RICH: Real-time Identification of negative Cycles for High-efficiency Arbitrage

Bingqiao Luo
National University of Singapore
luo.bingqiao@u.nus.edu

Jiaxin Jiang*
National University of Singapore
jiangjx@comp.nus.edu.sg

Yuhang Chen
National University of Singapore
yuhangc@comp.nus.edu.sg

Junyi Hou
National University of Singapore
junyi.h@comp.nus.edu.sg

Cheng Jun Tey
National University of Singapore
chengjuntey@u.nus.edu

Ziyang Qiu
National University of Singapore
ziyangqiu@u.nus.edu

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Spencer Xiao
Tokka Labs
spencer.xiao@tokkalabs.com

Dominic Ong
Tokka Labs
dominic.ong@tokkalabs.com

Wee Howe Ang
Tokka Labs
weehowe.ang@tokkalabs.com

ABSTRACT

Arbitrage is a challenging data science problem characterized by rapidly fluctuating price discrepancies across multiple markets, necessitating real-time solutions. To overcome the challenge, we model it as a k-hop negative cycle detection problem in graphs and introduce RICH: **R**eal-time **I**dentification of negative **C**ycles for **H**igh-efficiency arbitrage. RICH is a novel framework that leverages color-coding and dynamic programming to accelerate the identification of negative-weight cycles without exhaustive graph traversal. Additionally, RICH incorporates encoding techniques and graph reduction to minimize computational overhead while maintaining probabilistic guarantees. Our extensive experiments on real-world datasets demonstrate that RICH is up to **32.69× faster** than state-of-the-art methods, enabling timely arbitrage execution while outperforming existing methods in both speed and accuracy. We further validate its effectiveness in identifying arbitrage opportunities in cryptocurrency markets and foreign exchange markets.

PVLDB Reference Format:
Bingqiao Luo, Jiaxin Jiang, Yuhang Chen, Junyi Hou, Cheng Jun Tey, Ziyang Qiu, Bingsheng He, Spencer Xiao, Dominic Ong, and Wee Howe Ang. RICH: Real-time Identification of negative Cycles for High-efficiency Arbitrage. PVLDB, 18(11): 4081 - 4089, 2025.
doi:10.14778/3749646.3749678

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Xtra-Computing/RICH.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749678

1 INTRODUCTION

Arbitrage is a fundamental concept in the financial system, referring to the practice of exploiting price differences across multiple markets to generate profit [16]. It is widely observed in various financial markets such as capital asset [42], foreign currency trading [26], and stock exchanges [10], where pricing inefficiencies create opportunities for arbitrageurs. For example:

(1) **Arbitrage in foreign currency trading [21, 26].** Triangular arbitrage is a classic FX strategy, where a trader exchanges through three currencies to exploit price discrepancies and return to the original currency. This opportunity is more common during periods of high market volatility.
(2) **Arbitrage in cryptocurrency market [18, 31]**. The rise of cryptocurrencies has enabled new arbitrage opportunities. Automated market makers (AMMs) [2] set prices on decentralized exchanges, often causing liquidity imbalances. Cyclic arbitrage that trading through multiple assets and returning to the original for profit is common.

In this paper, we study arbitrage in the cryptocurrency market, which offers real-time, open data, unlike traditional markets with restricted or delayed access [33, 49]. In collaboration with our industry partner Tokka Labs, we leverage a robust data pipeline to systematically identify arbitrage opportunities across multiple exchanges. In this paper, we use this data pipeline as an example and it is common for arbitrage activities in different companies [34, 50].

Data Pipeline for Arbitrage (Figure 1). Tokka Labs is a proprietary trading firm specializing in high frequency on-chain trading strategies. Deployed on over 40 on-chain venues, the team focuses on market making, searching, and solving for top protocols on the most popular blockchains in the world. Based on our discussions, we illustrate this arbitrage data pipeline for trading firms as a general industry framework, outlining four key steps in the process: (1) **Graph Construction**, where a directed token graph is built with nodes representing cryptocurrency tokens (such as Bitcoin (BTC) and Ethereum (ETH)) and edges representing tradable pairs derived from AMM-based liquidity pools [45, 50]. Exchange rates are

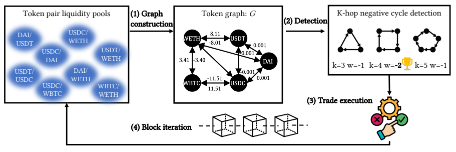
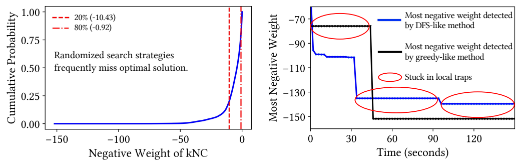


Figure 1: Data pipeline for arbitrage.



(a) CDF of kNC weights. **(b) Weight vs. Time.**

Figure 2: Example of challenges in a token graph.

transformed using logarithms to convert multiplicative price ratios into additive edge weights, facilitating negative cycle detection; (2) **Negative Cycle Detection**, which involves identifying k-hop most negative-weight cycles (kMNC) in the transformed graph, indicating potential profitable arbitrage sequences. The hop constraint k prevents excessive gas fees and execution failures [39]. (3) **Trade Execution**, where a detected negative cycle is traversed starting from an arbitrary token, executing the sequence of trades to obtain a net gain in the starting token; and (4) **Block Iteration**, in which the entire pipeline is repeated for each newly mined blockchain block to account for rapid updates in liquidity and prices [1, 18].

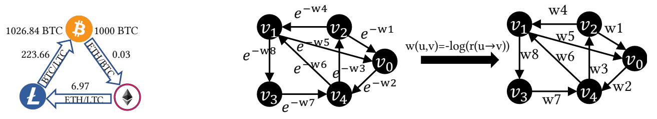
Efficient kMNC detection in large token graphs is crucial for real-time, scalable arbitrage. Key challenges include:

Challenge 1: High Computational Complexity of kMNC *Detection.* The kMNC problem is proven to be NP-complete, making exact solutions infeasible for large-scale crypto markets with thousands of tokens and pools [34]. As k increases, the number of potential cycles grows exponentially, significantly extending computation times and making results highly sensitive to search heuristics. Figure 2a shows the skewed distribution of k-hop negative cycle (kNC) weights in a token graph. Randomized search methods often get stuck in local optima and miss profitable cycles without structural guidance [46].

Challenge 2: Inability of Existing Methods to Escape Local Subgraph Searches. State-of-the-art arbitrage detection algorithms typically rely on local subgraph searches, restricting exploration to limited regions of the token graph [50]. This prevents the discovery of globally optimal negative cycles, resulting in suboptimal solutions and longer detection times. As illustrated in Figure 2b, the most negative cycle weights improve gradually, but identifying the optimal solution with existing methods can take over 40 seconds [34, 46]. Such local search traps hinder arbitrage detection effectiveness in rapidly evolving markets and reduce potential profits.

To address these challenges, we propose RICH for efficient kMNC detection in financial graphs. Our main contributions are:

- **Color-Coding Based Detection:** We propose a Color-Coding based algorithm that efficiently detects kMNC in polynomial time with a high probability of identifying the optimal solution with rigorous probabilistic guarantees.



(a) Arbitrage. **(b) Negative cycle.**

Figure 3: Example of arbitrage and negative cycles.

- **Efficient Dynamic Programming:** We design tailored dynamic programming (DP) techniques specialized for cycle detection under hop constraints, which avoid redundant computation and speed up detection within each color-coding instance.
- **Color-Coding-Aware Optimizations:** We develop bitwise color-set encoding and color-guided graph reduction within the color-coding framework, greatly reducing search space and overhead. These optimizations enable RICH to scale to large graphs and support real-time cycle detection.
- **Empirical Validation:** We conduct extensive experiments demonstrating that our solution is up to **32.69x faster** than state-of-the-art methods. Additionally, case studies in cryptocurrency markets and foreign exchange markets validate RICH's capability to execute arbitrage swiftly in real-world scenarios, highlighting its practical effectiveness and reliability.

2 BACKGROUND AND RELATED WORK

2.1 Preliminaries

Graph. We consider a directed and weighted graph $G = (V, E)$, where V represents the set of vertices (nodes) and $E \subseteq V \times V$ represents the set of directed edges. For any vertex u, we use $N(u)$ to denote the set of neighbors of u, i.e., $N(u) = \{v \mid (u, v) \in E\}$.

Cyclic Arbitrage. Cyclic arbitrage exploits price discrepancies across a cycle of token exchanges to generate profit. An arbitrageur trades through multiple pairs and returns to the original token with more than they started. Figure 3a shows an example of arbitrage that sequentially trades BTC for ETH, ETH for Litecoin (LTC), and LTC back to BTC to capture profitable price discrepancies [41]. Detecting such cycles is key for automated crypto trading strategies.

Example 2.1. Consider the cycle $v_2 \to v_0 \to v_4 \to v_2$ in Figure 3b (LHS), where each edge encodes an exchange rate $r(u \to v)$. After a logarithmic transformation, the edge weights become $w(u, v) = -\log(r(u \to v))$ (RHS). If the total weight $w_1 + w_2 + w_3 < 0$, this implies a profitable arbitrage, since the product of the exchange rates exceeds one: $r(v_2 \to v_0) \cdot r(v_0 \to v_4) \cdot r(v_4 \to v_2) > 1$. The more negative the total weight, the higher the potential arbitrage profit.

Edge Weight. Each edge $(u, v) \in E$ represents a trading pair from token u to token v with an exchange rate $r(u \to v)$. For example, the exchange rate $r(\text{BTC} \to \text{ETH})$ has fluctuated between 12 and 48 over the past five years. To facilitate efficient cycle detection, we transform the exchange rates into edge weights with a logarithmic function: $w(u, v) = -\log(r(u \to v))$. This process converts the multiplicative relationship of exchange rates into an additive framework, enabling the use of negative cycle detection algorithms.

2.2 Problem Statement and Baseline Solution

Simple Cycle. A *simple cycle* in G is a sequence of vertices $C = (v_1, v_2, \ldots, v_k, v_1)$ such that each pair (v_i, v_{i+1}) is a directed edge in E, and all vertices $v_1, v_2, \ldots, v_k$ are non-repeated except $v_{k+1} = v_1$.

K-hop Negative Cycle. The weight of a cycle C is defined as the total sum of the edge weights along the cycle. A k-hop negative cycle is a cycle $C = (v_1, v_2, \ldots, v_k, v_1)$ in the graph, where the sum of the edge weights along the k-hop cycle is negative:

$$w(C) = \sum_{i=1}^{k} w(v_i, v_{i+1}) < 0,$$

where $v_{k+1} = v_1$. For instance, a 3-hop negative cycle consists of three distinct vertices v_1, v_2, v_3 and three directed edges (v_1, v_2), (v_2, v_3), and (v_3, v_1), forming a complete cycle with exactly 3 hops.

Problem Statement (kMNC). Given a directed and weighted graph $G = (V, E)$, a specified cycle hop k, and edge weights $w(u, v)$ defined for all $(u, v) \in E$, the objective is to identify a simple k-hop most negative-weight cycle (kMNC).[1]

Theorem 2.2. *Let $G = (V, E)$ be a directed, weighted graph with an edge-weight function $w : E \to \mathbb{R}$. Given an integer k and a threshold $\Theta \in \mathbb{R}$, the decision problem*

"Is there a simple cycle of exactly k hops with total weight $< \Theta$?"

is NP-complete. Consequently, the optimization problem of finding a most negative *k-hop cycle is NP-hard.*

Proof Sketch. The problem is NP-complete. The detailed proof appears in Section A.2 of [32]. □

2.3 Related Work

Negative cycle detection. Negative cycle detection has been studied for decades [12, 13]. Algorithms like BFM [4, 19, 36], Goldberg–Radzik [24], and Pallottino's Algorithm [20] integrate cycle detection strategies such as walk-to-the-root, amortized search, and admissible graph search [23]. Recent advances enhance efficiency with low-diameter SCC partitioning, recursive scaling, and dense distance graphs [7, 8, 29]. However, most of these methods focus on detecting the existence of a negative cycle rather than accurately identifying the most negative cycles. Additionally, they often struggle with hop constraints, reducing their efficiency.

Cycle Enumeration. Graph query techniques have been widely applied to real-world problems [11, 27, 47], in particular for cycle and s-t path enumeration in graphs [6, 28, 37, 43, 44]. The enumeration of hop-constrained cycles has been approached using hot point based index [40], subgraph isomorphism optimization [15], adjacency matrix trace and matrix multiplication [22], and advanced pruning techniques [25]. While these algorithms can enumerate all hop-constrained cycles or paths, they often ignore edge weights during detection. In cryptocurrency markets, the set of tokens and trading pairs is stable over short periods, but edge weights (exchange rates) change rapidly, especially in active pools. These frequent updates impact many cycles, making cycle weight computation the main bottleneck. Existing methods struggle to efficiently solve the kMNC problem under such conditions.

[1] This supports fixed-hop arbitrage strategies used in practice, where exact cycle length is crucial. The at-most-k variant returns the best cycle within a hop budget but cannot isolate the optimal one at a specific length, limiting its use in layered strategies. It can be supported with minor extensions (see Appendix C of [32]).

Table 1: Comparison of RICH and previous algorithms.

Feature	PathEnum **[43]**	DeFi-ARB **[50]**	Goldphish **[34]**	BriDe **[46]**	RICH **(Ours)**
Accuracy Guarantees	✓	✗	✗	✓	✓
Hop Constraint Support	✓	✗	✓	✓	✓
Avoids Local Subgraph Traps	✗	✗	✗	✗	✓
Avoids Redundant Cycle Traversal	✗	✗	✓	✗	✓

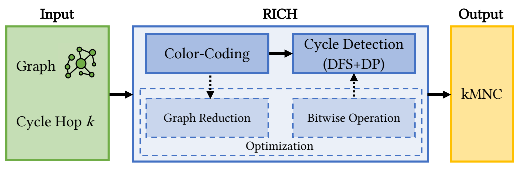


Figure 4: An overview of RICH.

Crypto arbitrage detection. Arbitrage detection in cryptocurrencies has seen significant advancements recently [5, 38, 39, 45]. DeFi-ARB [50] was among the first to detect arbitrage transactions on DEXs, using the BFM algorithm for negative cycle detection and logical modeling to identify non-cyclic complex strategies. Goldphish [34] employed a greedy-based algorithm to traverse the token graph, combining heuristics to limit cycle length, exchange, and pivot tokens for detecting arbitrage cycles. DFS-based methods have been used to find arbitrage cycles [14, 46]; [48] adapts BFM for non-loop arbitrage paths. However, these approaches depend on heavy heuristics for graph reduction, limiting scalability.

Therefore, we propose RICH: **R**eal-time **I**dentification of negative **C**ycles for **H**igh-efficiency arbitrage. Table 1 summarized the comparison between RICH and the previous methods.

3 DESIGN AND IMPLEMENTATION OF RICH

Overview (Figure 4). We break down RICH's core components:

(1) *Color-Coding.* Section 3.1 revisits the color-coding method, which assigns k colors to graph vertices to detect *multicolored* cycles, ensuring each vertex in a cycle has a unique color and avoiding local substructure traps.
(2) *Weighted Cycle Detection.* Section 3.2 extends color-coding to weighted graphs via DFS to identify the kMNC. Repeated randomization ensures the most negative multicolored k-hop cycle is detected with high probability.
(3) *Dynamic Programming (DP) for Efficiency.* Although DFS can find multicolored cycles, it often revisits overlapping subpaths. Section 3.3 presents a DP approach that caches intermediate results by vertex and color set, greatly reducing redundancy and accelerating the search.
(4) *Optimizations.* Section 4 presents two optimizations: bitwise color-set encoding for $O(1)$ operations and graph reduction that prunes nodes with a single valid same-color neighbor. Both greatly shrink the search space while preserving k-hop multicolored cycle detection.

3.1 Randomized Color-Coding Revisit

Color-coding [3] efficiently detects fixed-length simple cycles by focusing on *multicolored* paths, reducing redundant computation.

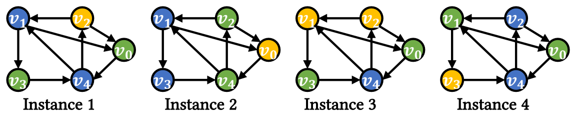


Figure 5: Example of four color coding instances with k=3.

Algorithm 1: Randomized Color-Coding for k-Hop Cycles

```
Input: $G = (V, E)$, cycle hop $k$, number of color-coding instances $\ell$
Output: A set $C$ of all discovered $k$-hop simple cycles
1 Initialize: $C \leftarrow \varnothing$
2 for $i \leftarrow 1$ to $\ell$ do // Perform $\ell$ color-coding instances
3     foreach // Assign random color to vertex $v$ $v \in V$ do
4         $\alpha(v) \leftarrow \text{rand}(\{1, \ldots, k\})$
5     $C_i \leftarrow$ FindMulticoloredCycles$(G, \alpha, k)$
6     $C \leftarrow C \cup C_i$
7 return $C$
```

Unlike repeated random search, which may revisit nodes and miss valid cycles, color-coding guarantees efficient and correct detection.

Definition 3.1 (Color-Coding Instance). Given a directed graph $G = (V, E)$, a cycle length k, and a random color assignment $\alpha : V \longrightarrow \{1, \ldots, k\}$, a *color-coding instance* is one execution of assigning colors to vertices, where each vertex v independently receives a color $\alpha(v)$ chosen uniformly at random from $\{1, \ldots, k\}$.

Color-coding Paradigm (Algorithm 1). This paradigm generates ℓ distinct *color-coding instances*, each by assigning a random color $\alpha(v) \in \{1, \ldots, k\}$ to every vertex v. In each instance, FindMulticoloredCycles identifies any simple cycles of length k where all vertices have distinct colors (i.e., *multicolored* cycles). Such cycles are guaranteed to be simple since no vertex color repeats. All cycles discovered across the color-coding instances are accumulated in C. Ultimately, C contains the union of all k-hop simple cycles found.

Lemma 3.2. *Any k-hop multicolored cycle in G is necessarily a k-hop simple cycle.*

Proof. If each of the k vertices in a cycle has a distinct color, none of the vertices (and thus none of the edges) can repeat in the cycle. Hence, the cycle is simple [35]. □

Example 3.3. Figure 5 illustrates four color-coding instances of the graph G in Figure 3b. Each instance is created by independently assigning one of three colors to every vertex in G. In the first instance, a multicolored cycle $C_1 = \{v_2, v_0, v_4\}$ is successfully detected because each vertex in the cycle has a distinct color. Similarly, in the second instance, another multicolored cycle $C_2 = \{v_1, v_0, v_4\}$ is identified. Consequently, the set of simple cycles found in the original graph is $C = \{C_1, C_2\}$.

3.2 Color-Coding for Weighted Cycle Detection

Traditional color-coding [3] effectively detects simple cycles but does not account for cycle weights, limiting its applicability in arbitrage scenarios. To overcome this limitation, RICH introduces an extended color-coding algorithm designed for kMNC. The most negative k-hop cycle is formalized as follows.

Definition 3.4 (k-Hop Most Negative Cycle). The *k-hop most negative cycle* in a directed graph $G = (V, E)$ is denoted by C^*:

$$C^* = \arg \min_{\substack{C \subseteq G \\ \text{length}(C) = k}} w(C),$$

where $w(C)$ represents the total weight of cycle C, calculated as the sum of the weights of its constituent edges.

Lemma 3.5 (Minimality Preservation in Color-Coding). *Let C^* be the k-hop most negative cycle in a directed graph G. In any color-coding instance that assigns each vertex in G a random color from $\{1, \ldots, k\}$, if all vertices of C^* receive distinct colors (thus forming a multicolored cycle), then C^* remains the unique minimal-weight cycle of length k in that instance.*

Proof. Suppose, for contradiction, that there exists a multicolored k-hop cycle C' in the same color-coding instance such that

$$w(C') < w(C^*).$$

Since the color assignment does not alter the edge weights, C' is a valid k-hop cycle in G with total weight lower than that of C^*. This contradicts the definition of C^* as the cycle with the minimal weight among all k-hop cycles in G. Hence, no such cycle C' can exist, and C^* remains the unique minimal-weight cycle in that instance. □

Probability of Detecting C^*. Lemma 3.5 underpins the analysis of detecting the globally minimal cycle C^*. In any color-coding instance, if all k vertices of C^* are colored distinctly, then C^* remains the unique minimal-weight k-hop cycle in that instance. Because each vertex's color is chosen uniformly and independently from $\{1, \ldots, k\}$, the probability of C^* becoming a multicolored cycle in one instance is $\frac{k!}{k^k}$. Over ℓ independent color-coding instances, the probability of never observing C^* in a fully distinct coloring is $\left(1 - \frac{k!}{k^k}\right)^\ell$, leading to a detection probability of $1 - \left(1 - \frac{k!}{k^k}\right)^\ell$. As ℓ increases, the probability of C^* appearing in at least one color-coding instance approaches 1. This analysis provides a probabilistic guarantee that C^* will be captured as ℓ increases, indicating that the optimal solution is found with high probability.

In the following, we demonstrate how to employ depth-first search (DFS) to efficiently detect all such multicolored cycles.

Definition 3.6 (Color-Coding for Paths). Let $\alpha : V \longrightarrow \{1, \ldots, k\}$ be a color function that assigns each vertex $v \in V$ a color $\alpha(v)$. For a path $p = [v_1, v_2, \ldots, v_n]$, the color set is defined as: $\alpha(p) = \{\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_n)\}$. The set denotes unique colors of p.

Definition 3.7 (Path Concatenation). Let $p = [v_1, v_2, \ldots, v_n]$ and $q = [v_n, v_{n+1}, \ldots, v_m]$ be two paths in a graph $G = (V, E)$, where the last vertex of p is the first vertex of q (i.e., $v_n = v_n$). The concatenation of paths p and q, denoted by $p \cdot q$, is the new path: $p \cdot q = [v_1, v_2, \ldots, v_n, v_{n+1}, \ldots, v_m]$. This operation combines the two paths into a single continuous path.

Color-Coding Enhanced Depth-First Search (Algorithm 2). The algorithm searches for the most negative k-hop cycle C^* by initiating a depth-first search (DFS) from each vertex in the graph (Line 3). For every DFS invocation, it examines each neighbor u of the current vertex v. If the color of u is already present in the current path and u is not the starting vertex $p[0]$ (Line 8), the

Algorithm 2: DFS for kMNC

```
Input: Graph $(V, E)$, color function $\alpha : V \rightarrow \{1, \ldots, k\}$, cycle hop $k$
Output: $k$-hop most negative cycle $C^*$
$C^* \leftarrow \emptyset$, $w(C^*) \leftarrow +\infty$
foreach $v \in V$ do
    DFS($v$, $[v]$, $0$) remove $v$ from $V$;
return $C^*$
Function DFS($v, p, w$):
    foreach $u \in N(v)$ do                /* Explore neighbors of $v$ */
        if $\alpha(u) \in \alpha(p)$ and $u \neq p[0]$ then
            continue
        $p' \leftarrow p \cdot [v, u]$, $\Delta \leftarrow w + w(v, u)$
        if $u = p[0]$ and $|p| = k$ then
            if $\Delta < w(C^*)$ then
                $C^* \leftarrow p'$, $w(C^*) \leftarrow \Delta$
            continue
        if $|p| < k$ then /* Continue DFS if path not yet $k$-hop */
            DFS($u, p', \Delta$)
```

neighbor is skipped to maintain a multicolored path. Otherwise, u is appended to the path, and the cumulative weight Δ is updated (Line 9). If appending u returns to the starting vertex and the path length reaches k (Line 10), a valid k-hop cycle is formed by closing the loop. The algorithm then compares the weight of this cycle with the current most negative cycle C^* and updates C^* if the new cycle has a smaller weight (Lines 12). If the path length is still less than k (Line 14), the DFS continues recursively with the extended path and updated weight. After all vertices have been explored, the algorithm returns C^* as the most negative k-hop cycle found.

Vertex Removal for Efficiency. When DFS starts from vertex v (Line 3, Algorithm 2), it explores all multicolored paths of length up to k from v. Any cycle containing v can thus be found with v as the root. After finishing DFS from v, we remove v from the graph (Line 3) so later DFS will not revisit it. In practice, this can reduce the branching factor for the remaining vertices, thereby accelerating searches in dense graphs.

Lemma 3.8 (Safe Vertex Removal). *Let $G = (V, E)$ be a directed graph and $k \geq 2$. Given a color assignment $\alpha : V \rightarrow \{1, \ldots, k\}$, define $\mathrm{DFS}(v)$ as a depth-k search from v that only extends a path $[v_1, \ldots, v_i]$ to v_{i+1} if $\alpha(v_{i+1}) \notin \{\alpha(v_1), \ldots, \alpha(v_i)\}$, except when closing a cycle to v_1. Then, any simple k-hop cycle $C = (v = u_1, u_2, \ldots, u_k, u_{k+1} = v)$ with distinct colors $\alpha(u_i)$ is found by $\mathrm{DFS}(v)$. Thus, removing v after $\mathrm{DFS}(v)$ does not miss any such cycle containing v.*

Proof. Since $\{\alpha(u_1), \ldots, \alpha(u_k)\}$ are all distinct, the DFS from $v = u_1$ will not skip any edge (u_i, u_{i+1}) for $1 \leq i < k$. It can thus form the path $[u_1, \ldots, u_k]$. Closing the cycle at $u_{k+1} = u_1$ is allowed because it returns to the start vertex. Hence, every multicolored k-hop cycle through v appears in $\mathrm{DFS}(v)$. Removing v afterward thus excludes no cycle that includes v. □

Detection Probability in Multiple Color-Coding Instances. As the number of coloring processes increases, the probability of detection improves. For example, when the length of a cycle C is $k = 4$ and $\ell = 50$ independent instances are performed, the probability of missing C across all trials is approximately 0.73%. This corresponds to a probability of successfully detecting the most negative cycle of about 99.27%.

Algorithm 3: DP for kMNC

```
Input: $G = (V, E)$, color function $\alpha : V \rightarrow \{1, \ldots, k\}$, cycle length $k$
Output: $k$-hop most negative cycle $C^*$
$C^* \leftarrow \emptyset$,   $w(C^*) \leftarrow +\infty$
foreach $v \in V$ do
    $dp[v][\{\alpha(v)\}] \leftarrow 0$              /* Initialize DP state */
for $l = 1$ to $k - 1$ do
    foreach $v \in V$ do
        foreach color set $\alpha(p)$ with $dp[v][\alpha(p)]$ defined do
            $w \leftarrow dp[v][\alpha(p)]$
            foreach $u \in N(v)$ do
                if $\alpha(u) \notin \alpha(p)$ then
                    $\alpha(p') \leftarrow \alpha(p) \cup \{\alpha(u)\}$
                    $w' \leftarrow dp[u][\alpha(p')]$
                    $dp[u][\alpha(p')] \leftarrow \min\{w', w + w(v, u)\}$
foreach $v \in V$ do
    foreach color set $\alpha(p)$ with $|\alpha(p)| = k$ and $dp[v][\alpha(p)]$ defined do
        if $p[0] \in N(v)$ then
            $w \leftarrow dp[v][\alpha(p)] + w(v, p[0])$
            if $w < w(C^*)$ then
                Update $C^*$ with the cycle corresponding to state
                $w(C^*) \leftarrow w$
return $C^*$
```

3.3 Dynamic Programming

While Algorithm 2 correctly solves the kMNC problem, its DFS-based approach suffers from significant redundancy. In the DFS process, many subpaths are revisited multiple times as they are shared among different search branches. This repeated exploration increases the computational cost, particularly in dense graphs. To alleviate this inefficiency, we propose a dynamic programming (DP) method that reuses the results of previously computed subpaths.

DP State. We represent the DP state as $dp[v][\alpha(p)]$, where v denotes the current vertex in the path, and $\alpha(p)$ includes the colors of nodes in the current path p. Each state stores the minimum weight of any valid path ending at v with color sequence $\alpha(p)$, enabling efficient reuse of previously computed subpaths.

Transition Function. When extending a path ending at vertex v with cumulative weight w to a neighbor u, we update the DP table:

$$dp[u][\alpha(p')] = \min(dp[u][\alpha(p')], dp[v][\alpha(p)] + w(v, u))$$

where v is the current node, u is its neighbor, p' is the extended path ensuring unique colors, and $w(v, u)$ is the edge weight. The transition updates the DP table by keeping the minimal weight path for each state, ensuring only lower-weight paths are considered.

Dynamic Programming for kMNC (Algorithm 3). The algorithm detects kMNC C^* using a DP approach. For each vertex $v \in V$, the recursive function DP is invoked with the starting path $p = [v]$, initial weight $w = 0$, and an empty DP table (Line 3). Starting from v , the algorithm extends the current path p by exploring all neighbors u (Line 8). If the color of u does not exist in the current path, u is appended to p, and the cumulative weight is updated. The DP table is checked and updated if the new path offers a lower weight for the state $dp[u][\alpha(p)]$ (Lines 9–12). If the path length remains less than k, the algorithm recurses with the updated state. When the path reaches k, the algorithm checks if the current vertex connects back to the starting vertex (Line 15). If the edge exists, the cycle

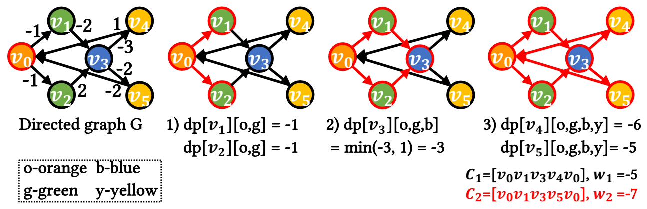
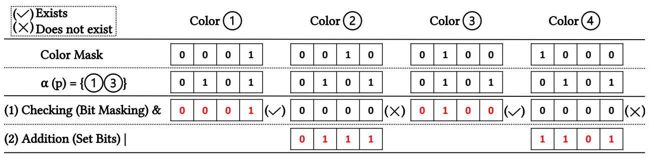


Figure 6: Example of dynamic programming state transition.



(✓) Exists
(✕) Does not exist

	Color ①		Color ②		Color ③		Color ④	
Color Mask	0 0 0 1		0 0 1 0		0 1 0 0		1 0 0 0	
α (p) = {①③}	0 1 0 1		0 1 0 1		0 1 0 1		0 1 0 1	
(1) Checking (Bit Masking) &	0 0 0 1	(✓)	0 0 0 0	(✕)	0 1 0 0	(✓)	0 0 0 0	(✕)
(2) Addition (Set Bits) |			0 1 1 1				1 1 0 1	

Figure 7: Example of bitwise operation.

weight is computed and compared with C^*. If the new cycle is more negative, C^* is updated (Lines 19). At the end, C^* is returned.

Complexity. The overall time complexity of Algorithm 3 is $O(2^k \cdot |V| \cdot |E|)$. This is because in the worst case each vertex can have up to 2^k distinct DP states (one for each subset of the k colors), and for each state the algorithm processes all outgoing edges.

Example 3.9. Figure 6 illustrates a traversal starting from v_0 to find a 4-hop cycle. The algorithm begins by traversing from v_0 through v_1 and v_2, updating the DP table accordingly. Upon reaching v_3, it detects that two paths share the same color set and updates $dp[v_3]$ with the smallest weight as the state. The traversal then continues to v_4 and v_5, further updating the DP table. While the path reaches k, the algorithm checks for cycles and identifies two negative cycles: C_1 with a weight of -5 and C_2 with a weight of -6. Among them, C_2 is the kMNC.

4 OPTIMIZATIONS

4.1 Bitwise Operation for Color-Coding

We introduce a bitwise color-set representation to accelerate color-coding operations. Instead of using unordered sets with $O(n)$ complexity, we encode each color set as a single integer, with the i-th bit indicating the presence of color i. This enables constant-time ($O(1)$) checks and updates, while reducing memory usage and avoiding pointer overhead. This representation is both efficient and compact.

Operation 1 (Color Checking). *Checks if the color of v_n, $\alpha(v_n)$, is in the color set $\alpha(p)$ via bitwise AND operation:*

$$\text{Check}(\alpha(p), \alpha(v_n)) = \alpha(p) \& (1 \ll \alpha(v_n)),$$

Operation 2 (Color Addition). *Adds the color of v_n, $\alpha(v_n)$, to the set $\alpha(p)$ via bitwise OR operation:*

$$\text{Add}(\alpha(p), \alpha(v_n)) = \alpha(p) \mid (1 \ll \alpha(v_n)),$$

The complexity of Operations 1 and 2 is $O(1)$, as bitwise AND/OR are constant-time regardless of the color set size.

Example 4.1. Consider $k = 4$, each color can be efficiently represented as a 4-bit integer as shown in Figure 7. A color set can also be represented using a 4-bit integer, where each bit position corresponds to a specific color. For example, for $\alpha(p) = \{①, ③\}$, it is encoded as 0101. (1) To check if color ① exists in $\alpha(p)$, perform a bitwise AND with 0001: 0101&0001 = 0001. Since the result is nonzero, color ① exists; and (2) To add color ④, apply a bitwise OR with 1000: 0101 | 1000 = 1101.

4.2 Color-Coding Based Graph Reduction

We introduce a color-guided graph reduction that prunes nodes based on coloring semantics and graph structure to address a key inefficiency in kMNC. Nodes sharing the same color with their only valid neighbor cannot be part of any multicolored cycle and are removed with their edges. This one-pass reduction significantly shrinks the graph before enumeration while preserving correctness. Though simple, this targeted use of color information is novel in the context of color-coding and proves highly effective in practice.

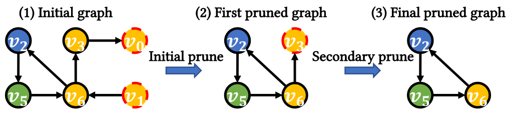


Figure 8: Example of the iterative graph pruning process.

Example 4.2. Figure 8 illustrates the iterative graph pruning process in three steps. (1) In the initial graph, nodes v_0 and v_1, each with one valid neighbor (v_3 and v_6, respectively) and the same color, are marked for removal. (2) After pruning, v_0 and v_1 are removed, and v_3, now with one valid neighbor (v_6), is marked for removal. (3) In the final step, v_3 is removed, leaving v_2, v_5, and v_6 with essential edges for cycle detection.

Lemma 4.3 (Pruning Single-Neighbor Nodes with the Same Color). *Let a be a node in a directed graph G with exactly one neighbor b, and suppose a and b are assigned the same color in an instance. Both a and the edge (a, b) can be safely removed from G without impacting the detection of k-hop multicolored cycles.*

Proof. If a multicolored cycle C contained a, it must include b (since a has only b as a neighbor), which contradicts that all nodes in C have distinct colors. Thus, such a cycle cannot exist. □

5 EXPERIMENTS

5.1 Experiment Setup

Experiments were run on a Linux system with two Intel Xeon Silver 4314 CPUs, 503 GiB RAM, and g++ 11.4.0 (-O3 optimization). Each was repeated three times with different seeds to ensure robustness.

Datasets. Experiments were conducted on six datasets, as detailed in Table 2. These datasets represent graphs where nodes are tokens and edges are exchange pools, constructed from Uniswap V2 liquidity pools at various blocks. On Ethereum, new blocks are produced every 12–15 seconds on average [1]. As block number increases, both node ($|V|$) and edge ($|E|$) counts grow, while average degree decreases, reflecting increasing ecosystem complexity. Each dataset

Table 2: Statistics of experiment datasets.

Dataset	$\|V\|$	$\|E\|$	Avg. Degree	Max Degree
UNI1	11.6K	27.2K	4.69	11,177
UNI2	44.9K	99.7K	4.44	43,152
UNI3	70.4K	151.3K	4.30	68,202
UNI4	148.1K	307.9K	4.16	143,894
UNI5	262.1K	536.0K	4.08	257,868
UNI6	359.6K	734.1K	4.08	354,284

Table 3: Comparison of detection time (in seconds) across datasets. The "Speedup" column shows the speedup of RICH compared to the second-best algorithm. "TLE" indicates that the runtime exceeded 3,600 seconds.

Dataset	RICH	PathEnum	DeFi-ARB	GoldPhish	BriDe	Speedup vs. 2nd
UNI1	0.94	51.42	36.20	1.50	37.23	**1.6x** ⇑
UNI2	2.80	2160.00	551.12	17.65	493.80	**6.3x** ⇑
UNI3	3.64	TLE	1292.53	45.75	594.82	**12.57x** ⇑
UNI4	18.61	TLE	TLE	292.76	955.14	**15.73x** ⇑
UNI5	33.67	TLE	TLE	1011.79	1064.73	**30.05x** ⇑
UNI6	47.52	TLE	TLE	1892.73	1553.31	**32.69x** ⇑

represents the complete token graph accumulated up to a specific block, thereby capturing all pools available at that time.

Baselines. We choose three state-of-the-art solutions for arbitrage detection in cryptocurrency markets. We include a state-of-the-art algorithm for hop-constrained path enumeration, modified for negative cycle detection. The evaluated algorithms are listed below:

- PathEnum (2021) [43] developed an efficient index-based algorithm for hop-constrained s-t path enumeration. We follow the open-source implementation of PathEnum, building query-dependent indexes for each source node. For each node, we identify all incoming neighbors, enumerate hop-constrained s–t path, and reconnect them to the source to form cycles.
- DeFi-ARB (2021) [50] applied BFM-like algorithms to detect negative cycles in directed weighted graphs.
- GoldPhish (2023) [34] applied greedy-based algorithm to identify cyclical arbitrage. We extended the original code, which only detected 3-hop cycles, to handle longer cycles.
- BriDe (2024) [46] applied DFS-like method to detect arbitrage cycles and optimize transaction order to maximize profits.

Metrics. We evaluate *detection time*—how quickly the algorithm finds the kMNC, critical for timely arbitrage [17]—and *most negative weight*, which indicates profit potential (Section 2.1). Detection accuracy is measured by *relative error* (σ): $\sigma = \frac{|w^* - w|}{|w^*|} \times 100\%$, where w^* and w are the optimal and detected cycle weights.

Default Settings. In our experiments, the default parameters are $k = 5$ and $\ell = 30$. For practical reasons such as gas fees and high runtime of exact enumeration, we focus on $k \leq 5$. Prior work shows cycles longer than 5 hops are rare (under 2%) in crypto markets [34, 45]. Additional results for larger k appear in Appendix C [32].

5.2 Efficiency

Overall efficiency. Table 3 compares the detection time of different algorithms across six datasets, showing that RICH consistently outperforms all baselines. Specifically, RICH outperforms GoldPhish (speedup: 1.6×–39.82×) and BriDe (31.62×–176.21×), while PathEnum and DeFi-ARB exceeds the runtime limit when graph size is big. The efficiency of RICH arises from its linear-time techniques and optimizations that minimize redundant computations and reduce the search space. In contrast, other methods experience exponential growth due to exhaustive cycle enumeration, resulting in significantly higher detection times.

Impact of cycle length k (Figure 9). We evaluate the efficiency of cycle detection by varying the cycle hop k from 3 to 6. *First*, compared to enumeration-based methods such as BriDe and PathEnum, RICH demonstrates increasing advantages as k grows. Although BriDe slightly outperforms RICH at $k = 3$, RICH achieves 32× to 176× speedups by $k = 5$ and 330× to 2194× at $k = 6$. This highlights that RICH reduces redundant traversals for longer cycles. In contrast, enumeration methods become exponentially slower with k due to redundant path explorations. *Second*, compared to greedy methods such as GoldPhish and DeFi-ARB, RICH maintains an advantage, albeit with diminishing returns as k increases. This trend reflects the greedy nature of GoldPhish and DeFi-ARB, which reduce their search space as k increases while missing some cycles. We also evaluate RICH for larger k in Appendix C [32].

Scalability (Figure 10). We evaluate the scalability of RICH on datasets of varying sizes, with $|E|$ ranging from 27K to 734K. As shown in Figure 10, when the graph size increases, RICH demonstrates a clear scalability advantage over other baselines. Compared to the second-best method, RICH achieves a substantial speedup, ranging from 1.6× on the smallest dataset to 32.69× on the largest. These results highlight RICH's superior scalability. These results showcase RICH's superior scalability.

Impact of dynamic programming (Figure 11). Figure 11 compares the detection time of kMNC using DFS (Algorithm 2) and DP (Algorithm 3). As shown, RICH-DP achieves a speedup of 1.11× to 4.62× over the DFS-based approach. DFS-based detection suffer from redundant path traversals, leading to inefficiencies in large graphs. In contrast, the DP-based approach effectively eliminates these redundancies by storing and reusing intermediate results.

Ablation study (Figure 11). We analyze the impact of different optimizations in RICH by evaluating the detection time across five variants: (1) **Full**: The complete RICH algorithm with both graph reduction and bitwise operations. (2) **w/o bit operations**: The same as Full, except removing bitwise operations. (3) **w/o graph reduction**: The same as Full, except removing graph reduction. (4) **w/o both optimizations** (DP): A baseline dynamic programming approach without bitwise operations or graph reduction.

(1) **Impact of removing bit operations**: Excluding bitwise operations slows down RICH (Full) by 52.6% to 90.3%, with larger graphs exhibiting a more pronounced slowdown. This is because bitwise operations significantly accelerate state transitions in the dynamic programming process, particularly in large graphs where frequent state updates are required.
(2) **Impact of removing graph reduction**: Excluding graph reduction slows RICH by 2.2% to 9.8%, despite reducing graph size by over 20%. This indicates that while reducing the graph size lowers traversal cost, its direct impact on overall efficiency is smaller compared to bitwise operations.

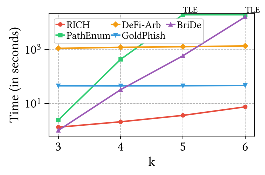


Figure 9: Vary k on `UNI3`.

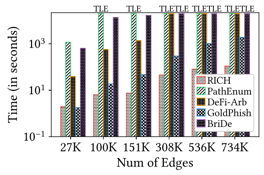


Figure 10: Scalability.

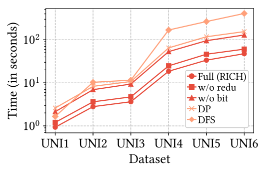


Figure 11: RICH variants.

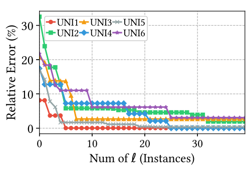


Figure 12: Vary ℓ.

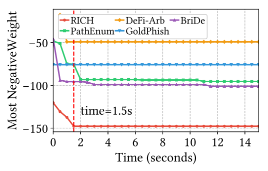


Figure 13: Time-Weight.

Table 4: Maximum memory consumption (MB).

Dataset	$k = 3$	$k = 4$	$k = 5$	$k = 6$
UNI1	59.8	65.0	67.7	69.5
UNI2	207.3	227.4	239.0	249.0
UNI3	310.2	342.9	361.9	376.9
UNI4	621.5	721.9	825.3	941.6
UNI5	1,077.0	1,245.5	1,432.5	1,668.2
UNI6	1,473.1	1,712.6	1,923.1	2,290.5

Table 5: Most negative cycle weights across datasets. For timeouts, the best result within 3,600 seconds is reported.

Dataset	RICH	PathEnum	DeFi-ARB	GoldPhish	BriDe	σ (%)
UNI1	-91.12	-91.14	-81.17	-91.14	-91.14	0.02
UNI2	-142.47	-148.25	-102.66	-148.25	-148.25	3.90
UNI3	-147.78	-151.87	-126.79	-151.87	-151.87	2.70
UNI4	-139.49	-110.28	-93.60	-139.49	-139.49	0.00
UNI5	-117.92	-106.41	-42.15	-118.20	-118.20	0.24
UNI6	-151.87	-126.48	-130.95	-156.69	-156.69	3.08

(3) **Impact of removing both optimizations**: When both bitwise operations and graph reduction are removed (DP), performance drops drastically, with slowdowns ranging from 59.3% to 99.6%. This highlights that the combination of both optimizations is crucial for achieving significant speed improvements, and their effects are complementary in improving efficiency.

Memory cost. Table 4 shows memory usage for $k = 3$ to 6, which grows with both k and graph size as expected ($O(|V|2^k)$). Even for the largest graph, usage stays under 1% of available memory, demonstrating that RICH is memory-efficient and scalable.

5.3 Effectiveness

Overall effectiveness. Table 5 compares the most negative weights detected by each algorithm within 3,600 seconds limit. Among the baselines, BriDe and PathEnum provide exact results, while DeFi-ARB and GoldPhish are greedy methods that do not guarantee exactness. For experiments that exceed the runtime limit, the most negative weight detected within the time limit is recorded. As shown, while RICH is a probabilistic method, it consistently identifies near-optimal negative weights, with relative errors ranging from 0.02% to 3.90%. This performance reflects RICH's ability to balance computational efficiency and accuracy effectively. In contrast, DeFi-ARB and PathEnum show noticeable discrepancies compared to the global optimal detected by BriDe, while GoldPhish, as a greedy method, lacks accuracy guarantees.

Impact of number of color coding instances ℓ (Figure 12). Theoretically, the number of ℓ required to detect C^* depends on the cycle length k and the desired detection probability δ. To ensure that the probability of detecting C^* is at least δ, ℓ should satisfy $\ell \geq \frac{\log(1-\delta)}{\log\left(1-\frac{k!}{k^k}\right)}$. We conducted experiments to evaluate the impact of varying ℓ on both solution quality and execution time. Across all datasets, the relative error with respect to the global optimum decreases rapidly at first and then gradually plateaus. With $\ell = 30$, the relative error drops below 5% across all datasets.

Time-weight trade-off (Figure 13). We evaluate the trade-off between runtime and solution quality by comparing the most negative weight detected by different algorithms as time progresses. Since coloring instances are assigned randomly, the detection time of RICH scales proportionally with the number of instances ℓ. Within 15 seconds, RICH finds the most negative cycle, outperforming others by over 45% while competitors remain stuck in local optima. This shows that, despite being probabilistic, RICH quickly reaches high-quality solutions, making it practical for arbitrage trading.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented RICH, an efficient algorithm for detecting k-hop Most Negative Cycles in graphs—an essential capability for real-time arbitrage detection in financial markets. By combining color-coding with dynamic programming and optimizing with bit-level operations and graph reduction, RICH achieves up to **32.69×** speedup over existing methods. Case studies (Appendix E [32]) show RICH provides accurate, real-time arbitrage insights in both forex and crypto markets. The framework is versatile and can be applied to other financial domains. Future work (Appendix D [32]) includes parallelization and support for dynamic or streaming graphs.

Disclaimer. RICH is developed exclusively for academic research. Its deployment in real-world trading scenarios must account for regulatory and ethical considerations. Any practical use is at the user's own risk and should carefully consider multiple market factors such as slippage and execution delays.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of Tokka Labs for funding this research. This research / project is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] [n.d.]. *Ethereum Average Block Time Chart.* https://etherscan.io/chart/blocktime Accessed: December 8, 2024.

[2] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap V2 Core. Whitepaper. https://uniswap.org/whitepaper.pdf Accessed: 2024-12-04.

[3] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *Journal of the ACM (JACM)* 42, 4 (1995), 844–856.

[4] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[5] Jan Arvid Berg, Robin Fritsch, Lioba Heimbach, and Roger Wattenhofer. 2022. An empirical study of market inefficiencies in Uniswap and SushiSwap. In *International Conference on Financial Cryptography and Data Security.* Springer, 238–249.

[6] Jean-Claude Bermond and Carsten Thomassen. 1981. Cycles in digraphs–a survey. *Journal of Graph Theory* 5, 1 (1981), 1–43.

[7] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. 2022. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd annual symposium on foundations of computer science (FOCS).* IEEE, 600–611.

[8] Karl Bringmann, Alejandro Cassis, and Nick Fischer. 2023. Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS).* IEEE, 515–538.

[9] Rohit Chandra. 2001. *Parallel programming in OpenMP.* Morgan kaufmann.

[10] Yong Chen, Zhi Da, and Dayong Huang. 2019. Arbitrage trading: The long and the short of it. *The Review of Financial Studies* 32, 4 (2019), 1608–1646.

[11] Yuhang Chen, Jiaxin Jiang, Shixuan Sun, Bingsheng He, and Min Chen. 2024. Rush: Real-time burst subgraph detection in dynamic graphs. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3657–3665.

[12] Boris V Cherkassky, Loukas Georgiadis, Andrew V Goldberg, Robert E Tarjan, and Renato F Werneck. 2010. Shortest-path feasibility algorithms: An experimental evaluation. *Journal of Experimental Algorithmics (JEA)* 14 (2010), 2–7.

[13] Boris V Cherkassky and Andrew V Goldberg. 1999. Negative-cycle detection algorithms. *Mathematical Programming* 85, 2 (1999).

[14] Tianyang Chi, Ningyu He, Xiaohui Hu, and Haoyu Wang. 2024. Remeasuring the Arbitrage and Sandwich Attacks of Maximal Extractable Value in Ethereum. *arXiv preprint arXiv:2405.17944* (2024).

[15] Mina Dalirrooyfard, Thuy Duong Vuong, and Virginia Vassilevska Williams. 2019. Graph pattern detection: Hardness for all induced patterns and faster non-induced cycles. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing.* 1167–1178.

[16] Philip H Dybvig and Stephen A Ross. 1989. Arbitrage. In *Finance.* Springer, 57–71.

[17] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2019. SoK: Transparent Dishonesty: front-running attacks on Blockchain. *CoRR* abs/1902.05164 (2019). arXiv:1902.05164 http://arxiv.org/abs/1902.05164

[18] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2020. Sok: Transparent dishonesty: front-running attacks on blockchain. In *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23.* Springer, 170–189.

[19] Lester Randolph Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian journal of Mathematics* 8 (1956), 399–404.

[20] Giorgio Gallo and Stefano Pallottino. 1988. Shortest path algorithms. *Annals of operations research* 13, 1 (1988), 1–79.

[21] Robert Gębarowski, Paweł Oświęcimka, Marcin Wątorek, and Stanisław Drożdż. 2019. Detecting correlations and triangular arbitrage opportunities in the Forex by means of multifractal detrended cross-correlations analysis. *Nonlinear Dynamics* 98, 3 (2019), 2349–2364.

[22] Pierre-Louis Giscard, Nils Kriege, and Richard C Wilson. 2019. A general purpose algorithm for counting simple cycles and simple paths of any length. *Algorithmica* 81 (2019), 2716–2737.

[23] Andrew V Goldberg. 1995. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.* 24, 3 (1995), 494–504.

[24] Andrew V Goldberg and Tomasz Radzik. 1993. *A heuristic improvement of the Bellman-Ford algorithm.* Citeseer.

[25] Anshul Gupta and Toyotaro Suzumura. 2021. Finding all bounded-length simple cycles in a directed graph. *arXiv preprint arXiv:2105.10094* (2021).

[26] Harald Hau. 2014. The exchange rate effect of multi-currency risk arbitrage. *Journal of International Money and Finance* 47 (2014), 304–331.

[27] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. 2022. Spade: A real-time fraud detection framework on evolving graphs. *Proceedings of the VLDB Endowment* 16, 3 (2022), 461–469.

[28] Donald B Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1975), 77–84.

[29] Adam Karczmarz. 2024. Max s, t-Flow Oracles and Negative Cycle Detection in Planar Digraphs. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* SIAM, 1606–1620.

[30] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[31] Sinan Krückeberg and Peter Scholz. 2020. Decentralized efficiency? Arbitrage in bitcoin markets. *Financial Analysts Journal* 76, 3 (2020), 135–152.

[32] Bingqiao Luo, Jiaxin Jiang, and Yuhang Chen et al. 2025. *RICH: Real-Time Identification of Negative Cycles for High-Efficiency Arbitrage.* Technical Report. Technical Report. Available online at https://drive.google.com/drive/folders/13VD_zcsp2TPs1R3eG1JVG91mV6JBaQlm?usp=sharing. Accessed on May 31, 2025.

[33] Bingqiao Luo, Zhen Zhang, Qian Wang, and Bingsheng He. 2024. Multi-Chain Graphs of Graphs: A New Approach to Analyzing Blockchain Datasets. *Advances in Neural Information Processing Systems* 37 (2024), 28490–28514.

[34] Robert McLaughlin, Christopher Kruegel, and Giovanni Vigna. 2023. A large scale study of the ethereum arbitrage ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23).* 3295–3312.

[35] Gary L Miller. 1984. Finding small simple cycle separators for 2-connected planar graphs. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing.* 376–382.

[36] Edward F Moore. 1959. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching.* Harvard University Press, 285–292.

[37] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained s-t simple path enumeration. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 463–476. https://doi.org/10.14778/3372716.3372720

[38] Julien Piet, Jaiden Fairoze, and Nicholas Weaver. 2022. Extracting godl [sic] from the salt mines: Ethereum miners extracting value. *arXiv preprint arXiv:2203.15930* (2022).

[39] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP).* IEEE, 198–214.

[40] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[41] Igor Radovanovic. [n.d.]. Crypto Arbitrage Guide. https://algotrading101.com/learn/crypto-arbitrage-guide/. Accessed: 2025-01-30.

[42] Stephen A Ross. 2013. The arbitrage theory of capital asset pricing. In *Handbook of the fundamentals of financial decision making: Part I.* World Scientific, 11–30.

[43] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. Pathenum: Towards real-time hop-constrained st path enumeration. In *Proceedings of the 2021 international conference on management of data.* 1758–1770.

[44] Robert Tarjan. 1973. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.* 2, 3 (1973), 211–216.

[45] Ye Wang, Yan Chen, Haotian Wu, Liyi Zhou, Shuiguang Deng, and Roger Wattenhofer. 2022. Cyclic arbitrage in decentralized exchanges. In *Companion Proceedings of the Web Conference 2022.* 12–19.

[46] Hulin Yang, Mingzhe Li, Jin Zhang, Alia Asheralieva, Qingsong Wei, and Siow Mong Rick Goh. 2024. BriDe Arbitrager: Enhancing Arbitrage in Ethereum 2.0 via Bribery-enabled Delayed Block Production. *arXiv preprint arXiv:2407.08537* (2024).

[47] Siyuan Yao, Yuchen Li, Shixuan Sun, Jiaxin Jiang, and Bingsheng He. 2024. ublade: Efficient batch processing for uncertainty graph queries. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–24.

[48] Yu Zhang, Tao Yan, Jianhong Lin, Benjamin Kraner, and Claudio J Tessone. 2024. An Improved Algorithm to Identify More Arbitrage Opportunities on Decentralized Exchanges. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE, 1–7.

[49] Zhen Zhang, Bingqiao Luo, Shengliang Lu, and Bingsheng He. 2023. Live graph lab: Towards open, dynamic and real transaction graphs with NFT. *Advances in Neural Information Processing Systems* 36 (2023), 18769–18793.

[50] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. 2021. On the just-in-time discovery of profit-generating transactions in defi protocols. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 919–936.