximizing the exchange rate $\max \text{price}(u \to v)$, thus avoiding the necessity of considering the multi-edge graphs. This weight assignment converts the maximization of a product into the minimization of a sum, thereby reformulating the optimal routing task as a Single-Source Shortest Path (SSSP) problem. A critical consideration in this model is the presence of negative-weight cycles, which correspond to arbitrage opportunities: trading paths that start and end with the same token and yield a profit. Such cycles can lead to infinitely cheap paths, a complication that must be handled by the chosen algorithm.

Graph Notation and Terminology. To develop an algorithm that leverages the specific topology of the DEX graph, we must first establish a formal language for discussing its local properties. We define the *neighborhood* of a vertex v, denoted $N(v)$, as the set of all vertices directly accessible from it $N(v) := \{u \mid (v, u) \in E\}$. Furthermore, for any subset of vertices $S \subseteq V$, the *subgraph induced by S*, denoted $G[S]$, consists of the vertices in S and all edges from the original graph that connect any two vertices in S: $G[S] := (S, E \cap (S \times S))$.

Tree Decompositions and Treewidth. The efficiency of our algorithm hinges on a structural parameter known as *treewidth*, which quantifies how "tree-like" a graph is. Formally, the treewidth of a graph $G = (V, E)$ is defined via a *tree decomposition*. A tree decomposition is a pair $(\mathcal{T}, \{X_i \mid i \in I\})$, where $\mathcal{T} = (I, F)$ is a tree and each X_i (called a bag) is a subset of V, satisfying:

1) The union of all bags equals V; i.e., $\bigcup_{i \in I} X_i = V$.
2) For every edge $(u, v) \in E$, there is at least one bag X_i containing both u and v.
3) For any vertex $v \in V$, the set of bags containing v forms a connected subtree in $\mathcal{T}$.

The *width* of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G, denoted $tw(G)$, is the minimum width over all possible tree decompositions of G. A low treewidth indicates a structure amenable to efficient dynamic programming, a property we exploit extensively.

Chordal Graphs and Chordal Completion. Our algorithm operates not on the original graph G, but on a *chordal supergraph* of it. A graph is chordal if every cycle of

3. We assume rates are adjusted for all transaction fees.

four or more vertices has an edge connecting two non-consecutive vertices (a chord). While most graphs are not chordal, any graph G can be transformed into one by adding a set of "fill-in" edges to create a *chordal completion*, $\widehat{G}$. A key result in graph theory states that the treewidth of a graph is intrinsically linked to its best chordal completion: $tw(G) = \min_{\widehat{G}} \omega(\widehat{G}) - 1$, where the minimum is taken over all chordal completions of G and $\omega(\widehat{G})$ is the size of the largest clique in $\widehat{G}$. This relationship is fundamental to our approach.

All-Pairs Shortest Paths via Directed Path Consistency. The core of our algorithm is an efficient method for solving the All-Pairs Shortest Paths (APSP) problem on graphs with low treewidth. The method relies on enforcing *Directed Path Consistency* (DPC) using a PEO. A weighted graph is DPC with respect to an ordering $\pi = (v_1, \ldots, v_n)$ if for every triple of vertices v_i, v_j, v_k with $i < j < k$, the path cost from v_i to v_k is no greater than the cost of the path through v_j; i.e., $w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$.

Enforcing this property on an arbitrary graph with an arbitrary ordering takes $O(n^3)$ time. However, by using the PEO of a chordal completion $\widehat{G}$, the process can be dramatically accelerated. The algorithm iterates backward through the PEO (from v_n to v_1), and at each step v_k, it only needs to check for path updates between pairs of neighbors of v_k that appear earlier in the ordering. Because a PEO ensures these neighbors form a clique of size at most $\omega(\widehat{G})$, the complexity of enforcing DPC is reduced to $O(n \cdot \omega(\widehat{G})^2)$, which is equivalent to $O(n \cdot tw(G)^2)$.

The advantage of the DPC property is that all shortest paths become bitonic: any shortest path from u to v consists of a segment where vertices decrease in the PEO order, followed by a segment where they increase. This structure allows SSSP queries to be answered efficiently in two passes over the PEO array, see Algorithm 4 for details.

4. Routing Algorithm

4.1. The Algorithm Overview

Our proposed algorithm transforms a general graph into a structure where shortest path queries can be answered with remarkable efficiency. This transformation is achieved through a multi-phase process, illustrated in Figure 1. The first three phases constitute an offline preprocessing stage, which is performed only once. The final phase is the online query stage, designed for rapid, repeated execution.

The main idea is to construct a chordal supergraph leveraging graph's tree decomposition. The chordal graph admits a Perfect Elimination Ordering (PEO). This ordering is then used to enforce Directed Path Consistency (DPC). The DPC property is used to answer the Single-Source Shortest Path (SSSP) queries efficiently. The algorithm is designed to handle graphs with negative-weight cycles, ensuring that the results are correct even in the presence of such cycles. The entire process is summarized in Algorithm 1.

Algorithm 1 Unified Algorithm for Treewidth-Based Batch SSSP Queries

Input: A weighted graph $G = (V, E, d)$, a set of source vertices $Q = \{s_1, \ldots, s_q\}$.
Output: A list of SSSP distance arrays $(D[s_1], \ldots, D[s_q])$, where each $D[s_i]$ contains the distances from source s_i to every vertex in V.

```
# — Structural Preprocessing —
1: T ← ComputeTreeDecomposition(G)
2: (Ĝ, π) ← ComputeCompletionAndPEO(T)
# — Weights Preprocessing —
3: d* ← EnforceDPC(Ĝ, π, d)
# — Process each SSSP query —
4: for i ← 1 to q do
5:     D[s_i] ← QuerySSSP(Ĝ, d*, s_i)
6: end for
```

The correctness and efficiency of this framework are formalized in Theorem 1.

Theorem 1 (Complexity and Correctness)**.** *Let $G = (V, E, d)$ be a weighted graph with $n = |V|$ vertices and treewidth $tw(G)$. The algorithm described in Algorithm 1 has an overall preprocessing time of $O(n \cdot tw(G)^2)$ and answers each SSSP query in $O(n \cdot tw(G)^2)$ time.*

For each source vertex $s_i \in Q$, the resulting distance array D_i has the following properties for any target vertex $t \in V$:

- *If there are no negative-weight cycles on any path from s_i to t, then $D[s_i][t]$ stores the correct shortest path distance from s_i to t in the original graph G.*
- *If there is a negative-weight cycle on a path from s_i to t, the corresponding distance $D[s_i][t]$ will be less or equal to the shortest simple path (a path with no repeated vertices) distance from s_i to t in G.*

Phase 1: Tree Decomposition. This initial phase (line 1 of Algorithm 1) computes a tree decomposition for the input graph G. The goal is to find a low-width decomposition, as its width, the treewidth $tw(G)$, is the primary parameter governing the complexity of the entire preprocessing stage.

A landmark result in parameterized complexity theory by Bodlaender proves that it is Fixed-Parameter Tractable (FPT).

Lemma 1 (Complexity of Tree Decomposition)**.** *For any fixed parameter k, there exists a linear-time algorithm that can determine if a graph G has treewidth at most k and, if so, produce a corresponding tree decomposition in $O(f(k) \cdot |V(G)|)$ time. [16].*

Additionally, many heuristic approaches have been developed. These heuristics have been refined and implemented in highly optimized solvers. Modern, competitive

solvers can often find low-width decompositions for very large graphs arising in practice [17].

Phase 2: Fill-in and PEO. In this phase (line 2 of Algorithm 1), we construct a minimal chordal completion $\widehat{G}$ and a corresponding Perfect Elimination Ordering (PEO) π from the tree decomposition. The process, detailed in Algorithm 2, iterates through the tree bags from leaves to the root. In each step, it turns the current leaf bag into a clique and prepends its unique vertices (those not in its parent bag and have not been seen before) to the PEO.

Algorithm 2 ComputeCompletionAndPEO($G = (V, E), \mathcal{T}$)

1: $\widehat{G} \leftarrow G$, $\pi \leftarrow ()$
2: **while** $\mathcal{T}$ is not empty **do**
3: $X \leftarrow$ leaf($\mathcal{T}$)
4: $Y \leftarrow$ parent($\mathcal{T}, X$) or $\emptyset$ if X is the root
5: $\widehat{E} \leftarrow \widehat{E} \cup \{(u, v) \mid u, v \in X, u \neq v\}$
6: $\pi \leftarrow [(X \setminus Y) \setminus \pi] + \pi$
7: $\mathcal{T} \leftarrow \mathcal{T} \setminus \{X\}$
8: **end while**
9: **return** $(\widehat{G}, \pi)$

We summarize the key results about the correctness and complexity below.

Lemma 2 (Properties of the Completion and PEO)**.** *Let $\widehat{G}$ and π be the outputs of Algorithm 2. The following properties hold:*

(a) The graph $\widehat{G}$ is a chordal supergraph of G.
(b) The clique number of the completion satisfies $\omega(\widehat{G}) = tw(G) + 1$.
(c) The sequence π is a valid Perfect Elimination Ordering for $\widehat{G}$.
(d) The algorithm runs in $O(n \cdot (tw(G) + 1)^2)$ time.

Proof (sketch). For *(a)-(c)*, we refer to the standard properties relating tree decompositions to chordal graphs [18]. For (d), we analyze the algorithm as follows: we can assume the given tree decomposition has $O(n)$ nodes. The algorithm iterates through each node, where the dominant operation is making the bag a clique (line 5). This step takes $O((tw(G) + 1)^2) = O(tw(G)^2)$ time. The total complexity is therefore $O(n \cdot (tw(G) + 1)^2)$. □

Phase 3: Enforcing the DPC. This phase executes the 'EnforceDPC' function (Algorithm 3), which corresponds to line 3 of the main algorithm. It takes the chordal graph $\widehat{G}$, its PEO π, and the original distances d to produce a new distance matrix d^*.

This procedure does not compute the final all-pairs shortest paths. Instead, it modifies the edge weights to satisfy the Directed Path Consistency (DPC) property relative to the PEO π. Specifically, after 'EnforceDPC' terminates, for any path $v_i \to v_k \to v_j$ where v_k is a common neighbor of v_i and v_j that appears later in the PEO (i.e., $i, j < k$), the triangle inequality $d^*(v_i, v_j) \leq d^*(v_i, v_k) + d^*(v_k, v_j)$ is guaranteed to hold.

Algorithm 3 EnforceDPC($\widehat{G} = (V, \widehat{E}), \pi, d$)

1: $d^*(u, v) \leftarrow d(u, v)$ for all $(u, v) \in E$ and $d^*(u, v) \leftarrow \infty$ for all $(u, v) \in \widehat{E} \setminus E$.
2: $\pi = (v_1, v_2, \ldots, v_n)$ is the PEO of $\widehat{G}$.
3: **for** $k \leftarrow n$ **down to** 1 **do**
4: **for** each pair of neighbors v_i, v_j of v_k in $\widehat{G}$ with $i < k$ and $j < k$ **do**
5: $d^*(v_i, v_j) \leftarrow \min(d^*(v_i, v_j), d^*(v_i, v_k) + d^*(v_k, v_j))$
6: **end for**
7: **end for**
8: **return** d^*

Lemma 3 (Complexity of Enforcing DPC)**.** *The 'EnforceDPC' procedure (Algorithm 3), which applies Directed Path Consistency on the chordal graph $\widehat{G}$ using its PEO, has a time complexity of $O(n \cdot tw(G)^2)$.*

Phase 4: SSSP Queries. The final query phase (lines 4–5) processes the batch of SSSP queries. For each source s_i in the query set Q, the algorithm calls the QuerySSSP procedure (Algorithm 4). The algorithm is an adaptation of the `Min-path` procedure [19], [20].

Algorithm 4 QuerySSSP($\widehat{G} = (V, \widehat{E}), \pi, d^*, s$)

1: Initialize distance array $D[v] \leftarrow \infty$ for all $v \in V \setminus \{s\}$, $D[s] \leftarrow 0$.
2: $D[s] \leftarrow 0$.
3: Let $\pi = (v_1, \ldots, v_n)$ be the PEO. Let $s = v_{idx}$.
4: **for** $k \leftarrow idx$ **down to** 1 **do**
5: **for** each neighbor v_j of v_k in $\widehat{G}$ such that $j < k$ **do**
6: $D[v_j] \leftarrow \min(D[v_j], D[v_k] + d^*(v_k, v_j))$
7: **end for**
8: **end for**
9: **for** $k \leftarrow 1$ **to** n **do**
10: **for** each neighbor v_j of v_k in $\widehat{G}$ such that $j > k$ **do**
11: $D[v_j] \leftarrow \min(D[v_j], D[v_k] + d^*(v_k, v_j))$
12: **end for**
13: **end for**
14: **return** D

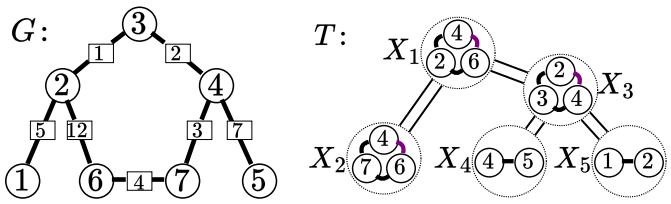
Lemma 4 (Complexity of SSSP Query)**.** *The* QuerySSSP *procedure has a complexity of $O(|\widehat{E}|)$, which in a chordal graph is bounded by $O(n \cdot tw(G)^2)$.*

4.2. Handling Dynamic Edge Additions

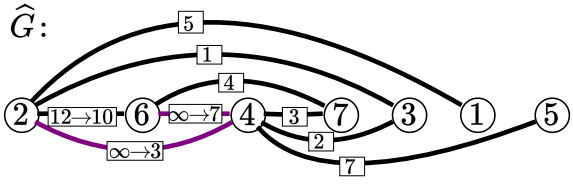
Real-world graphs are often dynamic. In our context, new liquidity pools can be added, creating new edges in the graph. A key advantage of our framework is its ability to handle many such updates efficiently. If a new edge (u, v) already exists within the pre-computed chordal completion

$\widehat{G}$, the underlying structure remains valid. In this scenario, we can bypass the expensive structural preprocessing steps (lines 1-2 of Algorithm 1). Instead, we only need to update the edge weights and re-run the weights preprocessing (line 3). If the new edge is not in $\widehat{G}$, the full pipeline must be re-executed.

4.3. Example of the Algorithm



(a) The input graph G. (b) A tree decomposition of G.



(c) The chordal completion $\widehat{G}$ with fill-in edges ($\widehat{E}$) added.

Figure 1: An illustration of the preprocessing pipeline for the treewidth-based routing algorithm.

We now illustrate the entire pipeline using the graph from Figure 1.

Input Graph. The process begins with the graph G in Figure 1(a). We chose the weights to be symmetric $d(u, v) = d(v, u)$ for simplicity of presentation, generally the algorithm does not require this. The graph has 7 vertices $\{v_1, \ldots, v_7\}$, the vs are omitted in the figure for clarity. The edges has the following weights: $d(v_1, v_2) = 5$, $d(v_2, v_3) = 1$, $d(v_2, v_6) = 12$, $d(v_3, v_4) = 2$, $d(v_4, v_5) = 7$, $d(v_4, v_7) = 3$, and $d(v_6, v_7) = 4$.

Phase 1 (Tree Decomposition):. Figure 1(b) shows a valid tree decomposition $\mathcal{T}$ of G. The largest bag is of size 3 (e.g., $X_1 = \{2, 4, 6\}$), so the treewidth is $tw(G) = 3-1 = 2$. It is straightforward to verify that (i) for each edge (v_i, v_j) in G, there exists a bag in $\mathcal{T}$ containing both v_i and v_j, and (ii) for each vertex v_i, the bags containing v_i form a connected subtree in $\mathcal{T}$.

Phase 2 (Fill-in and PEO):. We apply Algorithm 2 to $\mathcal{T}$ to get the chordal completion $\widehat{G}$ and a PEO π. Processing the bags from the leaves of $\mathcal{T}$ inwards yields the chordal graph in Figure 1(c). The required fill-in edges (shown in purple) are (v_2, v_4) and (v_4, v_6) as the pairs of vertices are present in the bags X_1 and X_3. A valid PEO $\pi = (v_2, v_6, v_4, v_7, v_3, v_1, v_5)$ that can be generated from this process by trimming the bags in the following order: $X_4 \to X_5 \to X_3 \to X_2 \to X_1$.

Phase 3 (Enforcing DPC):. This is the core computational step (Algorithm 3). We initialize d^* with the weights from G, setting $d^*(2, 4) = \infty$ and $d^*(4, 6) = \infty$. The algorithm iterates backward through the PEO. The only updates that change a distance value occur for the following vertices:

$$\begin{aligned}
\text{For } v_3: \quad & d^*(v_2, v_4) \leftarrow \min(\infty, d^*(v_2, v_3) + d^*(v_3, v_4)) \\
& = \min(\infty, 1 + 2) = 3 \\
\text{For } v_7: \quad & d^*(v_4, v_6) \leftarrow \min(\infty, d^*(v_4, v_7) + d^*(v_7, v_6)) \\
& = \min(\infty, 3 + 4) = 7 \\
\text{For } v_4: \quad & d^*(v_2, v_6) \leftarrow \min(12, d^*(v_2, v_4) + d^*(v_4, v_6)) \\
& = \min(12, 3 + 7) = 10
\end{aligned}$$

Phase 4 (SSSP Query):. Suppose we are given the query $s = v_4$. We execute Algorithm 4. The distance array D, ordered by the PEO $\pi = (v_2, v_6, v_4, v_7, v_3, v_1, v_5)$, is initialized to $[\infty, \infty, 0, \infty, \infty, \infty, \infty]$. The updates proceed as follows:

Pass 1 (Backward Pass): *Pass 2 (Forward Pass):*

v_6 :$[3, 7, 0, \infty, \infty, \infty, \infty]$ v_2 :$[3, 7, 0, \infty, 4, 8, \infty]$
v_4 :$[3, 7, 0, \infty, \infty, \infty, \infty]$ v_6 :$[3, 7, 0, 11, 4, 8, \infty]$
v_2 :$[3, 7, 0, \infty, \infty, \infty, \infty]$ v_4 :$[3, 7, 0, 3, 2, 8, 7]$

The remaining vertices in the PEO do not produce further updates. The final distance array from source v_4, ordered by the PEO, is $[3, 7, 0, 3, 2, 8, 7]$.

5. Implementation and Experimental Results

Implementation. We implemented our algorithm in Python 3 as a tool named Hermes. Hermes is open-source and released into the public domain. Our implementation leverages the Python library NetworkX [21] for graph operations and tree decompositions. Hermes is available at https://github.com/SanazSafaei/Hermes-Structure-Aware-Optimal-DEX-Routing.

Benchmarks and Experimental Setting. We collected a comprehensive dataset by capturing snapshots of all Uniswap liquidity pools across 20,000 consecutive blocks, specifically from block 22,820,000 to block 22,839,999. Data were collected using the official Uniswap APIs [5]. We evaluated Hermes in comparison with the state of the art Modified Moore Bellman Ford algorithm [4], which is the only existing graph based method. For additional comparison, we also included the standard Bellman Ford algorithm as a baseline. For each block, we constructed the corresponding token graph and queried SSSPs from 100 randomly selected tokens, recording the average query time per block. To further assess scalability, we repeated this procedure for four different token set sizes: the top 100, 1,000, 10,000, and 100,000 tokens, ranked by TVL in USD.

	Hermes (Ours)						MMBF						BF					
	Solved		Failed		Timeout		Solved		Failed		Timeout		Solved		Failed		Timeout	
	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
100	2,000,000	100.0%	0	0.0%	0	0.0%	1,999,998	99.9%	0	0.0%	2	0.0%	60,000	3.0%	1,940,000	97.0%	0	0.0%
1,000	2,000,000	100.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	2,000,000	100.0%	27,994	1.3%	1,972,006	98.6%	0	0.0%
10,000	2,000,000	100.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	2,000,000	100.0%	10,634	0.5%	13,107	0.6%	1,976,259	98.8%
100,000	2,000,000	100.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	2,000,000	100.0%	12,884	0.6%	3,627	0.1%	1,983,489	99.1%
Total	8,000,000	100.0%	0	0.0%	0	0.0%	1,999,998	25.0%	0	0.0%	6,000,002	75.0%	111,512	1.4%	3,928,740	49.1%	3,959,748	49.5%

TABLE 1: Success rates for Hermes (ours), MMBF, and BF on routing queries for four token set sizes. "Failed" indicates cases where the tool did not work due to negative cycles.

	Hermes (Ours)		MMBF		BF	
	Runtime	Completed	Runtime	Completed	Runtime	Completed
100	0.0002s	2,000,000	2.8191s	1,999,998	0.0158s	2,000,000
1,000	0.0021s	2,000,000	TIMEOUT	0	1.1653s	2,000,000
10,000	0.0197s	2,000,000	TIMEOUT	0	1.1304s	23,741
100,000	0.1942s	2,000,000	TIMEOUT	0	0.8155s	16,511

TABLE 2: Average runtimes (in seconds) for Hermes (ours), MMBF, and BF across different token set sizes. The "Completed" column indicates the number of queries finished within the timeout; "TIMEOUT" denotes that all queries exceeded the time limit.

Each query was subject to a 12-second timeout, matching the average Ethereum block interval.

Treewidth. Experimental analysis shows that the average treewidth for token sets of sizes 100, 1,000, 10,000, and 100,000 is 8, 18, 38, and 71, respectively. As illustrated in Figure 2, while treewidth increases with the number of tokens, its growth rate is substantially slower than that of the total number of tokens. These results demonstrate the suitability of parameterized algorithms for this problem.

Dynamics of Pool Updates. For the token set sizes studied, we observed an average of 9.7 price updates per block, about 1 new token every 12 blocks, and about 1 new pool every 33 blocks, reflecting few edge weight changes and infrequent additions of vertices and edges. Figure 3 shows the distribution of price updates across blocks. These results show that the proposed approach for handling dynamic edges is practical for real-world pools.

Robustness. We measured the success rate of Hermes in finding valid routes, compared to MMBF and BF, by running 2,000,000 queries for each of the four token set sizes. Hermes successfully solved **100%** of queries, while MMBF and BF solved only **25%** and **1.4%**, respectively. The primary limitation for MMBF was frequent timeouts when the token count exceeded 100. For BF, failures on smaller token sets 100, 1,000 were mainly due to negative cycles, while timeouts dominated for larger sets 10,000 and 100,000. Table 1 summarizes the success rates and coverage for all three algorithms.

Scalability. We evaluated the runtime performance of all algorithms. MMBF completed within the time limit only for the smallest token set of 100, which precludes direct runtime comparison on larger sizes. On this benchmark, Hermes achieved an average query time of **0.0002** seconds, while MMBF required 2.8191 seconds, representing an improvement of four orders of magnitude. BF completed within the time limit for token sets of size 100 and 1,000 mainly. For queries solved by both Hermes and BF, Hermes averaged 0.054 seconds per query, compared to 0.5947 seconds for BF. It is important to note that BF frequently failed to return valid routes on 97.8% of queries owing to the presence of negative cycles, which rendered the majority of its results unusable. Hermes was the only method able to process the largest token sets, with average query times of **0.0197** seconds and **0.1942** seconds for 10,000 and 100,000 tokens, respectively. Detailed results are presented in Table 2.

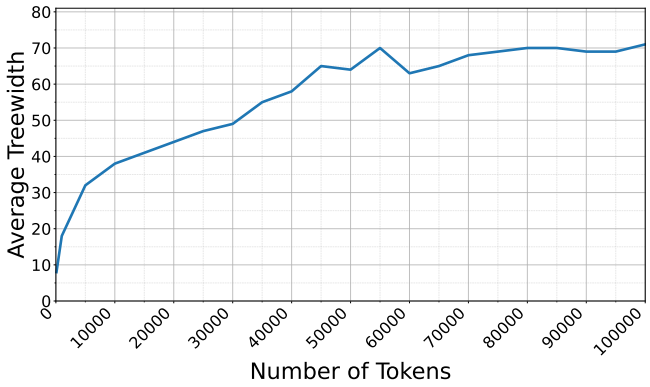


Figure 2: Comparison of the growth rates of the number of tokens and the treewidth.

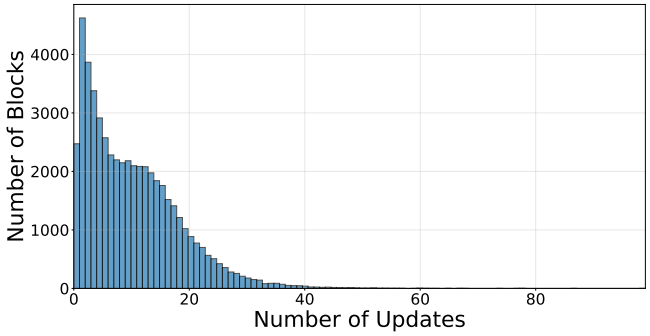


Figure 3: Histogram of token price updates per block.

6. Discussion and Future Work

Broader Adoption. Similar to other graph-based routing algorithms that have been widely applied for arbitrage detection, our method has the potential to identify negative cycles within its steps. In particular, because it efficiently processes large sets of tokens, it opens opportunities for future research to evaluate the potential profitability of arbitrage opportunities in large-scale token graphs.

Impact of Slippage. Slippage arises when the execution price of a trade differs from the spot price, often as a result of market volatility or limited liquidity. Previous studies indicate that slippage can influence trading outcomes, particularly for large trades [3]. Since our algorithm relies on spot prices, similar to prior research, it encounters comparable limitations for high-volume trades. Thus, this approach can benefit from preprocessing steps such as imposing additional restrictions on liquidity for a given set of pools.

Transaction Costs and Trade Efficiency. Executing trades incurs transaction fees, which increase as the route includes more steps. To adjust for this, one can increase the edge prices by an upper bound estimate that reflects the cost of a single trade. Previous works have estimated such upper bounds for smart contract functions, and incorporating these estimates can prevent infeasible routes [22].

7. Conclusion

This paper presents an extension of a treewidth-based algorithm for optimal routing on DEXs. By leveraging the structural properties of liquidity pools, the proposed approach bridges the gap between scalability and robustness. To our knowledge, this is the first application of parameterized algorithms in the context of DEXs. We implemented the algorithm in a tool named Hermes and demonstrated its effectiveness through extensive experiments. Notably, Hermes is the only available tool that efficiently manages large token sets, as found in real-world DEXs.

Acknowledgments

The research was partially supported by the Ethereum Foundation Research Grant FY24-1793. Authors are ordered alphabetically.

References

[1] R. McLaughlin, C. Kruegel, and G. Vigna, "A large scale study of the ethereum arbitrage ecosystem," in *USENIX Security Symposium*, 2023, pp. 3295–3312.

[2] "Defillama - defi dashboard," https://defillama.com/, accessed: 2025-07-03.

[3] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols," *ACM Comput. Surv.*, vol. 55, no. 11, pp. 238:1–238:50, 2023.

[4] Y. Zhang, T. Yan, J. Lin, B. Kraner, and C. J. Tessone, "An improved algorithm to identify more arbitrage opportunities on decentralized exchanges," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2024, pp. 1–7.

[5] "The uniswap subgraph," https://docs.uniswap.org/api/subgraph/overview, accessed: 2025-07-03.

[6] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized Algorithms*, 2015.

[7] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, and R. Wattenhofer, "Cyclic arbitrage in decentralized exchanges," in *WWW (Companion Volume)*, 2022, pp. 12–19.

[8] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *SP*, 2021, pp. 919–936.

[9] V. Danos, H. E. Khalloufi, and J. Prat, "Global order routing on exchange networks," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 12676, 2021, pp. 207–226.

[10] S. Park, W. Jeong, Y. Lee, B. Son, H. Jang, and J. Lee, "Unraveling the MEV enigma: Abi-free detection model using graph neural networks," *Future Gener. Comput. Syst.*, vol. 153, pp. 70–83, 2024.

[11] T. Diamandis, M. Resnick, T. Chitra, and G. Angeris, "An efficient algorithm for optimal routing through constant function market makers," in *FC*, ser. Lecture Notes in Computer Science, vol. 13951, 2023, pp. 128–145.

[12] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, and A. Shakiba, "Optimal blocks for maximizing the transaction fee revenue of bitcoin miners," *J. Comb. Optim.*, vol. 49, p. 15, 2025.

[13] T. Barakbayeva, S. Farokhnia, A. K. Goharshady, M. Gufler, and S. Novozhilov, "Pixiu: Optimal block production revenues on cardano," in *Blockchain*, 2024, pp. 491–496.

[14] K. Chatterjee, A. K. Goharshady, and E. K. Goharshady, "The treewidth of smart contracts," in *SAC*, 2019, pp. 400–408.

[15] H. Adams, M. Salem, N. Zinsmeister, S. Reynolds, A. Adams, W. Pote, M. Toda, A. Henshaw, E. Williams, and D. Robinson, "Uniswap v4 core [draft]," 2023.

[16] H. L. Bodlaender, "A linear time algorithm for finding tree-decompositions of small treewidth," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, 1993, pp. 226–234.

[17] D. Delling, D. Fleischman, A. Malucelli, and R. F. Werneck, "The 2nd annual parameterized algorithms and computational experiments challenge (pace 2017)," vol. 89, 2017, pp. 32:1–32:11.

[18] P. Heggernes, "Treewidth, partial k-trees, and chordal graphs," *Partial curriculum in INF334-Advanced algorithmical techniques, Department of Informatics, University of Bergen, Norway*, 2005.

[19] L. Planken, M. de Weerdt, and R. van der Krogt, "Computing all-pairs shortest paths by leveraging low treewidth," *J. Artif. Int. Res.*, vol. 43, no. 1, p. 353–388, 2012.

[20] N. Chleq, "Efficient algorithms for networks of quantitative temporal constraints," *Constraints*, vol. 95, 1995.

[21] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[22] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "*Don't run on fumes* - parametric gas bounds for smart contracts," *J. Syst. Softw.*, vol. 176, p. 110923, 2021.