ORDER: Optimal Routing with Path Indexing in Exchange Graph [Data-Intensive & Data Science Application]

BINGQIAO LUO, National University of Singapore, Singapore
YUHANG CHEN, National University of Singapore, Singapore
YUHENG CONG, Shanghai Jiao Tong University, China
ZIYU HE, Shanghai Jiao Tong University, China
JIAXIN JIANG*, National University of Singapore, Singapore
SHIXUAN SUN, Shanghai Jiao Tong University, China
BINGSHENG HE, National University of Singapore, Singapore
WEE HOWE ANG, Tokka Labs, Singapore

We study the problem of *optimal routing* on financial exchange graphs. Given a directed graph $G = (V, E)$ where vertices represent assets and edges encode tradable pairs with effective exchange rates and capacities, the goal is to find a sequence of routes from a source asset to a target asset that maximizes the delivered output. This problem is data-intensive and time-critical, requiring real-time decisions under fragmented liquidity, size-dependent prices, heterogeneous fees, and tight capacity limits. To address the challenges, we present ORDER: **O**ptimal **R**outing with path in**D**exing in **E**xchange g**R**aphs, a high-efficiency routing solution for fast, iterative execution in financial exchange graphs. ORDER introduces a hierarchical bucket path index that localizes maintenance to impacted candidates and eliminates expensive global rescans. It further incorporates lazy computation and an adaptive controller for active-bucket sizing to stabilize update cost under discrete tier shifts and heterogeneous market depths. On real-world DeFi datasets, ORDER achieves up to **114**× speedup over state-of-the-art routers while preserving execution quality. We also demonstrate robustness across market regimes and token pairs, enabling timely, high-quality routing.

CCS Concepts: • **Applied computing** → **Digital cash**.

Additional Key Words and Phrases: decentralized finance, routing, graph algorithms

ACM Reference Format:
Bingqiao Luo, Yuhang Chen, Yuheng Cong, Ziyu He, Jiaxin Jiang, Shixuan Sun, Bingsheng He, and Wee Howe Ang. 2026. ORDER: Optimal Routing with Path Indexing in Exchange Graph [Data-Intensive & Data Science Application]. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 220 (June 2026), 27 pages. https://doi.org/10.1145/3802097

*Corresponding author.

Authors' Contact Information: Bingqiao Luo, National University of Singapore, Singapore, Singapore, luo.bingqiao@u.nus.edu; Yuhang Chen, National University of Singapore, Singapore, Singapore, yuhangc@comp.nus.edu.sg; Yuheng Cong, Shanghai Jiao Tong University, Shanghai, China, shineeternal@sjtu.edu.cn; Ziyu He, Shanghai Jiao Tong University, Shanghai, China, box-fish@sjtu.edu.cn; Jiaxin Jiang, National University of Singapore, Singapore, Singapore, jiangjx@comp.nus.edu.sg; Shixuan Sun, Shanghai Jiao Tong University, Shanghai, China, sunshixuan@sjtu.edu.cn; Bingsheng He, National University of Singapore, Singapore, Singapore, dcsheb@nus.edu.sg; Wee Howe Ang, Tokka Labs, Singapore, Singapore, weehowe.ang@tokkalabs.com.

ACM 2836-6573/2026/6-ART220
https://doi.org/10.1145/3802097

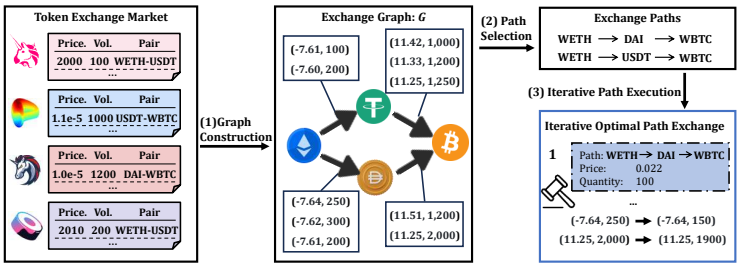


Fig. 1. Data pipeline for optimal routing.

1 Introduction

Graphs provide a natural abstraction for financial systems: vertices represent assets, venues, or accounts, and edges encode feasible transfers or conversions with associated costs and capacities [37, 58]. A central task on such graphs is *routing*, the process of converting one token into another by traversing a sequence of exchanges (edges) through intermediate assets. *Optimal routing* further requires deciding not only which paths to take but also how to allocate the input token amount across multiple paths to maximize final output under fragmented liquidity, heterogeneous fees, and dynamically changing pool states. This is critical because the realized outcome of a transaction depends not only on the chosen destination but also on how liquidity is sourced along alternative paths. For example:

(1) *Token exchange on decentralized markets [7, 18].* On *token exchange graphs* built from DEXs (decentralized exchanges without centralized intermediaries), the same target token can often be reached through different intermediates. For example, WBTC can be swapped from WETH via DAI (WETH→DAI→WBTC) or via USDT (WETH→USDT→WBTC). These alternative routes may exist across venues or within a single DEX, and their effective prices differ due to slippage (price impact caused by limited liquidity), transaction fees, and gas costs. As a result, the same WETH input can produce different amounts of WBTC depending on the chosen path.

(2) *Capacity-constrained execution over fragmented venues [23, 26, 53].* In many exchange settings, liquidity is fragmented across venues and price levels. Each venue offers only *finite* volume at the best rate, and consuming it forces execution onto worse levels. As a result, effective routing must *jointly select multiple paths/venues and decide the split ratio* to aggregate liquidity and minimize realized cost.

Industry-standard Pipeline for Optimal Token Routing. In this paper, we focus on optimal routing in cryptocurrency markets, which provides publicly-accessible, real-time data of cryptocurrency transactions across a large and diverse set of tokens [38, 59]. In collaboration with our industry partner ABC Company, we summarize an industry-standard optimal-routing pipeline in Figure 1: (1) **Graph construction:** aggregate prices and available volumes from token exchange markets, and model each executable price level as a directed edge with a *weight* (marginal cost/price) and a *capacity* (available volume), yielding an exchange graph G. Executing volume on an edge consumes its capacity and may shift the edge to a worse price level, i.e., updating edge weights for subsequent routing decisions. (2) **Path selection:** choose candidate routes from source to target using current edge weights and capacities. (3) **Iterative path execution:** execute the chosen route in chunks, consuming the best-priced levels first; after each fill, update remaining capacities and marginal prices, then select the next best route. In other words, execution feeds back to the graph state by depleting capacities and changing marginal costs, which necessitates an iterative

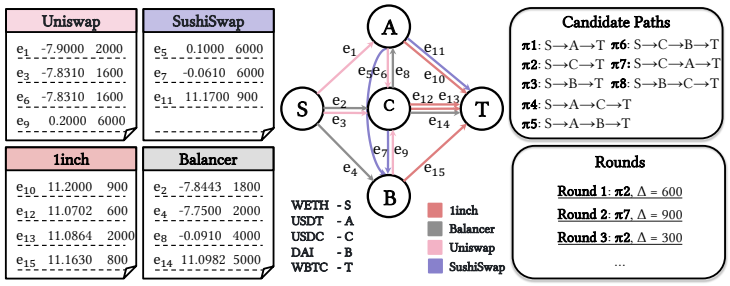
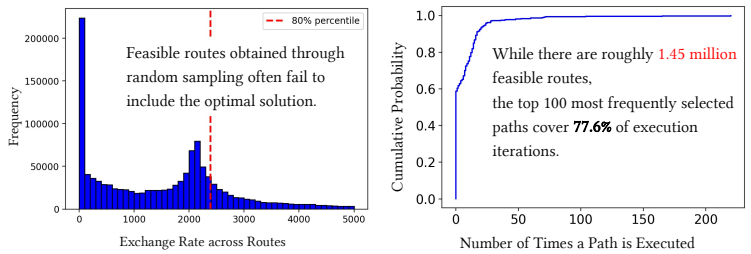


Fig. 2. Example of graph construction and optimal routing.

select–execute–update loop. In this paper, we focus on the self-induced feedback of a single swap execution under a fixed snapshot. This pipeline enables us to capture fleeting routing opportunities in rapidly evolving crypto markets, and closely mirrors the execution workflow adopted by major commercial aggregators such as 0x (ZRX) [50], ODOS [42], 1INCH [2], and other industry routers. Although the above workflow is conceptually simple, scaling this process reveals key computational traits arising from price dynamics and shared liquidity. We illustrate the iterative execution process with a running example below.

Example 1.1. Consider an exchange of a total quantity $Q = 2000$ from S to T on an exchange graph. For each node pair (u, v), there may exist multiple parallel edges, corresponding to different venues, liquidity sources, or price tiers (Figure 2). A small set of candidate paths is maintained. After each execution, if an edge is fully or partially consumed, the router updates the affected path weights based on the residual graph and reranks the candidate paths accordingly.
Round 1. The router selects π_2 with the smallest weight and executes $\Delta_1 = 600$ using edges e_2 and e_{12}.
Round 2. The affected path weights are updated, and π_7 becomes optimal. The router then executes $\Delta_2 = 900$ along edges e_2, e_5, and e_{11}.
Round 3. π_7 becomes more expensive after its low-weight residual capacity is consumed, while π_2 remains feasible via e_{13} on $C \longrightarrow T$. Hence, π_2 becomes minimum again, and the router executes $\Delta_3 = 300$.
Subsequent rounds. It repeatedly selects the minimum-weight path and executes the maximum feasible amount until demand is met or no feasible path remains.

There are three key characteristics of optimal routing in cryptocurrency markets. *First, heterogeneous prices and limited depth* induce a natural ordering over executable price levels. For the same asset pair, quotes differ across venues; even within one venue (e.g., a DEX), they vary across liquidity pools and fee settings. Under Automated Market Makers (AMMs), which deterministically update prices as available liquidity is consumed, the marginal rate worsens monotonically with executed volume. Thus, any plan that skips a cheaper level and consumes a more expensive one is dominated; execution should proceed from better to worse levels, switching levels once the current level no longer has sufficient remaining capacity. *Second,* edge weights and capacities *evolve rapidly in the live market.* Quotes change as trades execute and as liquidity is added or removed; on Ethereum, a new block arrives roughly every 12 seconds [1]. Moreover, within a block, earlier transactions can invalidate a previously computed route, leading to unacceptable price impact, failed execution, or missed opportunities. *Third,* the search space is huge, and capacities are *shared*



(a) Distrib. of exchange rates. (b) CDF of selection-counts.

Fig. 3. Example of challenges in a token exchange graph.

and tightly coupled across routes. Millions of multi-hop routes may exist, yet usable depth is finite and overlapping: consuming one edge's current price level reduces its remaining capacity and may activate a worse level, immediately reweighting many other routes that share that edge.

Taken together, the properties make routing both path-dependent and highly dynamic, necessitating an iterative *select–execute–update* cycle rather than a one-shot static solve. We next identify the main computational challenges arising from these characteristics:

Challenge 1: Route selection exhibits highly skewed quality. Even for the same source–target pair, different routes can yield sharply different outcomes due to heterogeneous edge weights, fees, and limited capacities, resulting in large execution variance. For example, for a WETH→USDC swap at Ethereum block 22,000,000, there are more than 1.45M feasible paths, while achievable rates span over three orders of magnitude ($P_{99}/P_1 > 10^3$). Randomly sampled routes show large dispersion: some are near-optimal, while others incur severe losses. This highlights the need for optimal routing, as naive or greedy choices can deviate sharply from best execution.

Challenge 2: Discrete price-level changes trigger widespread reweighting. Because executing volume depletes capacities and can switch an edge to a worse price level, any static plan becomes stale after each partial fill; routing must therefore proceed iteratively with localized updates. Meanwhile, a single execution can influence many candidate routes at once: when a trade drains an edge's current price level (tier), the next level activates with a new price and capacity, altering both the route weight and the bottleneck for every route that uses that edge. For example, in the WETH→USDC *swap*, a single execution that touches the DAI–USDC edge(s) (e.g., pools) can reweight more than 70K five-hop routes containing that edge. Without a mechanism to identify and update *only the affected routes/edges*, naive rescans repeatedly traverse nearly the entire catalog, resulting in prohibitive per-iteration costs.

Challenge 3: Skewed path popularity concentrates computation. In practice, a small set of paths accounts for most *selections across iterations*, while a long tail is rarely chosen. This reflects a long-tailed distribution of usable capacity across venues and price levels: most liquidity concentrates in a few popular options, while the rest is thin. As Figure 3b shows for swapping 100 WETH to USDT, although roughly 1.45M routes exist within 5 hops, the top 100 most frequently selected paths in the iterative execution cover 77.60% of iterations. Such skew makes naive global re-ranking or full shortest-path searches at every step wasteful, spending cycles on candidates that almost never win.

To address these challenges, we introduce a bucket-indexed data structure for iterative optimal path execution. Our contributions are summarized as follows:

- **Query formulation.** We formalize the *Iterative Optimal-Path Exchange (*IOPE*)* query, which captures liquidity-aware, path-dependent routing on token exchange graphs. IOPE models iterative execution by re-optimizing after each partial fill within a fixed market snapshot, where the

only state changes are the capacity/price-level updates induced by the query's own executions (i.e., the same *swap*).

- **Iterative Framework.** We present ORDER, a framework for efficient IOPE queries that iteratively extend optimal routing sequences without full recomputation. It preserves correctness under dynamic price and capacity changes while enabling low-latency updates. Source code and data are publicly available[1].
- **Prioritized bucket index.** We design a prioritized bucket index that groups edges by price tiers and links adjacent buckets for incremental maintenance. Each update touches only a bounded set of buckets affected by the latest execution, converting global rescans into localized index operations and sustaining high-throughput iterative optimization.
- **Adaptive bucket sizing.** We propose an online controller that dynamically adjusts the active-set size K. Through smoothing and guarded probing, it adaptively balances scan cost and heap maintenance across different tokens and market regimes, thereby optimizing overall throughput under heterogeneous liquidity depth, volatility, and tier dynamics.
- **Evaluation and applications.** We conduct extensive experiments on real on-chain data, showing that ORDER outperforms existing routing systems by up to **114**× times. Case studies on live DeFi data and comparisons with industry routers further demonstrate its practicality and robustness in production environments.

2 Background and Related Work

In this section, we introduce essential background, preliminaries, and formally state the problem addressed in this paper. Frequently used notations are summarized in Table 1.

2.1 Background

Blockchain. A blockchain is a decentralized ledger that groups transactions into cryptographically linked blocks [49]. Consensus protocols determine the canonical chain and provide evidence and eventual finality. Blocks are produced at a characteristic *block time*, which governs how quickly new transactions are confirmed and appended. A block time is very fast, for example, Ethereum targets about 12 seconds per block [1], while BNB Smart Chain targets about 0.75 seconds [55].

Crypto Token. Crypto tokens are digital assets on blockchains. Common categories include stablecoins (e.g., USDT, DAI) for price stability, governance tokens (e.g., UNI) for protocol voting, utility tokens (e.g., LINK) for decentralized services, and wrapped assets (e.g., WETH, WBTC) representing value bridged from other forms. Tokens typically conform to interface standards, enabling composability across exchanges, lending markets, and other DeFi applications. Traders swap tokens for speculation, hedging, portfolio rebalancing, or to access protocol functionality.

Decentralized Exchanges (DEXs). Decentralized exchanges (DEXs) are non-custodial on-chain marketplaces for token swaps, eliminating intermediaries. Built around liquidity pools rather than order books, they let anyone provide liquidity for fees and power DeFi price discovery, liquidity, and cross-asset conversion.

Automated Market Makers (AMMs). Automated Market Makers (AMMs) are DEX designs that replace order books with deterministic pricing. Trades occur against token pools, with prices set by an invariant (e.g., $xy = k$ in Uniswap V2). Different AMM variants, such as constant-product (Uniswap V2), concentrated liquidity (Uniswap V3), and stable-swap (Curve) trade off capital efficiency, slippage, and risk, forming the core of modern on-chain trading.

[1] https://github.com/Xtra-Computing/ORDER

Table 1. Frequently used notations

Notation	Meaning
$G = (V, E)$	Directed token-exchange graph (nodes V are tokens; edges E are tradable pairs).
E_g	Edge group g, implemented as a deque of price/capacity tiers (front is effective).
$\Pi = \{\pi_0, \ldots, \pi_{n-1}\}$	Set of explicit candidate paths admitted/considered by the index.
π_i	Path i as a sequence of edge-group identifiers.
(P, X)	Edge–path pointer: catalog P and postings X.
$P[i] = (\mathsf{W}_i, \mathsf{C}_i, \pi_i)$	Catalog entry: path weight key W_i, bottleneck capacity C_i, and group sequence.
$X[g] = \{\, i \mid g \in \pi_i \,\}$	Posting list: all in-bucket paths that contain group g.
W_i	Ordering key of π_i: $\mathsf{W}_i = \sum_{g \in \pi_i} w_{g0}$.
C_i	Executable bottleneck capacity of π_i.
$r(u,v),\ w(u,v),\ c(u,v)$	Exchange rate, edge weight, and edge capacity.
s, t, Q	Source token, target token, and initial quantity for the IOPE query.
Q_{rem}	Remaining quantity during the execution.
$\text{capacity}(\pi)$	On-demand path capacity operator (min residual along π at selection time).
$B_{\text{act}},\ B_{\text{res}}$	Active and reserve buckets.
$K,\ \tau$	Active bucket size and its admission threshold.
$\mathcal{B} = \{B_0, \ldots, B_{L-1}\}$	Priority (multi-level) buckets; $B_0 \equiv B_{\text{act}}$.
$\lvert\mathcal{P}\rvert$	Total number of paths under consideration in an epoch/analysis.
T	Number of path-selection iterations drained within an epoch.
α	Bucket utilization of an epoch.
$K^\star$	Asymptotically optimal active-bucket size.
m, n	Number of edge groups and valid paths.
$\mathcal{A}$	Affected-path set sharing at least one group with the executed path.

Table 2. Comparison of ORDER and previous algorithms.

Feature	EMUF [35]	GoldPhish [40]	BriDe [54]	DeFi-ARB [60]	ORDER (Ours)
Iterative Query Model	✗	✗	✗	✗	✓
Localized Update Mechanism	✗	✗	✗	✗	✓
Traversal Efficiency	✓	✗	✗	✓	✓
Correctness Guarantee (w.r.t. IOPE)	✓	✗	✓	✓	✓

2.2 Preliminaries

Exchange Graph. We model the token exchange market as a directed, *multi directed graph* $G = (V, E)$, where vertices V denote tokens and edges E denote feasible swaps. For each ordered token pair (u, v), there may exist multiple parallel edges, corresponding to different venues, liquidity sources, or price tiers.

Path. A path specifies only the token sequence $P = (v_1, v_2, \ldots, v_k)$. Each hop (v_i, v_{i+1}) indicates that at least one feasible exchange from token v_i to v_{i+1} exists in the market. Unless otherwise stated, we consider simple paths, i.e., paths with distinct vertices.

Edge Weight and Capacity. Each edge $(u, v) \in E$ represents an exchange from u to v and may expose multiple price levels (tiers) with associated executable volumes. We denote a level by two attributes: (i) the effective exchange rate $r(u, v)$, and (ii) its available volume (capacity) $c(u, v)$. To linearize the multiplicative nature of chained rates, we define the edge weight $w(u, v) = -\log\big(r(u \to v)\big)$, so that path weights add and finding the most advantageous route reduces to minimizing total weight. Transaction fees and slippage can be incorporated by defining $r(\cdot)$ as the

net effective rate after such adjustments. The capacity $c(u, v)$ encodes the maximum executable volume supported at the corresponding rate, and tiered liquidity can be modeled piecewise across parallel edges.

Example 2.1. Consider the path $S \to B \to T$ with edges e_4 ($S \to B$) and e_{15} ($B \to T$) in Figure 2. Let $r_1 = r(S \to B)$ and $r_2 = r(B \to T)$ denote the effective marginal rates, and define edge weights by the standard transform $w = -\log r$. Then $w_1 = -\log r_1$, $w_2 = -\log r_2$, and the path weight is

$$w_\pi = w_1 + w_2 = -\log r_1 - \log r_2 = -\log(r_1 r_2).$$

Minimizing w_π is therefore equivalent to maximizing the effective exchange rate $r_1 r_2$.
Note. With $w = -\log r$, edges and thus paths have *negative* weight whenever $r > 1$. A directed path with negative total weight means the product of effective rates exceeds 1, so for a fixed source amount the target received is greater than the original. Hence, a more negative path weight indicates a more favorable exchange.

2.3 Problem Statement

In real-world decentralized exchanges, swap routers and aggregators do not attempt to compute a globally optimal routing plan for the entire input amount in advance. Instead, they follow an *iterative execution paradigm*: at each step, the router selects the currently best feasible path under the prevailing liquidity state, executes as much volume as allowed, and then re-optimizes after the residual state is updated. This paradigm is driven by the inherent dynamics of token exchange markets—discrete price tiers, volume-dependent exchange rates, and rapid state changes after each partial execution—under which precomputed global plans are fragile and often invalidated in practice. Accordingly, modern routers prioritize adaptivity and execution robustness over multi-step global lookahead (e.g., 1inch Router v5 [3], Uniswap V3 [47]).

To reason about this industry practice in a principled way, we formalize the routing process as a query over token exchange graphs.

Problem Statement. We formalize the *Iterative Optimal-Path*[2] *Exchange* (IOPE) query on a weighted directed graph $G = (V, E, w)$ with edge capacities $c : E \to \mathbb{R}_{\geq 0}$. Given a source token $s \in V$ with initial quantity $Q \geq 0$ and a target token $t \in V$, the query returns an ordered list of $s \to t$ paths $\mathcal{P} = (\pi_1, \ldots, \pi_n)$.

Iterative Optimal Path Exchange Paradigm (Algorithm 1). This paradigm outlines the iterative optimal path exchange process. It first precomputes a fixed pool of candidate paths and ranks them by initial path weights (Line 1). In each iteration, it calls SelectBestPath based on path weight, ComputeCapacity for that path to determine the executable amount, PathExecution to route Δ amount and update residual capacities (Lines 3- 5). Then, it calls UpdateAffectedPath for each edge and path, and continues until $Q_{\text{rem}} = 0$ or no available path exists (Line 6). The process repeats and incrementally accumulates an ordered allocation sequence until the demand is fully exhausted or no feasible path remains.

Remark. IOPE is defined on a *single fixed graph snapshot*. Across iterations, we re-optimize on the same snapshot, and the only state changes are the residual capacities (and the resulting price-level changes) on edges affected by the query's own prior executions; no exogenous market updates or

[2]Here, "optimal" is defined with respect to the *iterative execution paradigm*: at each iteration, the currently best feasible path under the updated residual snapshot is selected and executed. IOPE does not aim to compute a globally optimal allocation for the entire input amount with multi-step lookahead. Scenarios where a locally best choice leads to worse future continuations are therefore outside the scope of IOPE, and consistent with the behavior of industry routing systems, such as 0x, DODO [22].

Algorithm 1: Iterative Optimal Path Exchange Paradigm

Input: weighted digraph $G = (V, E)$; source s; target t; initial quantity Q, candidate paths P
Output: ordered list $\mathcal{P} = \big((\pi_1, \Delta_1), (\pi_2, \Delta_2), \dots\big)$

Initialize: $\mathcal{P} \leftarrow \emptyset$; $Q_{\text{rem}} \leftarrow Q$
while $Q_{\text{rem}} > 0$ ***and*** ExistsValidPath(G, s, t) **do**
  $\pi \leftarrow$ SelectBestPath(G, s, t) // choose the current best path
  $\Delta \leftarrow \min\big(Q_{\text{rem}}, \text{ComputeCapacity}(\pi)\big)$
  PathExecution(π, Δ) // execute and reduce capacities
  UpdateAffectedPath(G, π, Δ)
  $\mathcal{P} \leftarrow \mathcal{P} \cup (\pi, \Delta)$; $Q_{\text{rem}} \leftarrow Q_{\text{rem}} - \Delta$
return $\mathcal{P}$

concurrent executions by other agents are assumed. Thus, the reactivity in IOPE captures solely the trader's path-dependent liquidity impact within one *swap*, rather than real-time market dynamics.

Moreover, classical min-cost max-flow (MCMF) [44] cannot adequately model token swaps for two key reasons. First, MCMF assumes fixed linear costs, whereas in automated market makers the effective exchange rate is inherently volume-dependent, degrading as more volume is routed (slippage). Second, MCMF computes a one-shot global allocation, but in practice such allocations are unstable: once part of the trade is executed, pool states change and the precomputed plan may no longer hold. In contrast, the IOPE query captures these dynamics by re-optimizing path selection after each partial execution.

System Scope. Building on this abstraction, we develop ORDER: **O**ptimal **R**outing with path in**D**exing in **E**xchange g**R**aphs. ORDER efficiently processes IOPE queries through incremental path indexing and localized updates. Table 2 summarizes the main differences between ORDER and existing routing methods.

3 Design and Implementation

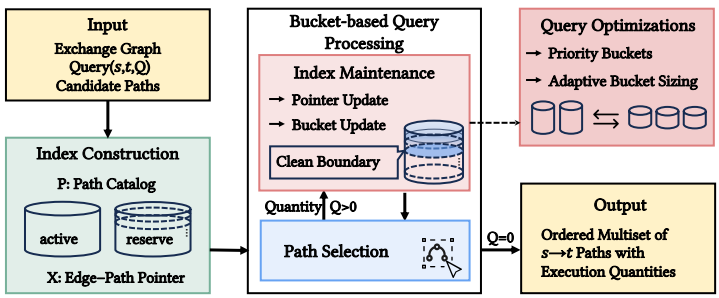


Fig. 4. System workflow of ORDER.

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

(1) *Path Index Construction* (Section 3.1). We build a path index that separates candidates into the *active* bucket for immediate selection and the *reserve* bucket for deferred admission. The index records compact per-path summaries and a lightweight edge–path pointer so that initialization is a single pass over paths and later lookups remain direct.
(2) *Bucket-based Query Processing* (Section 3.2). After each selection, the system updates only the paths affected by the executed route: paths sharing an edge are revised, exhausted paths are

Listing 1. Two-bucket path index: core data structures

```
class PathIndex {
public:
  void Build(Query q, int K);
  PathID Top();
  void Update(PathID i, Cap Delta);
private:
  // Buckets
  MinHeap Active;          // (i, W_i, C_i), size <= K
  Vector<PathID> Reserve;
  // Ordered edge groups: L(g) = [(w0,c0),(w1,c1),...]
  Map<EdgeGroup, Vector<Pair<Weight,Cap>>> L;
  Map<EdgeGroup, int> p;   // front pointer p_g
  // Edge-path pointer EP=(P,X)
  struct Entry { double W; double C; Vector<Vertex> seq; };
  Vector<Entry> P;         // path catalog
  Map<EdgeGroup, Vector<PathID>> X; // postings
};
```

invalidated, and viable ones are repositioned in the active set. Weight changes are applied immediately, while capacity is refreshed on demand when a path is actually selected.

(3) *Prioritized Bucket Optimization* (Section 4.1). We refine the reserve bucket into an *ordered sequence of prioritized buckets* and progressively rebuild the active set following this priority order. A clean-boundary condition halts reconstruction once the prefix of higher-priority buckets remains unaffected, thereby limiting updates to a small prefix of the sequence. This prioritized design confines the rebuild scope, eliminates redundant work, and stabilizes runtime as workloads evolve.

(4) *Adaptive Active Bucket Sizing* (Section 4.2). We develop an adaptive active-bucket sizing scheme that resizes online. It expands when rebuilds are frequent to amortize reconstruction costs and contracts when in-bucket churn dominates to reduce maintenance, with conservative adjustments that prevent oscillation and preserve path-selection correctness.

3.1 Data Structure for Index Construction

We design a two-buckets structure to build the path index for fast IOPE queries (Listing 1). The index organizes a *fixed,* (s,t)*-specific* set of bounded-hop multi-hop swap paths between a given token pair.[3] For each (s,t) query, ORDER operates on this *pre-enumerated* candidate set and adaptively re-optimizes the routing solution by selecting and updating these paths. It maintains two containers: an *active bucket* B_{act}, a size-K priority queue for immediate selection, and a *reserve bucket* B_{res} for remaining candidates.

Definition 3.1 (Ordered Edge Group). For a token pair (u,v), let $E_g(u,v)$ denote the *ordered edge group* that contains all parallel edges from u to v: $E_g(u,v) = [e_0, e_1, \ldots, e_{m-1}]$. Each edge e_ℓ corresponds to a feasible swap and is associated with a weight–capacity pair (w_ℓ, c_ℓ). Edges in $E_g(u,v)$ are ordered by nondecreasing weight. If multiple edges have the same weight, ties are broken arbitrarily but consistently. We denote the current *front* (active price level) of the group by

[3] In production, these pairs are drawn from a stable set of tokens, which is an industry norm [16].

$\textsc{front}(E_g(u,v)) := (w_g, c_g)$, which refers to the minimum-weight edge in the group that still has remaining capacity. Here, m is the number of price levels included in this edge group.

Active Bucket B_{act}. B_{act} maintains the current top-ranked paths with minimal path weights. Specifically, it is a priority queue over K selected paths $(\pi_0, \pi_1, \ldots, \pi_{K-1})$, ordered by their current path weight. Each heap entry stores $(i,\ \mathsf{W}_i,\ \mathsf{C}_i)$, where i is the path id, W_i is the ordering key, and C_i is the executable bottleneck capacity of path π_i. For each hop in π_i, we use the corresponding ordered edge group and its current front $\textsc{front}(\cdot)$ (Definition 3.1). Thus, W_i and C_i are computed over edge groups as: $\mathsf{W}_i = \sum_{g\in\pi_i} w_g$, $\mathsf{C}_i = \min_{g\in\pi_i} c_g$, where $g \in \pi_i$ ranges over the ordered edge groups referenced by the hops of π_i, and $(w_g, c_g) = \textsc{front}(g)$ denotes the current active price level of group g.

Definition 3.2 (Edge–Path Pointer: Catalog and Postings). Let $\Pi = \{\pi_0, \ldots, \pi_{n-1}\}$ be the in-bucket paths stored in B_{act}. Each path π_i is represented as a token sequence $\pi_i = (v_1, \ldots, v_k)$. For each hop (v_j, v_{j+1}) in π_i, let $g_{i,j} := E_g(v_j, v_{j+1})$ denote the ordered edge group referenced by the j-th hop of path π_i. Let $\textsc{front}(g_{i,j}) = (w_{g_{i,j}}, c_{g_{i,j}})$ denote the current active price level of that group.

Path catalog. The catalog P maps each path index i to $P[i] = (\mathsf{W}_i, \mathsf{C}_i, \pi_i)$, where the path weight and bottleneck capacity are computed from the fronts of the referenced edge groups:
$\mathsf{W}_i := \sum_{j=1}^{k-1} w_{g_{i,j}}$, $\mathsf{C}_i := \min_{j=1}^{k-1} c_{g_{i,j}}$.

Posting lists. The posting lists X map each edge group g to the set of path indices that traverse it: $X[g] := \{\, i \mid \exists j \in [1, k-1] \text{ s.t. } g_{i,j} = g \,\}$. We denote $EP := (P, X)$ as the edge–path pointer.

Edge–path (EP) Pointer Construction (Algorithm 2). To efficiently localize updates from edge-group changes to the impacted in-bucket paths, we construct an *edge–path pointer* over B_{act}. Specifically, for each ordered edge group g (i.e., $g = E_g(u,v)$), we initialize an empty postings list $X[g]$ (Line 2). Then, for each in-bucket path $\pi_i \in \Pi$ in B_{act} (Line 3), we initialize running accumulators W_i and C_i (Line 4) and scan its hops in order (Line 5). For each hop (v_j, v_{j+1}), we obtain the corresponding edge group $g \leftarrow E_g(v_j, v_{j+1})$. If g is empty ($|g| = 0$), we skip it (Line 8); otherwise, we read its current front $\textsc{front}(g) = (w_g, c_g)$, accumulate the path weight and bottleneck, and append the path ID to $X[g]$ (Line 11). After the scan, we record the path catalog entry $P[i]$ (Line 12) and, once all paths are processed, return the pointer (P, X) (Line 13). This pointer enables fast access to the in-bucket paths containing a given edge group via $X[g]$. It supports the following two lookup operations:

- $EP_{\text{post}}(g)$: returns the posting list $X[g]$.
- $EP_{\text{path}}(i)$: returns the path catalog entry $P[i] = (\mathsf{W}_i, \mathsf{C}_i, \pi_i)$.

Reserve Bucket B_{res}. B_{res} holds all candidate paths not currently admitted to B_{act}, $(\pi_K, \pi_{K+1}, \ldots, \pi_{n-1})$. It acts as a stable reserve from which candidates are later drawn into B_{act}. At any time, each path resides in exactly one of $\{B_{\text{act}}, B_{\text{res}}\}$, with no overlap. This design confines the update surface to B_{act} and shifts the maintenance cost for most paths into a one-time, on-demand admission cost.

3.2 Index Update for IOPE Queries

After the index is constructed, the path with the optimal weight is iteratively selected and executed across successive rounds. Each execution triggers localized updates: all paths containing any edge of the executed path must refresh their weights and residual capacities, which in turn prompts incremental adjustments to both the edge–path pointer and the bucket index. This section introduces the key update, propagation, and maintenance procedures that ensure efficient consistency throughout this iterative process.

Algorithm 2: Build Edge–Path Pointer for B_{act}

Input: edge groups $EG = \{g_0, \ldots, g_{m-1}\}$; in-bucket paths $\Pi = \{\pi_0, \ldots, \pi_{n-1}\}$ in B_{act}
Output: edge–path pointer (P, X) with catalog $P[\cdot]$ and postings $X[\cdot]$

for $g \leftarrow 0$ **to** $m-1$ **do**
  $X[g] \leftarrow$ empty list `// init postings`
foreach $i \in T_{\text{in}}$ **do**
  $\mathrm{W}_i \leftarrow 0$; $\mathrm{C}_i \leftarrow +\infty$
  foreach $(v_j, v_{j+1}) \in \pi_i$ ***in order*** **do**
    $g \leftarrow E_g(v_j, v_{j+1})$
    if $|g| = 0$ **then**
      continue
    $(w_g, c_g) \leftarrow \textsc{front}(g)$
    $\mathrm{W}_i \leftarrow \mathrm{W}_i + w_g$; $\mathrm{C}_i \leftarrow \min(\mathrm{C}_i, c_g)$;
    append i to $X[g]$
  $P[i] \leftarrow (\mathrm{W}_i, \mathrm{C}_i, \pi_i)$
return (P, X)

Definition 3.3 (Path Selection Iteration). An *iteration* selects the current best path from the active bucket as the exchange path, updates the weights and capacities of its edge groups, and ends after localized maintenance of all affected groups and paths.

Pointer Update. After each iteration, the edges on the executed path either become unavailable when their capacities are fully drained which requires a front replacement at E_g, or experience a capacity decrease when only part of their capacity is consumed. Specifically, the pointer is updated and maintained as follows:

- **Front replacement at E_g.** When the front of E_g is replaced: (i) if E_g becomes empty, all paths in $X[g]$ are removed from the catalog; (ii) otherwise, every path in $X[g]$ has its catalog entry recomputed by traversing its group sequence, thereby reflecting both the front–weight w_{g_0} change and the updated bottleneck capacity c_{g_0} of each path.
- **Capacity decrease without replacement in $X[g]$.** When the capacity of the current front of E_g decreases but the front edge itself is unchanged, the total weight is unchanged since the front weight is the same, while the bottleneck c_{g_0} may decrease. We apply a *lazy capacity update* that defers bottleneck evaluation to selection time and restricts updates to the currently selected path in the iteration (Lemma 3.4).

Lemma 3.4 (On-Demand Bottleneck Update). *Let τ be the selection time of path π_i. Define $\widehat{C}(\tau) = \min_{g \in \pi_i} c_g(\tau)$, where $c_g(\tau)$ is the residual capacity at the head of front(E_g) at time τ. Then $\widehat{C}(\tau)$ equals the true bottleneck capacity for executing π_i at time τ. Therefore, maintaining a cached C_i across selections is unnecessary.*

Proof. At time τ, each edge $g \in \pi_i$ has residual capacity $c_g(\tau)$. By definition, $\widehat{C}(\tau) = \min_{g \in \pi_i} c_g(\tau)$.

Assume, for contradiction, that $\widehat{C}(\tau)$ is not the true bottleneck at time τ. Then one of the following must hold:

(1) There exists $g^\star \in \pi_i$ with $c_{g^\star}(\tau) < \widehat{C}(\tau)$. Then some edge has capacity smaller than the alleged bottleneck, contradicting that $\widehat{C}(\tau)$ is the minimum along π_i.
(2) For all $g \in \pi_i$, $c_g(\tau) > \widehat{C}(\tau)$. Then every edge exceeds $\widehat{C}(\tau)$, so the path minimum would be larger than $\widehat{C}(\tau)$, again contradicting its definition.

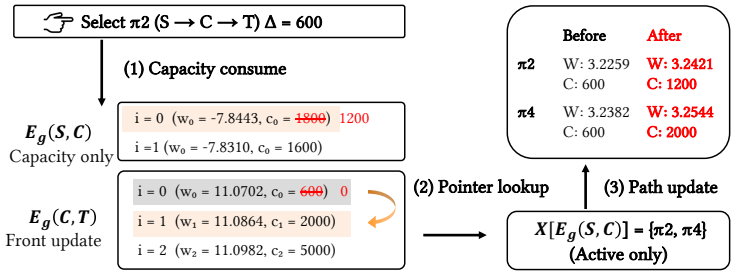


Fig. 5. Example of pointer update.

Both cases lead to a contradiction. Hence, $\widehat{C}(\tau)$ must be the correct bottleneck capacity for executing π_i at time τ. □

Example 3.5. Figure 5 illustrates how pointers are updated during execution. Consider the edge group $E_g(S,C)$ with three price levels ordered by effective weight:

$$\begin{aligned} E_g(S,C) &= \big[(w_0,c_0),(w_1,c_1),(w_2,c_2)\big] \\ &= \big[(11.07,0.6k),(11.09,2k),(11.10,5k)\big]. \end{aligned}$$

(1) When path π_2 is executed with $\Delta = 600$, the front tier of $E_g(S,C)$ is exhausted, triggering a pointer update to (w_1,c_1). In contrast, since $\Delta < c_0$ for $E_g(C,S)$, its front tier is only partially consumed and p_g remains unchanged. (2) After the pointer update, the posting list $X[g] = [\pi_2,\pi_4]$ is retrieved to identify the affected paths. (3) Only the paths in the active bucket are then re-keyed under the new front price level.

Definition 3.6 (Bucket Rebuilt Epoch). An *epoch* begins when the active bucket is constructed or rebuilt from the reserve bucket and ends once the active bucket becomes empty. Multiple *iterations* can occur within a single *epoch*.

Lemma 3.7 (No rebuild unless the active bucket is empty). *If* $|B_{\text{act}}| > 0$ *and there exists* $\pi_i \in B_{\text{act}}$ *with* $C_i > 0$, *then rebuilding from* B_{res} *is not needed.*

Proof. Suppose, for contradiction, that a rebuild is triggered while $|B_{\text{act}}| > 0$ and some $\pi_i \in B_{\text{act}}$ has $C_i > 0$. Let τ be the most recent admission threshold. By the maintenance invariants, $W_i \le \tau$ for all $i \in B_{\text{act}}$ and $W_j \ge \tau$ for all $j \in B_{\text{res}}$. Thus a feasible in-bucket candidate satisfies $W_i \le \tau \le W_j$, so no reserve item can dominate it. Rebuilding is therefore unnecessary, leading to a contradiction. □

Bucket Update (Algorithm 3). Given the selected exchange path (i, W_i, C_i), we first identify the impacted path set $\mathcal{A}_{\text{all}} = \textsc{Affected}(\pi_i)$ that shares at least one edge with π_i. We then restrict updates to those impacted paths that are currently in the active bucket (Line 1). For each $j \in \mathcal{A}$, we recompute its score by updating the path weight and capacity, and update its position in B_{act} accordingly: items with $W_j \le \tau$ are removed (Line 5), while the others are reinserted according to their updated weights (Line 7). When the active bucket becomes empty, a new epoch begins by refilling B_{act} from B_{res}, which also refreshes τ (Line 9). At any time, each path resides in exactly one of $\{B_{\text{act}}, B_{\text{res}}\}$. This routine localizes work to the affected set *within the active bucket*, maintains a compact selection structure, and iterates until no available candidates remain.

Algorithm 3: Bucket Update and Rebuild

Input: current $(B_{\text{act}}, B_{\text{res}}, \tau)$; active size K; selected path (i, W_i, C_i)
Output: updated $(B_{\text{act}}, B_{\text{res}}, \tau)$

/* Localized updates within the active bucket */
$\mathcal{A} \leftarrow \textsc{Affected}(\pi_i) \in B_{\text{act}}$ // affected paths in B_{act}
foreach $j \in \mathcal{A}$ **do**
  $W_j \leftarrow \textsc{UpdateWeight}(j)$; $C_j \leftarrow \textsc{UpdateCapacity}(j)$;
  if $W_j \le \tau$ **then**
    $\textsc{RemoveFrom}\, B_{\text{act}},\ (j, W_j, C_j)$
  else
    $\textsc{InsertOrAdjust}\, B_{\text{act}},\ (j, W_j, C_j)$

/* New epoch: rebuild active bucket if empty */
if $|B_{\text{act}}| = 0$ **then**
  $(B_{\text{act}}, \tau) \leftarrow \textsc{TopS}(B_{\text{res}}, S)$

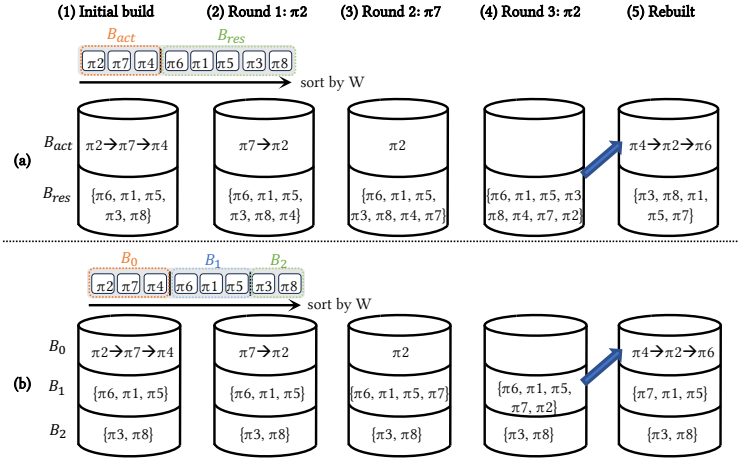


Fig. 6. Example of bucket update and rebuild: (a) Active and reserve buckets, (b) Multi buckets.

Example 3.8. Figure 6 (a) illustrates the two-bucket design.
Initial build. Paths are sorted by increasing W. With $K = 3$, the top three paths π_2, π_7, π_4 form the active bucket $B_{\text{act}} = \{\pi_2, \pi_7, \pi_4\}$, while the rest enter the reserve bucket $B_{\text{res}} = \{\pi_6, \pi_8, \pi_5, \pi_3, \pi_1\}$.
Bucket updates. After executing π_2, only paths sharing the updated edge group (π_2, π_4) are reweighted, while others remain unchanged. Here, π_4 moves to B_{res} after its weight increases, whereas π_2 is reinserted into B_{act}. Subsequent rounds update only a small subset of paths.
Rebuild. When B_{act} becomes empty, it is rebuilt by promoting the best K paths from B_{res}. This localized update scheme avoids global rescans and keeps each iteration lightweight.

Theorem 3.9 (Amortized Complexity). *Consider the iterative path selection algorithm on $|\mathcal{P}|$ paths with active bucket size K for T selection iterations. The amortized time per selection iteration is* $\mathbb{E}[\textit{per-iteration}] = O\left(\frac{|\mathcal{P}|}{T}\right) + O\left(\left(1 + \frac{K}{T}\right)\log K\right)$.

Proof sketch. We prove by aggregate amortized analysis and decompose a bucket-rebuild epoch into three parts.
(1) Bucket assignment and thresholding $O(|\mathcal{P}|)$. We scan all $|\mathcal{P}|$ paths once to compute their current weights and assign each path to a reserve-bucket range. The top K paths are then selected using a KLL-style quantile sketch [33], which returns the admission threshold τ and emits the top K items. At most K paths are inserted into the active bucket, and then these K paths are inevitably consumed within the same epoch, either by being extracted as the current minimum or by being evicted after weight updates.
(2) Building the active bucket $O(K \log K)$. We insert up to K paths into the active bucket, implemented as a binary-heap priority queue keyed by path weight. This costs $O(K \log K)$.
(3) Per-iteration maintenance $O(T \log K)$. Across the T selections in the epoch, each step performs an EXTRACT-MIN and triggers a constant number of localized heap updates and occasional INSERT operations on affected candidates. With a binary heap, each operation costs $O(\log K)$; thus the $O(1)$ operations per selection accumulate to $O(T \log K)$ time over the epoch.
Summing the three parts yields $\text{Time}(|\mathcal{P}|, T, K) = O(|\mathcal{P}|) + O(K \log K) + O(T \log K)$. Dividing by T gives the claimed amortized bound $\mathbb{E}[\text{per-iteration}] = O\left(\frac{|\mathcal{P}|}{T}\right) + O\left(\left(1 + \frac{K}{T}\right) \log K\right)$. □

Discussion. The amortized bound in Theorem 3.9 shows that enlarging the active bucket size K typically increases the number of selections T that can be drained before the next rebuild, thereby shrinking the scan term $O(\frac{|\mathcal{P}|}{T})$. However, this improvement is rarely linear in practice: as K grows, more marginal candidates enter the bucket and are often reprioritized or evicted by reweighting before extraction, so T tends to increase with K. Consequently, while the scan term decreases, the heap-maintenance term $O((1 + \frac{K}{T}) \log K)$ does not necessarily drop and can even grow if T fails to keep pace—because both the logarithmic factor and the multiplier $(1 + \frac{K}{T})$ are affected. We explore further in Section 4.2.

4 Query Optimizations

4.1 Prioritized Bucket Construction

While Algorithm 3 correctly updates the buckets, every bucket rebuilt epoch requires re-evaluating *all* paths to construct new active bucket and reserve bucket. When the path universe $|\mathcal{P}|$ is large, this induces a full scan per rebuild with worst-case cost. However, many computations are redundant since most paths remain far from the frontier and will hardly be selected. To alleviate this inefficiency, we propose a prioritized bucket construction method that partitions paths into coarse weight buckets.

Definition 4.1 (Prioritized Buckets). Let $\mathcal{B} = (B_0, B_1, \ldots, B_{L-1})$ denote an *ordered sequence* of buckets, where $L \in \mathbb{N}$ is the number of levels. Bucket indices encode *execution priority*: buckets are accessed strictly in increasing index order (lower indices are considered earlier). By construction, B_0 coincides with the active bucket B_{act}, while B_ℓ for $\ell \geq 1$ are reserve buckets activated on demand when higher-priority buckets are exhausted.

Data Structure for Prioritized Bucket. To generalize the reserve bucket B_{res} into a prioritized structure, we organize all paths into a multi-level buckets ordered by path weight. Specifically, each bucket layer corresponds to a progressively coarser weight interval, enabling the algorithm to localize updates within a small active subset while retaining global coverage over all candidates. In this way, selection correctness via a global ordering across buckets and enable provably safe early stopping during rebuilds to avoid redundant updates (Lemmas 4.2 and 4.3).

Algorithm 4: Multi-Bucket Update and Rebuild

Input: An ordered sequence $\mathcal{B} = (B_0, B_1, \ldots, B_{L-1})$, Active budget K, threshold τ, Selected path π_i with (W_i, C_i)
Output: Updated $\mathcal{B}$ and refreshed active heap B_0

```
/* Localized updates within the active bucket                                    */
1  A ← AFFECTED(π_i) ∈ B_0
2  foreach j ∈ A do
3      W_j ← UPDATEWEIGHT(j);
4      C_j ← UPDATECAPACITY(j);
5      if W_j ≤ τ then
6          ADJUSTINHEAP(B_0, (j, W_j, C_j))
7      else
8          MOVETOBUCKET(B_0, B, j)

/* Refill when the active heap is empty                                          */
9  if |B_0| = 0 then
10     b* ← LOCATEBUCKETS(B, K)    // Locate the nearest level containing unaffected valid paths
11     B_0 ← ACTIVATEUNTIL(B_0, {B_0, ..., B_b*}, K)
12     τ ← UPDATETHRESHOLD(B_0)
```

LEMMA 4.2 (NO-CROSS-BUCKET DOMINATION). *Given the prioritized buckets in Definition 4.1, for any two indices $i < i'$, no path in $B_{i'}$ has a strictly smaller ordering key than a path in B_i. Equivalently, if $\pi_p \in B_i$ and $\pi_q \in B_{i'}$ with $i < i'$, then $W(\pi_p) \leq W(\pi_q)$.*

PROOF. Suppose, for contradiction, there exist $i < i'$, a path $\pi_p \in B_i$, and a path $\pi_q \in B_{i'}$ such that $\mathrm{W}(\pi_q) < \mathrm{W}(\pi_p)$. By construction, buckets are formed from a global ordering of the paths by $\mathrm{W}(\cdot)$, and are assigned in non-overlapping, index-ordered fashion so that all members of B_i precede all members of $B_{i'}$ in that ordering. The strict inequality $\mathrm{W}(\pi_q) < \mathrm{W}(\pi_p)$ implies that π_q should appear before π_p in the global order, hence π_q must be placed in a bucket with index at most i, contradicting $\pi_q \in B_{i'}$ with $i' > i$. Therefore, our assumption is false and the claim follows. □

LEMMA 4.3 (SAFE REBUILD TRUNCATION). *During a rebuild, let B_i denote a bucket with threshold τ_i defined at the previous epoch. If B_i contains at least one path whose weight has not changed since the last rebuild, then the new global minimum weight is no greater than τ_i. Hence, the rebuild can be safely truncated to the prefix up to B_i.*

PROOF. Assume, for contradiction, that the new global minimum weight $w^\star$ exceeds τ_i. At the previous rebuild, all paths in B_i satisfied $W(\pi) \leq \tau_i$. Since at least one of them remains unaffected, its current weight is still $\leq \tau_i$, which contradicts the assumption that all paths now have $W(\pi) > \tau_i$. Therefore, the new global minimum must satisfy $w^\star \leq \tau_i$, and truncating the rebuild before B_i does not miss the optimal path. □

Multi-Bucket Update (Algorithm 4). We generalize the two-bucket layout into an ordered sequence $\mathcal{B} = (B_0, \ldots, B_{L-1})$ ordered by priority, where a smaller index denotes a higher activation level. Upon executing a path (i, W_i, C_i), we first identify its affected neighbors $\mathcal{A}$ (Line 1) and update each path's score. If a path's updated weight remains below the current threshold, it is adjusted in place within the active heap B_0 (Line 6); otherwise, it is demoted to an appropriate reserve bucket (Line 8). When the active heap becomes empty (Line 9), a new refill epoch begins: LOCATEBUCKETS identifies the nearest level that still contains unaffected valid paths as the new starting point, then sequentially merges subsequent buckets until the cumulative size reaches the active budget K (Line 11). The threshold τ is refreshed based on the boundary weight of the newly

admitted paths (Line 12). Together, these steps localize updates within B_0 while performing refills, maintaining a compact active set and minimizing rebuild cost across epochs.

Example 4.4. Figure 6 (b) illustrates the multi-bucket design.
Initial build. After a one-time global sort, paths are partitioned into ranked buckets: the top-K into B_0 (Active), the next-K into B_1, and the remainder into B_2. With $K = 3$, $B_0 = \{\pi_2, \pi_7, \pi_4\}$, $B_1 = \{\pi_6, \pi_1, \pi_5\}$, $B_2 = \{\pi_3, \pi_8\}$.
Localized updates. After executing π_2, only paths sharing the updated edge group (π_2, π_4) are re-keyed. For example, $\pi_4 : (3.2382, 300) \rightarrow (3.2545, 400)$. If an updated path falls below the top-K threshold, it is demoted from B_0 to B_1.
Rebuild. When B_0 becomes empty, it is rebuilt from B_1; only when B_1 is exhausted do we draw from B_2. This layered refill strategy confines most movements to upper buckets and keeps low-ranked candidates cold.

Amortized Time Complexity. Compared to Theorem 3.9, prioritized buckets reduce the per-rebuild $O(|\mathcal{P}|)$ rescans to $O\big(|\mathcal{A}| + P_{\text{aff}} + L\big)$, where the L reserve-bucket levels $\mathcal{B} = \{B_0, \ldots, B_{L-1}\}$ partition $\mathcal{P}$ by weight ranges, $\mathcal{B}_{\text{aff}} \subseteq \mathcal{B}$ is the set of levels whose ranges are crossed by the admission threshold within the epoch, $P_{\text{aff}} = \sum_{B \in \mathcal{B}_{\text{aff}}} |B|$ is the number of candidates inside these affected levels, and $\mathcal{A}$ is the set of candidates whose weights actually change. Consequently, the amortized per-iteration time becomes $\mathbb{E}[\text{per-iteration}] = O\Big(\frac{|\mathcal{A}|+P_{\text{aff}}+L}{T}\Big) + O\big(\big(1 + \frac{K}{T}\big) \log K\big)$.

4.2 Adaptive Active Bucket Sizing

While the active bucket localizes maintenance to affected candidates, choosing its size K is nontrivial. Empirically, a small K under-fills the heap and triggers frequent rebuilds, making the scanning and admission cost dominant; conversely, an overly large K inflates heap operations so the per-iteration cost is dominated by $\log K$, yielding diminishing returns. This trade-off is nonlinear (cf. Section 3.2): discrete reweighting, churn, and evictions limit how many selections are drained per rebuild, so increasing K does not yield a proportionally smaller number of rebuilds. In order to navigate this trade-off, we adopt an *adaptive* scheme that decides K online during IOPE queries.

Lemma 4.5 (Independence of K from Query Results). *For a fixed set of candidate paths $\mathcal{P}$ and ordering key $W(\pi)$, the size K of the active bucket affects only the runtime cost of maintenance, not the correctness of the resulting* IOPE *query. That is, the sequence of globally minimal paths extracted from $\mathcal{P}$ is invariant to K.*

Proof. The ranking of candidate paths is determined solely by their ordering key $W(\pi)$, which is independent of K. Changing K alters only how many candidates are maintained per epoch, not their relative order. Each rebuild re-computes all relevant keys before the next iteration, ensuring that the same minimal path will be extracted regardless of K. Hence K influences only computational efficiency, not the final or intermediate optimal path results. □

Therefore, K is purely an efficiency parameter: optimizing it improves throughput without altering the semantics of the results. This observation motivates an online adjustment of K based on runtime feedback. We formalize this feedback through *bucket utilization* as follows.

Definition 4.6 (Bucket Utilization). Let T be the number of selection iterations completed during an epoch with active bucket size K. The *bucket utilization* is defined as $\alpha = \frac{T}{K} > 0$.

In general, a smaller α indicates under-utilization—many candidates churn before extraction—suggesting that K may be too large; a larger α indicates near-full consumption before rebuild, suggesting that

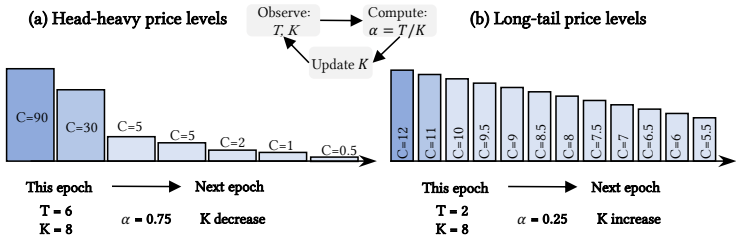


Fig. 7. Adaptive K update.

K may be too small. Notably, α can exceed one, since a path may be reinserted into the bucket according to Algorithm 3.

Lemma 4.7 (Optimal K per Epoch). *Plugging $T = \alpha K$ into Theorem 3.9 yields*

$$\mathbb{E}[\textit{per-iteration}] \;=\; O\!\left(\frac{|\mathcal{P}|}{\alpha K}\right) \;+\; O\!\left(\left(1+\tfrac{1}{\alpha}\right)\log K\right). \tag{1}$$

Consequently, the optimal bucket size scales as $K^\star \;\propto\; \frac{|\mathcal{P}|}{1+\alpha}$.

Proof. The right-hand side of Eq. (1) is quasi-convex in K, so an optimal $K^\star$ exists. Ignoring constants and lower-order terms yields the proportional relationship $K^\star \propto \frac{|\mathcal{P}|}{1+\alpha}$. Due to space constraints, the full proof appears in Appendix [36]. □

Alpha-aware Adaptive K Update. We adaptively update the active-bucket size K at epoch boundaries using feedback from the preceding epoch. Specifically, at the end of epoch t, we record the selections consumed T_t and the candidate count $|\mathcal{P}_t|$, and estimate utilization $\alpha_t = T_t/K_t$ (note $\alpha_t \geq 0$ and can exceed 1 with reinsertion). Guided by Lemma 4.7 and Eq. (1), we interpret α_t as a dominance signal: when α_t is high, the rebuild term $O(|\mathcal{P}|/(\alpha K))$ is already well amortized and the heap term $\Theta(\log K)$ dominates, so we decrease K in the next epoch; when α_t is low, churn inflates the rebuild share, so we increase K to amortize scans over more selections. This controller is purely efficiency-driven by Lemma 4.5, adjusting K does not alter the sequence of globally minimal paths—and K remains fixed within each epoch.

Example 4.8. Figure 7 illustrates adaptive-K under two representative liquidity regimes using the same initialization $K_1 = 8$ and feedback $\alpha_t = T_t/K_t$ (where T_t is the number of selection iterations completed in epoch t). **(a) Highly skewed liquidity $\Rightarrow$ $K \downarrow$.** The active bucket is quickly consumed after $T_1 = 6$ iterations, so $\alpha_1 = 6/8 = 0.75$ and the controller shrinks to $K_2 = 4$. **(b) Long-tail liquidity $\Rightarrow$ $K \uparrow$.** The bucket yields fewer completed iterations $T_1 = 2$, so $\alpha_1 = 2/8 = 0.25$ and the controller expands to $K_2 = 16$. In this way, adaptive-K balances exploration depth against bucket-maintenance cost by shrinking K when utilization is high and expanding K when utilization is low.

5 Experiments

5.1 Experimental Setup

All experiments were conducted on a Linux machine equipped with an AMD Ryzen Threadripper PRO 5995WX CPU, 503 GiB RAM, and g++ 11.4.0 (-O3 optimization).

Table 3. Statistics of experimental datasets.

Dataset	Num. Nodes	Num. Multi Edges	Max Node Degree	Density
U1	14.0K	297.2K	17,457	1.2E-04
U2	19.9K	1.0M	25,764	8.3E-05
U3	33.8K	1.4M	42,309	4.4E-05
U4	58.7K	1.7M	57,055	3.0E-05
U5	107.1K	3.2M	129,026	1.2E-05

Datasets. We conduct experiments on five token exchange graphs constructed from Uniswap[4] ranging from block 13,000,000 to block 20,500,000. Each dataset corresponds to a *distinct historical on-chain snapshot* (i.e., the five graphs are separated by time), collected from Uniswap states in August 2021, April 2022, November 2022, April 2023, and August 2024. For each snapshot block, we extract all available swap pools, including their fee tiers and price levels, and construct the exchange graph using a unified processing pipeline. Specifically, tokens are modeled as vertices, while each feasible swap option contributes a directed edge associated with both a weight, representing the exchange rate, and a capacity, derived from the pool state observed at that block. These snapshots are chosen to cover diverse liquidity and volatility conditions on Uniswap over time. In these graphs, vertices represent crypto tokens, and a directed edge (u, v) exists if a swap from token u to token v is available at the corresponding snapshot block. Edge weights are defined as $w(u, v) = -\log r(u \to v)$, where r is the swap rate computed from the pool reserves at that block. For a given token pair (u, v), there can be *multiple directed parallel edges*, each corresponding to a distinct price level and carrying its own capacity. Table 3 summarizes the statistics of these token exchange graphs.

Baselines. We compare our method against two groups of baselines: (1) shortest-path algorithms that handle negative weights or negative cycles; and (2) arbitrage detection algorithms in crypto markets, which are originally designed for cycle discovery and are adapted here to path queries.

- EMUF (2025) [35] uses the Shortest Path Faster Algorithm (SPFA) to find maximum-utility source–sink paths under encrypted edge utilities. It implicitly enumerates paths via relaxations, with hop constraints and support for negative weights.
- DeFi-ARB (2021) [60] applies Bellman–Ford–style relaxation to detect negative cycles in token exchange graphs. We adapt it to path queries by running bounded relaxations from the source and extracting shortest paths.
- GoldPhish (2023) [40] greedily searches profitable 3-hop arbitrage cycles. We generalize it to larger hop limits and use it as a lightweight path enumerator.
- BriDe (2024) [54] detects arbitrage opportunities using a DFS-style exploration with optimized trade ordering. We adapt its detector to enumerate candidate paths with negative edges and select the shortest routes.

Queries. For each graph G, we construct an identical query set of 1,028 source–target pairs corresponding to commonly used exchange pairs, each either originating from Wrapped Ethereum (WETH) or targeting at WETH. WETH is widely selected because it is a highly liquid base asset on Ethereum, serving as a central hub for most trading activities and forming direct pairs with the majority of ERC20 tokens. We limit paths to a maximum of 5 hops. Within this bound, the number of feasible paths for a query ranges from 94K to 230K. In practice, this length constraint implicitly controls on-chain gas costs by avoiding longer, gas-inefficient routes.

[4]https://app.uniswap.org. Accessed on Sep 8, 2025.

Table 4. Comparison of average detection time per iteration (in microseconds) across pairs. "Speedup" shows the speedup of ORDER v.s. the second-best algorithm (underlined).

Dataset	ORDER	EMUF	DeFi-ARB	GoldPhish	BriDe	**Speedup vs. 2nd**
U1	3.8	14,040.4	432.6	911.9	462.9	**113.8**× ⇑
U2	10.2	21,251.4	511.6	1287.0	524.3	**50.2**× ⇑
U3	12.4	25,262.8	602.7	1489.2	655.9	**48.6**× ⇑
U4	15.4	24,973.8	625.9	1569.2	725.8	**40.6**× ⇑
U5	30.3	34,101.6	765.6	1795.0	917.5	**25.3**× ⇑

Table 5. Speedup of ORDER over the best baseline on non-WETH queries, grouped by endpoint degree (High/Low). Values are computed from Tables 8–11 in the Appendix [36].

Dataset	High–High	High–Low	Low–High	Low–Low
U1	256.00×	364.81×	546.41×	637.21×
U2	166.52×	195.24×	385.18×	599.46×
U3	143.85×	146.21×	388.83×	364.02×
U4	140.87×	147.63×	422.07×	286.82×
U5	124.11×	128.42×	268.09×	293.89×

Metrics. We evaluate the router on two complementary dimensions that jointly capture *efficiency* and *effectiveness* under real-time constraints. (1) *Iteration-time* (ms/iter): wall-clock time per algorithmic iteration. This normalizes across pairs with different inherent iteration counts and isolates algorithmic efficiency; it is fast to compute and aligns with our latency target. This time measures the entire IOPE pipeline in Algorithm 1: initializing buckets for the given pre-enumerated paths, iteratively selecting the best path, applying allocation updates, and updating affected bucket entries, until the requested input amount is fully routed or no feasible paths remain. In other words, the timing covers all operations between query input and returning the final routing result. (2) *Throughput under a time budget*: the total tradable amount executed within a fixed wall-clock budget T. This metric reflects end-to-end effectiveness—how much liquidity the method can realize within the latency bound. Formally, $\text{Throughput}_{\text{ratio}}(T) = \frac{Q_{\text{exec}}(T)}{Q_{\text{total}}} \times 100\%$. Unless otherwise specified, all metrics are averaged over queries.

Default Settings. We report results with Q=100 and L=100 by default, where Q is the initial tradable quantity (in source-token units) and L is the number of prioritized buckets. To clarify, Q denotes the total swap amount in a routing query, reflecting a realistic trader's order size. In practice, such orders are typically much larger than single-tick capacities; for example, in U1, over 90% of edge capacities are below 3, with a median of 0.0012. This motivates evaluating Q in the [1, 100] range. Each query proceeds iteratively until either Q=0 (no remaining quantity) or no feasible path remains under current residual capacities and fees. We cap the per-query wall-clock runtime at 500 ms; if the cap is reached, throughput is computed from the cumulative amount executed up to the deadline (partial execution counts toward the total). For efficiency and to limit slippage compounding, candidate routes are restricted to paths of at most five hops. Unless otherwise stated, we report averages over all pairs.

5.2 Efficiency

Overall efficiency (Table 4). We compare the detection time of different algorithms across the datasets. Specifically, ORDER outperforms all baselines: EMUF (speedup: 1,125.5×–3,794.7×),

GoldPhish (speedup: 59.2×–246.5×), BriDe (speedup: 30.3×–125.1×), and DeFi-ARB (speedup: 25.3×–113.8×). The efficiency of ORDER arises from its prioritized bucket path index and localized, incremental updates that refresh only the affected candidates, avoid global rescans, and keep a small active set. In contrast, baseline methods incur a rapid growth in detection time due to path-space explosion. On token exchange graphs, the number of feasible paths grows sharply with hop bounds. For example, in U1, the candidate paths grow from ~358 (3 hops) to 93,989 (5 hops). Baseline methods repeatedly re-enumerate or re-rank this enlarged candidate space, causing combinatorial growth in detection time and substantially higher runtime.

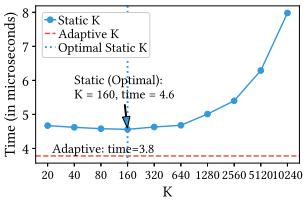
Non-WETH query diversity. Excluding WETH-related pairs, we additionally evaluate on non-WETH source–target queries to assess generality beyond hub-centric workloads. For each dataset, we rank all non-WETH tokens by (total) degree and select two endpoint sets: *high-degree* tokens from the top 0.1% of the degree distribution and *low-degree* tokens from the bottom tail.[5] We then form four query types by pairing endpoints from these sets: High–High, High–Low, Low–High, and Low–Low, and use the same query sets for all methods for fair comparison. Table 5 summarizes the speedup of ORDER over the best baseline across these four types. We observe two consistent trends: (i) all methods run faster as endpoint degrees decrease (due to smaller candidate path sets), and (ii) ORDER's advantage becomes more pronounced on low-degree queries, where localized maintenance avoids unnecessary global recomputation. Full per-method runtime results are reported in Appendix for completeness and detailed comparison [36].

Impact of Bucket Size K (Figure 9). We evaluate how different active bucket size K affects efficiency across datasets. As shown, when K increases, the average time per iteration first decreases, reaches an optimum, and then rises again. Across datasets, the speed gap between the best and worst-performing K settings ranges from **1.4**× to **2.2**× times. This unimodal trend reflects a fundamental trade-off: overly large K incurs heavy in-bucket maintenance including reinserting and localized updates, whereas overly small K triggers frequent bucket rebuilds. An intermediate K balances these costs and yields the best performance. However, the optimal K differs across datasets due to variations in path diversity, score dispersion, and update locality. For example, K=160 on dataset U1, K=640 on dataset U2. Further, we conduct a detailed quantitative and sensitivity analysis to study the impact of the adaptive-K controller in Appendix [36]. On U1, adaptive-K reduces the mean reranking latency on the same set of queries. This gain is driven by proactive bucket resizing, which lowers rebuild events from 13.71 to 4.81 per query (64.9% fewer) compared to fixed-K. Moreover, the benefits are stronger for queries with a large number of candidate paths, but become less pronounced on larger graphs.

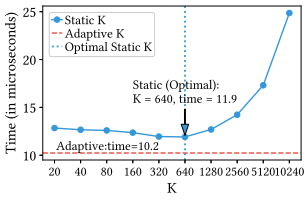
Therefore, we compare fixed-K with our adaptive-K scheme that adjusts K online based on utilization and iteration dynamics. Empirically, adaptive-K matches or surpasses the best fixed setting while eliminating manual tuning. In practice, this brings two advantages. First, no hyperparameter search over K is required. Second, the method automatically balances rebuild frequency and in-bucket maintenance in real time, yielding 10.0%–17.1% lower runtime than the best fixed K across datasets.

Impact of Prioritized Bucket (Figure 10). We assess the effect of introducing prioritized buckets by comparing the standard two-bucket design (B_{act}, B_{res}) with our multi-level variant. As shown, disabling the prioritized buckets slows ORDER by 28.7%–49.6% across datasets. The hierarchy improves efficiency by eliminating redundant rescans and confining maintenance to the activated prefix of buckets: only the lowest levels are refreshed, while higher levels remain dormant until needed. Moreover, as K grows, rebuild frequency drops markedly, yielding smoother per-round

[5]Across datasets, high-degree endpoints have average degrees of 48.46–68.62 (above the 99.9% percentile), while low-degree endpoints average 5.32–5.80.



(a) U1.



(b) U2.

Fig. 9. Detection time of various static K and adaptive K.

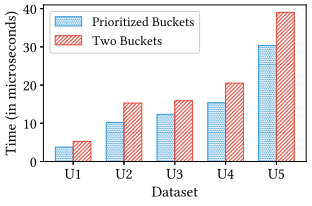


Fig. 10. Prio./Two buckets.

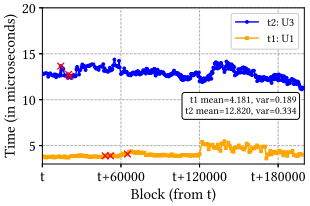


Fig. 11. Long term.

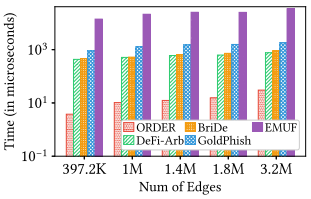


Fig. 12. Overall scalability.

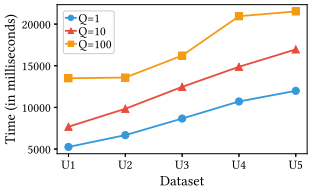


Fig. 13. Scalability v.s. Q.

latency and more stable throughput. Overall, the prioritized organization preserves accuracy while accelerating execution, especially in long-running queries where path weights evolve gradually.

Scalability (Figures 12 and 13). We assess the scalability of ORDER across graphs with $|E|$ ranging from 297.2K to 3.2M. As shown, ORDER maintains a consistent lead as graph size grows, with the gap to the second-best approach widening on larger inputs. On the smallest graph, ORDER reduces latency by about 428.8 ms compared to the second-best method, while on the largest graph, it remains faster by about 735.3 ms. These results indicate that ORDER sustains high throughput under increasing structural complexity.

Furthermore, we examine how scalability varies with the initial quantity Q. We evaluate $Q \in \{1, 10, 100\}$ under identical settings. Across datasets, total runtime grows slowly with Q: increasing Q by 10× raises runtime by only 1.36×–1.52× (from 1 to 10) and 1.16×–1.41× (from 10 to 100); over two orders of magnitude (from 1 to 100), the increase is only 1.81×–2.02×. Meanwhile, per-round averages remain nearly constant on most datasets, especially the larger datasets varying by at most 5.4% across Q. Taken together, these results show that ORDER exhibits sublinear growth in total runtime as Q increases while maintaining per-round efficiency.

Moreover, we evaluate ORDER under different market conditions using long-term continuous experiments starting from bull-market (U1) and bear-market (U3) snapshots over 200,000 blocks (Figure 11). Across both periods, ORDER exhibits stable runtime behavior, with mean execution times of 4.18 μs and 12.82 μs and small variances, indicating no long-term drift or degradation. These results demonstrate that ORDER maintains stable and robust performance across diverse market environments. More details are shown in Appendix [36].

5.3 Effectiveness

Throughput under a Time Budget (Table 6). We report *throughput* as the total quantity executed within a strict 500 ms latency bound, together with a normalized ratio showing how much of the full input amount can be processed under real-time constraints. As shown in Table 6, we compare ORDER with all baselines across multiple datasets under identical settings. First, ORDER

Table 6. Throughput under a 500 ms time budget (amount of input that can be executed), and two ratios: Ratio_t (fraction of the input executed) and Ratio_o (expected output ratio), across datasets.

Dataset	ORDER		EMUF		DeFi-ARB		GoldPhish		BriDe		Monte Carlo	
	Throughput	$Ratio_t$	Throughput	$Ratio_t$	Throughput	$Ratio_t$	Throughput	$Ratio_t$	Throughput	$Ratio_t$	Throughput	$Ratio_o$
U1	53.9	100.0%	12.0	25.1%	5.7	10.3%	0.2	2.1%	5.5	10.1%	53.9	87.4%
U2	50.3	100.0%	10.33	24.8%	4.3	9.5%	0.1	1.1%	4.1	9.1%	50.3	89.2%
U3	47.1	100.0%	8.18	22.2%	3.7	8.2%	<0.1	1.0%	2.6	6.3%	47.1	73.8%
U4	46.0	100.0%	6.2	17.6%	2.9	6.6%	<0.1	1.5%	2.1	4.9%	46.0	22.6%
U5	38.4	100.0%	5.9	20.1%	3.2	9.7%	0.1	2.1%	1.6	5.8%	38.4	85.6%

consistently maintains 100% throughput across all datasets, achieving the highest executed quantity under tight latency limits. This demonstrates its ability to fully utilize feasible paths and sustain stable performance regardless of graph scale. Second, EMUF, representing classical shortest-path detection, performs moderately well but exhibits sensitivity to graph density and path structure, leading to reduced throughput on larger graphs. Third, enumeration-based methods such as DeFi-ARB, GoldPhish, and BriDe suffer substantial drops as graph size increases. Their throughput drops below 10% on larger datasets, indicating poor scalability and high overhead.

Evaluation against Monte Carlo (Table 6). We compare ORDER with a Monte Carlo (MC) selector [32], which repeatedly samples feasible paths and performs the same execution and residual-capacity updates after each allocation. We report the *throughput ratio*, $\text{Output}_{\text{MC}}/\text{Output}_{\text{ORDER}}$, where output is the *executed output value*. As shown, MC achieves only 22.6%–89.2% of ORDER's value across datasets, suggesting that random exploration struggles to allocate budget to the best feasible paths under tight latency constraints. Additional results comparing ORDER with CFMM [7], a convex-program-based routing method, are reported in the Appendix [36], showing that CFMM provides only a marginal advantage despite being substantially slower.

Evaluation against Commercial Practices. We compare ORDER with a leading commercial DEX aggregator, 0x [50], which employs a proprietary routing algorithm that is not publicly disclosed. Because commercial routing services such as 0x do not provide an interface for querying routing decisions on historical block states, they cannot participate in our replay-based baseline experiments that require all methods to be evaluated on the same snapshot. Instead, we query the live 0x API at the current block (block 21,974,830, March 2025) using Uniswap V2 and V3 liquidity pools, and report its returned routes as an online reference point.

Our evaluation focuses on three representative trading pairs, each tested under three input amounts to reflect small, medium, and large demand levels. For each setting, we record the achieved output token quantity and compare ORDER against 0x to assess routing quality under varying trade sizes relative to a widely used black-box aggregator. These pairs capture distinct conversion semantics in decentralized trading:

- **rETH ↔ wstETH:** A swap between staking derivatives of the same underlying asset (ETH), highlighting precision routing and low-slippage behavior within the liquid-staking ecosystem.
- **USDT ↔ cbBTC:** A conversion between a fiat-pegged stablecoin and a BTC-wrapped custodial asset, representing capital inflows or outflows between stable and volatile stores of value.
- **WBTC ↔ USDT:** A high-liquidity trade between a BTC wrapper and a stablecoin, stressing execution performance and slippage control under large transaction sizes.

Results are summarized in Table 7. Across the tested token pairs and input scales, ORDER achieves routing outputs that are generally competitive with those returned by the commercial aggregator 0x, and in several cases exceeds them. For rETH↔wstETH and USDT↔cbBTC, the two systems produce identical outputs at smaller scales, while ORDER yields a higher output for

the largest rETH↔wstETH trade. For WBTC↔USDT, 0x performs slightly better at input amount 20, whereas ORDER outperforms 0x at input amounts 100 and 600. Notably, at the largest tested trade sizes, ORDER consistently matches or improves upon 0x, with gains of up to about 1.3% in our evaluation. Overall, these results indicate that ORDER can effectively exploit fragmented liquidity and compose high-quality routes under heavy demand and attains commercial-grade routing quality while operating under a transparent, snapshot-based execution model.

Table 7. Comparison with commercial routing services on block 21,974,830. Each pair is tested under three input amounts. The "Input Amount" and "Output Amount" columns represent token quantities for each pair. USD values are approximate: ETH ≈ $1.8k, BTC ≈ $80k.

Pairs	Router	Input Amt.	Input Value (USD)	Output Amt.
rETH ↔ wstETH	0x	20.0	~ $36k	18.8
		100.0	~ $180k	93.9
		500.0	~ $900k	221.1
	ORDER	20.0	~ $36k	18.8
		100.0	~ $180k	93.9
		500.0	~ $900k	224.0
USDT ↔ cbBTC	0x	200,000.0	~ $200k	2.4
		1,000,000.0	~ $1.0M	12.0
		5,000,000.0	~ $5.0M	59.7
	ORDER	200,000.0	~ $200k	2.4
		1,000,000.0	~ $1.0M	12.0
		5,000,000.0	~ $5.0M	59.7
WBTC ↔ USDT	0x	20.0	~ $1.6M	1,651,146.6
		100.0	~ $8.0M	8,133,557.4
		600.0	~ $48.0M	41,143,835.9
	ORDER	20.0	~ $1.6M	1,650,667.6
		100.0	~ $8.0M	8,135,058.3
		600.0	~ $48.0M	41,268,271.9

6 Related Work

Shortest Path Query. Research on shortest path queries includes both static indexing and dynamic algorithms. Static approaches such as Hub Labeling (HL) [4], Arterial Hierarchy (AH) [61], Transit Node Routing (TNR) [10], and 2-Hop Labeling [15] achieve ultra-fast queries on road networks via heavy preprocessing and hierarchical indexes, but incur high construction cost and limited adaptability. Classical shortest path algorithms include Dijkstra's [20], Bellman–Ford [11], SPFA [41, 43], and Floyd–Warshall [24]. To support evolving graphs, dynamic variants have been proposed, including landmark-based estimation [46], dynamic pruned landmark labeling (PLL) [5], fully dynamic acceleration combining bit-parallel SPTs with BFS [29], and methods for complex networks [56]. Despite these advances, two limitations remain: static index construction is costly under frequent updates, and most methods cannot handle negative weights or cycles, limiting their applicability to crypto token graphs.

Crypto Optimal Routing. Optimal routing in cryptocurrency markets has progressed from global convex formulations to more efficient algorithmic designs. Danos et al. [17] introduced unified convex programs for routing and arbitrage, with tractable lower bounds such as independent-path routing. Subsequent work modeled routing across heterogeneous constant-function market makers as a convex program with a unique solution [7], and later scaled it via dual decomposition to reduce per-edge cost [19]. Empirical studies revealed persistent inefficiencies on Uniswap and Sushiswap [12], motivating practical routers, while Danos et al. [18] proposed an on-chain push-and-solve scheme for Uniswap v3-style pools with gas-efficient splits. These approaches assume one-shot global planning, computing a complete allocation for the full input amount. While theoretically appealing, such plans are hard to make converge at DEX scale and are fragile in practice, since partial execution immediately invalidates the rest. In contrast, production routing systems follow an

iterative execution model, repeatedly selecting the currently best feasible route and re-optimizing after each partial fill. Our work adopts this execution model (formalized as IOPE) and focuses on making iterative re-optimization efficient at scale.

Another complementary line of research models crypto markets as graphs and applies graph algorithms for routing and arbitrage. Zhang et al. [57] proposed a line-graph formulation with dynamic-programming-style methods, but evaluated primarily on small instances. Related work on arbitrage detection formulates the problem as finding negative cycles in token exchange graphs [40, 54, 60], using Bellman–Ford-style relaxations, queue-accelerated variants, or greedy traversals. However, these methods often rely on heavy heuristic pruning, operate on static snapshots that require frequent recomputation as prices change, and only partially account for shared and tight capacities across routes. However, these methods typically either recompute from scratch when the effective edge weights change, or rely on coarse heuristic pruning, and thus do not provide an efficient mechanism to localize updates to only the affected paths under discrete tier shifts and shared capacity coupling during iterative execution.

Max Flow Min Cost (MCMF). Classical network flow algorithms, including Ford–Fulkerson [25], Dinic [21], push–relabel [27], and pseudoflow [30], form the basis of maximum and minimum-cost flow computation. Xu et al. [51] propose incremental flow computation strategies to efficiently find in temporal flow networks the optimal temporal flows with maximum average flow values based on their time interval length. However, these methods cannot cost constraints. Length-bounded flows with hop constraints are NP-hard [6, 8, 39] but admit approximation algorithms, and related bounded disjoint path problems further highlight these challenges [13, 14, 28, 45]. However, classical MCMF cannot model token swaps accurately, as it assumes fixed linear costs instead of volume-dependent slippage (Section 2.3).

7 Conclusion and Future Work

We introduced ORDER, an optimal routing framework with a priority bucket index and adaptive control. Experiments on real-world data show up to **114**× speedup over state-of-the-art routers while preserving execution quality and stability.

Future work. Beyond DeFi arbitrage, ORDER is a general solution for optimal routing in graphs, with applications in foreign exchange [31, 34], transportation [9, 52], and payment system [48]. In future work, we plan to extend ORDER in three directions: (i) generalizing to multi-source multi-target routing for portfolio-level and cross-chain optimization, (ii) incorporating stochastic and predictive models to anticipate liquidity and block-time effects, and (iii) co-designing with on-chain settlement mechanisms to further reduce latency and execution risk.

Disclaimer. ORDER is intended for academic research only and should not be used in live trading without rigorous risk, compliance, and security review, as real-world deployment involves substantial hazards such as slippage, MEV, and execution delays.

Acknowledgments

The authors gratefully acknowledge the support of Tokka Labs for funding this research. The work of Shixuan Sun is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62572303. This research/project is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech 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 and Infocomm Media Development Authority.

References

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

[2] 2025. 1inch. https://1inch.io/. Accessed: 2025-10-13.

[3] 1inch Network. 2023. 1inch Network Router v5. https://1inch.io/. Accessed: 2025-08-25.

[4] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2010. *A hub-based labeling algorithm for shortest paths on road networks.* Technical Report MSR-TR-2010-165. Microsoft Research.

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd International World Wide Web Conference (WWW)*. 237–248.

[6] K. Altmanová, P. Kolman, and J. Voborník. 2019. On polynomial-time combinatorial algorithms for maximum l-bounded flow. In *Algorithms and Data Structures (WADS 2019)*. Springer, 14–27.

[7] Guillermo Angeris, Alex Evans, Tarun Chitra, and Stephen Boyd. 2022. Optimal routing for constant function market makers. In *Proceedings of the 23rd ACM Conference on Economics and Computation*. 115–128.

[8] Georg Baier, Thomas Erlebach, Anthony Hall, Ekkehard Köhler, Petr Kolman, Ondrej Pangrác, and Martin Skutella. 2010. Length-bounded cuts and flows. *ACM Transactions on Algorithms (TALG)* 7, 1 (2010), 1–27.

[9] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm engineering: Selected results and surveys*. Springer, 19–80.

[10] Hannah Bast, Stefan Funke, Dennis Matijevic, Peter Sanders, and Dominik Schultes. 2007. In transit to constant time shortest-path queries in road networks. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 46–55.

[11] Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90.

[12] 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.

[13] Andreas Bley and João Neto. 2010. Approximability of 3-and 4-hop bounded disjoint paths problems. In *Integer Programming and Combinatorial Optimization (IPCO 2010)*. Springer, 205–218.

[14] Leizhen Cai and Jie Ye. 2019. Two edge-disjoint paths with length constraints. *Theoretical Computer Science* 795 (2019), 275–284.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[16] Coinsutra. [n.d.]. Best Crypto DEX Aggregators. https://coinsutra.com/best-crypto-dex-aggregator/. Accessed: 2026-02-04.

[17] Vincent Danos, Hamza El Khalloufi, and Julien Prat. 2021. Global order routing on exchange networks. In *International Conference on Financial Cryptography and Data Security*. Springer, 207–226.

[18] Vincent Danos, Leo Murao Watson, Hamza El Khalloufi, and Santiago Valencia. 2024. On-chain optimal aggregation of Uniswap v3 clones. (2024).

[19] Theo Diamandis, Max Resnick, Tarun Chitra, and Guillermo Angeris. 2023. An efficient algorithm for optimal routing through constant function market makers. In *International Conference on Financial Cryptography and Data Security*. Springer, 128–145.

[20] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271.

[21] E. A. Dinic. 1970. Algorithm for solution of a problem of maximum flow in a network with power estimation. In *Soviet Mathematics Doklady*, Vol. 11. 1277–1280.

[22] DODO Research. 2025. Reveal the Secrets of the Aggregator Problem: Analysis and Model Building. https://blog.dodoex.io/dodo-research-reveal-the-secrets-of-the-aggregator-problem-analysis-and-model-building-ba0ead85948c. Accessed: 2026-03-15.

[23] Martin DD Evans. 2002. FX trading and exchange rate dynamics. *The Journal of Finance* 57, 6 (2002), 2405–2447.

[24] Robert W. Floyd. 1962. Algorithm 97: Shortest Path. *Commun. ACM* 5, 6 (1962), 345.

[25] L. R. Ford and D. R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

[26] Thierry Foucault and Albert J Menkveld. 2008. Competition for order flow and smart order routing systems. *The Journal of Finance* 63, 1 (2008), 119–158.

[27] Andrew V. Goldberg and Robert E. Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4 (1988), 921–940.

[28] Bernhard Haeupler, D. Ellis Hershkowitz, and Thatchaphol Saranurak. 2023. Maximum length-constrained flows and disjoint paths: Distributed, deterministic, and fast. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1–13.

[29] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully dynamic shortest-path distance query acceleration on massive networks. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM)*. 1533–1542.

[30] Dorit S. Hochbaum. 2008. The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. *Operations Research* 56, 4 (2008), 992–1009.

[31] Oleg Itskhoki and Dmitry Mukhin. 2023. *Optimal exchange rate policy*. Technical Report. National Bureau of Economic Research.

[32] Malvin H Kalos and Paula A Whitlock. 2008. *Monte carlo methods*. John Wiley & Sons.

[33] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.

[34] Jarosław Kwapień, Sylwia Gworek, Stanisław Drożdż, and Andrzej Górski. 2009. Analysis of a network structure of the foreign currency exchange market. *Journal of economic interaction and coordination* 4, 1 (2009), 55–72.

[35] Zhetao Li, Young-June Choi, Hiroo Sekiya, Qingyong Deng, et al. 2025. Location and Reward Privacy-Preserving based Secure Task Allocation in Mobile Crowdsensing. *IEEE Transactions on Mobile Computing* (2025).

[36] Bingqiao Luo, Yuhang Chen, Yuheng Cong, et al. 2026. *ORDER: Optimal Routing with Path Indexing in Exchange Graph*. Technical Report. Technical Report. https://github.com/Xtra-Computing/ORDER/blob/main/technical_report/ORDER_Technique_Report.pdf Accessed: 2026-03-12.

[37] Bingqiao Luo, Jiaxin Jiang, Yuhang Chen, Junyi Hou, Cheng Jun Tey, Ziyang Qiu, Bingsheng He, Spencer Xiao, Dominic Ong, and Wee Howe Ang. 2025. RICH: Real-time Identification of negative Cycles for High-efficiency Arbitrage. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4081–4089.

[38] 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.

[39] Ali Ridha Mahjoub and S. Thomas McCormick. 2010. Max flow and min cut with bounded-length paths: complexity, algorithms, and approximation. *Mathematical Programming* 124, 1-2 (2010), 271–284.

[40] 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.

[41] Edward F. Moore. 1959. The shortest path through a maze. *Proceedings of an International Symposium on the Theory of Switching* (1959), 285–292.

[42] Odos. 2025. Odos (The Optimal Order Routing Solution). https://www.odos.xyz/.

[43] James B. Orlin. 1993. A faster strongly polynomial minimum cost flow algorithm. *Operations Research* 41, 2 (1993), 338–350.

[44] James B Orlin. 1997. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming* 78, 2 (1997), 109–129.

[45] Binglin Tao, Mingyu Xiao, and jingyang Zhao. 2020. Finding minimum-weight link-disjoint paths with a few common nodes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 938–945.

[46] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM)*. 1785–1794.

[47] Uniswap. 2023. Uniswap V3: Decentralized Exchange and Smart Order Router. https://app.uniswap.org/. Accessed: 2025-08-25.

[48] Sushil Mahavir Varma and Siva Theja Maguluri. 2021. Throughput optimal routing in blockchain-based payment systems. *IEEE Transactions on Control of Network Systems* 8, 4 (2021), 1859–1868.

[49] Dejan Vujičić, Dijana Jagodić, and Siniša Ranđić. 2018. Blockchain technology, bitcoin, and Ethereum: A brief overview. In *2018 17th international symposium infoteh-jahorina (infoteh)*. IEEE, 1–6.

[50] Will Warren and Amir Bandeali. 2017. 0x: An open protocol for decentralized exchange on the Ethereum blockchain. *URL: https://github.com/0xProject/whitepaper* (2017), 04–18.

[51] Lyu Xu, Jiaxin Jiang, Byron Choi, Jianliang Xu, and Bingsheng He. 2025. Bursting Flow Query on Large Temporal Flow Networks. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[52] Yehong Xu, Lei Li, Mengxuan Zhang, Zizhuo Xu, and Xiaofang Zhou. 2023. Global routing optimization in road networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2524–2537.

[53] Yokei Yamaguchi and Kaoru Yamaguchi. 2020. Modeling cross-border payments and foreign exchange dynamics. In *2020 Conf. System Dynamics Society*. University of Bergen, 20–24.

[54] 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).

[55] YCharts. 2025. *Binance Smart Chain Average Block Time (BSCABT)*. https://ycharts.com/indicators/binance_smart_chain_average_block_time Source: BscScan. Last updated Oct 14, 2025.

[56] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. Shortest-path queries on complex networks: experiments, analyses, and improvement. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2640–2652.
[57] Yu Zhang, Yafei Li, and Claudio Tessone. 2025. A Line Graph-Based Framework for Identifying Optimal Routing Paths in Decentralized Exchanges. *arXiv preprint arXiv:2504.15809* (2025).
[58] 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.
[59] 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.
[60] 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.
[61] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *IEEE International Conference on Data Engineering*. IEEE, 405–416.

Received October 2025; revised January 2026; accepted February 2026