We study the hop-constrained $s$-$t$ path enumeration (HcPE) problem, which takes a graph $G$, two distinct vertices $s$, $t$ and a hop constraint $k$ as input, and outputs all paths from $s$ to $t$ whose length is at most $k$. The state-of-the-art algorithms suffer from severe performance issues caused by the costly pruning operations during enumeration for the workloads with the large search space. Consequently, these algorithms hardly meet the real-time constraints of many online applications. In this paper, we propose PathEnum, an efficient index-based algorithm towards real-time HcPE. For an input query, PathEnum first builds a light-weight index aiming to reduce the number of edges involved in the enumeration, and develops efficient index-based approaches for enumeration, one based on depth-first search and the other based on joins. We further develop a query optimizer based on a join-based cost model to optimize the search order. We conduct experiments with 15 real-world graphs. Our experiment results show that PathEnum outperforms the state-of-the-art approaches by orders of magnitude in terms of the query time, throughput and response time.

1 INTRODUCTION

Because paths are widely used to measure the relationship between vertices, HcPE serves as an important building brick in a number of emerging real-world applications. Furthermore, HcPE can be easily extended with variant constraints to capture complexities of these applications. For example:

1) Detecting Money Laundering [12, 16, 24]. Money laundering is the illegal process of injecting "dirty" money into the legitimate financial system, typically by using a bank's services to move illegal money from source accounts into destination accounts through a series of transactions. We can construct a graph by representing bank accounts as vertices and transactions as edges. The report [12] lists a number of known "red flag indicators" which are regarded as indicative of money laundering. For example, the use of multiple bank accounts as well as that of intermediaries without good reasons is a red flag, which is also used in [16, 24]. They observe many instances of money laundering along short flow paths (e.g., two-hop), noting that longer paths can increase costs for the fraudsters. As such, this flag can be detected by enumerating hop-constrained paths between two target accounts. Moreover, banks or regulatory bodies may designate certain factors as risky (e.g., capital from foreign companies). Since a single risk factor may not be conclusive on its own, we want to find transactions exhibiting a certain level of total risk. In that case, we associate each edge with a weight representing the risk factor and extend HcPE by requiring that the accumulative value of weights on edges in a path is above a threshold.

2) E-Commerce Merchant Fraud Detection [32]. The activities of online shopping can be modeled as a graph in which vertices are individual users (e.g., sellers and buyers) and edges are online transactions (e.g., online payment and shipment of goods). In order to increase the popularity of products, some sellers create fake transactions. In brief, the entire process generates cycles in the graph. Therefore, the cycles triggered by new edges are strong indications of potential fraud. In the applications, [32] enumerates the cycles within a small hop constraint (e.g., $k = 6$) because a large hop constraint can result in a huge number of results causing massive false alarms. This also suggests the usage of hop constraints. We can issue a query $q(v', v, k - 1)$ to find paths from $v'$ to $v$ to enumerate cycles triggered by the new edge $e(v, v')$. Moreover, we may also impose constraints based on attributes of edges (e.g., monitor fake transactions with particular types of user activities [32]). Then, we can express the constraints as predicates on edges, and extend HcPE by requiring that each edge in a path satisfies conditions in predicates.

3) Knowledge Graph Completion [40]. A variety of applications such as recommendation systems, search and question answering depend on knowledge graphs (KGs). Because KGs are generally incomplete, the problem of knowledge graph completion, which aims to predict missing relations in KGs, is very important. In particular, paths between two entities indicate the relationship between them, and the knowledge graph completion methods generally use these paths to train models to predict the relationship. Previous work has observed that entities connected by many short paths have a higher tendency to be related, e.g., [34, 35], suggesting the utility of hop constraints in this setting. Furthermore, real-world applications may require that the paths satisfy the constraints on the sequence of actions (e.g., the sequence "write->mention"). In that case, each edge label represents an action, and we extend HcPE by requiring that the label sequence of each path meets the constraint on the sequence of actions.
Due to its importance, the HcPE problem has recently received significant interest. Existing approaches [14, 29, 33] focus on designing the polynomial delay algorithms such that the time between finding two successive results is bounded by a polynomial function of the input size in the worst case [19]. They adopt the backtracking method to recursively enumerate paths from the source to the target. To achieve polynomial delay, they introduce pruning rules at each recursive call to reduce the invalid search space, for example, performing a single source shortest path query from the target to update the distance between each vertex and the target [33]. Benefiting from the pruning strategies, the delay per output is within $O(k \times |E(G)|)$ time where $k$ is the length constraint and $|E(G)|$ is the number of edges in $G$.

Despite their theoretical guarantees, we find that these algorithms suffer from serious performance issues in practice. For the workloads with the large search space, the pruning at each step is expensive, to the extent that the pruning overhead can offset its benefits of reducing the search space. This fails to satisfy the requirement from many applications, especially the online scenarios [20, 32] with rigid real-time requirement on query time.

In this paper, we propose PathEnum, an efficient approach to the HcPE problem. In contrast to existing algorithms [14, 29, 33] that conduct pruning operations during the enumeration, the key design principle of PathEnum is to develop a light-weight index for the input query so that the index can be used to keep each step in the enumeration simple and efficient.

We first design a join-based model to abstract the HcPE problem, and analyze two key performance factors in the model, which are the number of edges involved in the enumeration and the order of enumerating results. Next, we develop an index-based approach to evaluate the query. Specifically, given a graph $G$ and a query $q(s, t, k)$, we first build a light-weight index $I$ at runtime, which is constructed based on the distance (i.e., the length of the shortest path) between each vertex to $s$ and $t$. The index is used to reduce the number of edges accessed during the enumeration. Given a vertex $v$ and an integer $b$, we can quickly retrieve the neighbors $v'$ of $v$ such that the distance from $v'$ to $t$ (or from $s$ to $v'$) is bounded by $b$ from $I$. The time complexity of constructing $I$ is $O(|E(G)| + |V(G)|)$.

We further develop a cost-based query optimizer to optimize the order of enumerating results. Specifically, we develop a depth-first search based method and a join-based method to enumerate the results based on $I$. The DFS-based method recursively extends the partial result by one vertex at a step to enumerate all results, whereas the join-based method first cuts the query into two subqueries, and then evaluates them with the DFS-based method, respectively, and finally join the intermediate results of the two subqueries. These two approaches can generate different number of partial results during the enumeration. The query optimizer selects the method with a lower cost to evaluate the query.

We conduct extensive experiments with 15 real-world datasets. The experiment results show that PathEnum provides speedups of 1.9 to 240.7 times over the state-of-the-art method [29] in terms of query time and 14.2 to 358.5 times in terms of response time. In summary, we make the following contributions in this paper.

- We study the hop-constrained $s$-$t$ path enumeration problem, and propose PathEnum, an efficient solution towards practical and real-time enumeration in many online applications.
- Different from existing backtracking solutions, PathEnum is an efficient index-based approach for HcPE. For each query, we first develop an efficient light-weight indexing method to prune the vertices involved in the subsequent enumeration. During the enumeration process, we design two index-based approaches, and an effective join order optimization method to reduce the search space.
- We conduct extensive experiments with a variety of workloads, and demonstrate PathEnum significantly outperforms state-of-the-art algorithms.

Supplement results are presented in the appendix. Our source code is publicly available at GitHub [37].

**Paper Organization.** We introduce the preliminaries and related work in Section 2. In Section 3, we formulate the HcPE problem in a join model and give an overview of PathEnum. We design a light-weight index, and develop index based enumeration approaches in Sections 4 and 5, respectively. We optimize the search order in Section 6. We present the experiment results in Section 7 and conclude in Section 8.

## 2 BACKGROUND AND RELATED WORK

### 2.1 Preliminaries

A directed graph $G = (V, E)$ denotes a directed graph where $V$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. $e = (v, v')$ denotes a directed edge from the vertex $v$ to the vertex $v'$. $N(v) = \{v' | e(v, v') \in E\}$ represents the outgoing neighbors of $v$, and $d(v)$ denotes the out degree of $v$, i.e., $d(v) = |N(v)|$. By default, the neighbors of $v$ refer to the outgoing neighbors. $G'$ represents the graph obtained by reversing the direction of each edge in $G$. Given two vertices $v$ and $v'$, the distance from $v$ to $v'$, denoted by $d(v, v')$, is the length of the shortest path from $v$ to $v'$ in $G$. $G - \{v\}$ represents the graph that removes $v$ as well as edges connecting with $v$ from $G$.

A walk $W$ is a sequence of vertices $(v_0, v_1, ..., v_i)$ such that $\forall 1 \leq i \leq |W| - 1$. An edge $(v_i, v_{i+1}) \in E \subseteq V \times V$. $|W|$ denotes the number of vertices in $W$, while $L(W)$ represents the number of edges in $W$. Therefore, $L(W) = |W| - 1$ when $W$ is not empty. $W[i]$ denotes the $i$th vertex in $W$ where $0 \leq i \leq |W| - 1$. Given two distinct vertices $s$ and $t$, we define a walk from $s$ to $t$ in Definition 2.1. $W(s, t, k, G)$ represents all walks $W$ from $s$ to $t$ in $G$ that satisfy $L(W) \leq k$. A path $P$ is a walk in which all vertices are distinct. Then, a path from $s$ to $t$ is a walk from $s$ to $t$ in which all vertices are distinct. $P(s, t, k, G)$ represents all paths $P$ from $s$ to $t$ such that $L(P) \leq k$. Apparently, $P(s, t, k, G)$ belongs to $W(s, t, k, G)$. Table 1 summarizes notations frequently used in this paper.

**Definition 2.1.** A walk from $s$ to $t$ is a walk $W$ such that (1) $W[0] = s \land W[|W| - 1] = t$; and (2) $\forall 0 < i < |W| - 1, W[i] \notin \{s, t\}$.

**Problem Statement.** Given $G = (V, E)$, two distinct vertices $s, t$ and a hop constraint $k$, the hop-constrained $s$-$t$ path enumeration (HcPE) problem aims to find all paths in $P(s, t, k, G)$. The query is denoted by $q(s, t, k)$. We assume that $k \geq 2$ in this paper.

### 2.2 State-of-the-art Approaches

Algorithm 1 illustrates a generic depth-first search based framework to find $P(s, t, k, G)$. It adopts the backtracking strategy. $M$ stores a sequence of vertices, which initially contains $s$ (Line 1). Line 5
Table 1: A summary of notations frequently used.

| Notations | Descriptions |
|-----------|--------------|
| $s, t$ and $k$ | source, target and length constraint |
| $G$ and $q(s, t, k)$ | graph and HcPE query |
| $Q$ and $R$ | join query and relation |
| $V(G)$ and $E(G)$ | vertex and edge sets of $G$ |
| $d(v)$ and $N(v)$ | degree and neighbors of $v$ |
| $P, W$ and $M$ | path, walk and partial result |
| $L(P), L(W), L(M)$ | number of edges in $P, W$ and $M$ |
| $\mathcal{P}(s, t, k, G)$ | paths $P$ from $s$ to $t$ with $L(P) \leq k$ |
| $W(s, t, k, G)$ | walks $W$ from $s$ to $t$ with $L(W) \leq k$ |
| $\Delta_p$ and $\Delta_W$ | $|\mathcal{P}(s, t, k, G)|$ and $|W(s, t, k, G)|$ |
| $d(s, t, k, G)$ | distance between $s$ and $t$ in $G$ |
| $I$ | the light-weight index |
| $I(i)$ | vertices $v$ satisfying $S(v, i|G - \{t\}) \leq i$ and $S(v, i|G - \{s\}) \leq k - i$ |
| $I_p(v, b)$ | neighbors $v'$ of $v$ satisfying $S(v', i|G - \{s\}) \leq b$ |

Algorithm 1: Generic DFS based Framework

**Input:** a graph $G$, two distinct vertices $s, t$, hop constraint $k$;

**Output:** all $k$ hop-constrained paths from $s$ to $t$;

$M \leftarrow \{s\}$;

1. **Search** $(t, k, M)$;
2. **Procedure** Search $(t, k, M)$
3. $a$ ← the last vertex in $M$;
4. if $v = t$ then emit $(M)$, return;
5. foreach $v' \in N(v)$ do
6. if $v' \notin M$ and $L(M) + 1 + B(v') \leq k$ then
7. **Search** $(t, k, M \cup \{v'\})$

2.3 Other Related Work

**s-t Path (or Cycle) Enumeration.** Another kind of algorithms [7, 27, 41] focus on developing construction methods to compile $s$-$t$ paths into a representation structure such that these paths can be quickly listed without explicitly storing each individual result. These algorithms can only handle graphs with hundred vertices because compiling $s$-$t$ paths of large graphs can consume a large amount of memory. Enumerating all $s$-$t$ paths (or cycles) without the hop constraint is a classical problem [6, 18, 21, 39]. However, these algorithms cannot be easily extended to the scenarios with the hop constraint because their enumeration procedure does not consider the impact of the hop constraint. Additionally, there are also a variety of works [4, 5, 15] that focus on detecting the existence of cycles in dynamic graphs instead of enumerating the results.

**Top-K Shortest Path Enumeration.** We can evaluate a query $q(s, t, k)$ with the Top-K shortest path algorithms [8, 11, 13, 25, 36, 42]. In particular, we set $K$ as a sufficient large value and terminate the query when the length of results is greater than $k$. Despite that these algorithms can find the results of $q(s, t, k)$, they enumerate results along the ascending order of the length of results, which is unnecessary for the HcPE problem and incurs overhead.

**Subgraph Matching.** Given a data graph and a query graph, subgraph matching finds all embeddings in the data graph that are identical to the query graph [22, 38]. Existing graph database systems such as EmptyHeaded [1] and GraphFlow [26] enumerate all results by performing self-joins on $G$, and propose variant join plan optimization methods in which the cardinality estimation plays an important role. Existing estimation methods [28] work on input relations of the join query [23] or catalogues [26] that are built in an offline preprocessing step and summarize the global statistics of $G$. For example, given $G$, GraphFlow uses sampling methods to build a catalog by collecting the number of subgraphs with some specific structures appearing in $G$. Given a query, GraphFlow optimizes the join plan by considering different plans of constructing the query graph from its subgraphs, estimating the cost (e.g., the number of partial results) based on the catalog and selecting the plan with the minimum cost. GraphFlow evaluates the query according to the plan and adopts the intersection caching to reduce the cost of set intersections. In summary, the query optimizer and the computation of existing systems are optimized for reducing the cost of finding all subgraphs in $G$ with a specific structure (e.g., a path).

In contrast, the HcPE problem targets at paths from $s$ to $t$ that satisfy the length constraint. Moreover, our method evaluates the query on a query-dependent index, which is built online for each query based on distances to $s, t$, without building relations. The
index rules out many invalid candidates for the query in $G$. Our query optimizer as well as the cardinality estimation method is designed specially to work with the index.

**Distance Queries.** A distance query asks the distance between two vertices in a graph, which receives a lot of research interests [3, 9, 10, 17, 30, 31]. Existing methods such as the pruned landmark labeling [3] construct an index in an offline preprocessing step to serve all queries, and evaluate the query with the pre-computed results. The index records the distance to a set of vertices for each vertex in the graph, which maintains the global statistics of $G$. They focus on balancing the cost of building indexes and query efficiency.

In contrast, the light-weight index proposed in this paper is query-dependent, which is built based on the distance to $s$ and maintains the local statistics for the given query.

## 3 ALGORITHM OVERVIEW

In this section, we first propose a join-based model to the HcPE problem, and then give an overview of our PathEnum.

### 3.1 A Join-based Model

Although existing algorithms [14, 29, 33] provide comprehensive analysis to the HcPE problem in terms of the time complexity, there lacks a method to model the practical computation cost of evaluating a query. To reveal the problem, we formulate a HcPE query $q(s, t, k)$ on $G$ as a chain join $Q$. The edge list $E(G)$ can be viewed as a binary relation $R(u, u') = \{(u, u') | e(u, u') \in E(G)\}$.

At first glance, the query $q(s, t, k)$ can be easily translated to a chain join $Q = R_1(u_0, u_1) \bowtie R_2(u_1, u_2) \bowtie \cdots \bowtie R_k(u_{k-1}, u_k)$ where $R_1 = \{(s, v) | e(s, v) \in E(G)\}, \ R_k = \{(v, t) | e(v, t) \in E(G)\}$ and $R_i = \{(v, v') | e(v, v') \in E(G)\}$ when $1 \leq i < k$. For the ease of presentation, we use $Q$ to represent the results of evaluating $Q$ as well. To obtain $P(s, t, k, G)$, we first evaluate $Q$, and then eliminate the tuples $r \in Q$ that have duplicate vertices. However, this method only returns the paths from $s$ to $t$ the length of which are exactly $k$. Although we can solve this problem by launching $k$ chain join queries to compute paths with different lengths, this approach incurs a large amount of redundant computations. To solve the problem, we propose to generate relations of $Q$ as follows.

1. $R_1 = \{(s, v) | e(s, v) \in E(G)\}$ and $R_k = \{(v, t) | e(v, t) \in E(G)\}$
2. $R_i = \{(v, v') | e(v, v') \in E(G)\}$ when $1 \leq i < k$
3. $R_i = R_i \cup \{(t, t)\}$ for $1 \leq i \leq k$

The first two properties ensure that each tuple in $Q$ starts and ends at $s$ and $t$, while the third property avoids eliminating the paths $P \in P(s, t, k, G)$ that satisfy $L(P) < k$. With the generation method, we can get Theorem 3.1. Example 3.2 presents a running example. Due to space limit, the proof of Theorem 3.1 and other propositions in this paper is presented in the appendix.

## 4 INDEX CONSTRUCTION

### 4.1 Relation Construction

Benefiting from the join-based model, we can evaluate $q(s, t, k)$ on $G$ as a chain join $Q$. To reduce the cost, we can eliminate the dangling tuples, i.e., the tuples not existing in any results, from each relation of $Q$ with the full reducer, which is a classical dangling

![Figure 1: A query $q(s, t, 4)$ on the graph $G$.](image)

![Figure 2: An overview of PathEnum.](image)
Figure 3c presents the relations after filtering from \( R \) a running example. Relations in \( R \) remove dangling tuples from these relations. Specifically, Lines 5–8 illustrate the details. Lines 1–4 generate relations understanding, we present the algorithm in graph context. Algorithm 2: Build Relations

**Input:** a graph \( G \), two distinct vertices \( s, t \), hop constraint \( k \);

**Output:** a set of relations \( R \);

```
1 R1 ← \{(s, s)\} \in E(G)\};
2 R2 ← \{(s, t)\} \in E(G) \land v \neq s \lor \{\langle t, t \rangle\};
3 for i ← 2 to k − 1 do
4 \[ R_i ← \{(u, v')\} \in E(G) \setminus \{s\} \land v \neq t \lor \{(t, t)\}; \]
```

**Algorithm 2: Build Relations**

```
5 /* Initialize relations. */
6 for i ← 1 to k − 1 do
7 \[ C ← \{v'|(u, v') \in R_i\}; \]
8 \[ \text{for } (u, v') \in R_i \text{ do } \]
9 \[ \text{if } u \in C \text{ then } R_{i+1} \leftarrow R_{i+1} \setminus \langle u, v' \rangle; \]
10 \[ C ← \{v|v' \in R_{i+1}\}; \]
11 \[ \text{for } v' \in C \text{ do } \]
12 \[ \text{if } v' \notin C \text{ then } R_i ← R_i \setminus \langle v, v' \rangle; \]
13 return \( R_{k-1} \).
```

**Algorithm 3: Build Relations**

```
1 u1 u2 v0 v1 v0 v4
2 u2 u3 v1 v2 v1 v3 v2 v0 v2 v3 v4 v4 v5 v6 v0 t t
3 u0 u1 s v0 s v1 s v3
4 u3 u4 v0 t v2 t v5 t t t
5 R1 R2 R3 R4
```

**Figure 3: Build relations \( R \) of \( Q \).**

R tuple elimination method in relational databases [2]. For ease of understanding, we present the algorithm in graph context. Algorithm 2 illustrates the details. Lines 1–4 generate relations \( R \) of \( Q \) based on the construction method introduced in Section 3.1. Lines 5–12 remove dangling tuples from these relations. Specifically, Lines 5–8 prune relations \( R_i \) along the increasing order of \( i \). Line 6 obtains the end vertex of edges in \( R_i \). Next, Lines 7–8 remove edges \( (v, v') \) in \( R_{i+1} \) such that \( v \) does not belong to \( C \). After that, Lines 9–12 filter relations \( R_i \) along the decreasing order of \( i \) with the same method as Lines 5–8. Finally, Line 13 returns the relations. The following is a running example.

**Example 4.1.** Given \( G \) and \( q(s, t, 4) \) in Figure 1, the relations generated by Lines 1–4 are shown in Figure 3a. Figure 3b illustrates the relations after pruning from \( R_2 \) to \( R_3 \). For example, \((u_4, v_5)\) is removed from \( R_2 \) because \( u_4 \) does not appear in values of \( u_1 \) in \( R_1 \). Figure 3c presents the relations after filtering from \( R_{k-1} \) to \( R_k \). For example, \((v_1, v_3)\) is eliminated from \( R_3 \) since \( v_3 \) does not exist in values of \( u_3 \) in \( R_4 \).

The space and time complexities are both \( O(k \times |E(G)|) \). The relations returned by Algorithm 2 satisfy the following proposition.

Algorithm 3: Build Index

```
1 R ← \{s, v|G\{\} \} and \{s, t|G\{\} \} \forall s \in V(G);
2 X ← \{k (k + 1) \times (k + 1)\} \text{ of sets};
3 \text{foreach } v \in V(G) \text{ do } \]
4 \[ \text{if } u \in X[s, v, u, t] \text{ then } Add v to } X[s, u, t]; \]
5 \[ H ← \text{a hash table}; \]
6 \[ \text{foreach } v \in X \{t \} \text{ do } \]
7 \[ \text{Add a key-value pair } (v, \{t\}) \text{ to } H; \]
8 \[ \text{foreach } v' \in N(v) \text{ do } \]
9 \[ \text{if } v \neq v \text{. } v' \text{. } t + 1 \leq k \text{ then } Add v' \text{ to } H[v]; \]
10 \[ \text{Add a key-value pair } (t, \{t\}) \text{ to } H; \]
11 \[ \text{Sort the values } v' \text{ of } H[v] \text{ of } v \text{ by ascending order of } v'.t; \]
12 return \( I(X, H) \).

**Proposition 4.2.** Each tuple in relations \( Q \) appear in the final results of evaluating \( Q \) [2].

**4.2 Light-Weight Index**

Algorithm 2 removes dangling tuples at the cost of scanning \( G \) and each relation several times. The cost can dominate the execution time of some queries, especially on large graphs. This makes the algorithm hard to meet the real-time constraint. In order to reveal the problem, we propose a light-weight index, which is built at a small overhead but provides competitive pruning power.

**General Idea.** Based on the definition of a path from \( s \) to \( t \), we have the following proposition.

**Proposition 4.3.** Given a vertex \( v \in V(G) \), if there exists a path \( P \in \mathcal{P}(s, t, k, G) \) such that \( P[i] = v \) where \( 0 \leq i < |P| - 1 \), then \( S \in \mathcal{P}(s, v|G\{\} \} \leq k \) and \( S \in \mathcal{P}(s, v|G\{\} \} \leq k \).

Let \( C \) denote the set of vertices \( v \in V(G) \) that satisfy \( S \in \mathcal{P}(s, v|G\{\} \} \leq k \). According to Proposition 4.3, if \( v \) appears at position \( i \) of \( P \in \mathcal{P}(s, t, k, G) \), then \( v \) belongs to \( C \). Moreover, given a vertex \( v \), suppose that the remaining budget traveling to \( t \) is \( b \). We only need to consider the neighbors \( v' \) of \( v \) such that \( S \in \mathcal{P}(s, v'|G\{\} \} \leq k \). Based on the observation, we want to build an index supporting two kinds of lookup operations to serve the subsequent enumeration: (1) given \( i \) where \( 0 \leq i < k \), retrieve \( C \); and (2) given \( v \) and an integer \( k \) where \( 0 < b < k \), retrieve the neighbors \( v' \) such that \( S \in \mathcal{P}(s, v'|G\{\} \} \leq k \). The index \( I(X, H) \) contains \( X \) and \( H \). In practice, \( H \) has three components that is the \( Neighbors \) array storing neighbors \( v' \) of each vertex \( v \) in \( X \) by the ascending order of \( v'.t \). Finally, we return the index \( I \) that contains \( X \) and \( H \).
which are the vertices in X and a pointer to the beginning position at the Offset array. The following is an example.

Example 4.4. Figure 4 presents I on G and q(s, t, 4) in Figure 1. Figure 4a shows the partitions X of v ∈ V(G) based on S(v, t) (G − {t}) and S(v, t) (G − {s}). For example, X[2, 3] = {v4, v6}. Figure 4b demonstrates the implementation of H. Take v0 as an example. It has three neighbors {t, v1, v6}, which are stored in the Neighbors array by the ascending order of the distance from t. As k = 4, v0 has five slots in the Offset array to index {t, v1, v6} based on the distance to t. The value in H is 0, which points to the begin position of slots belonging to v0 in the Offset array. Suppose that we want to retrieve the neighbors v of v0 such that S(v, t) (G − {s}) ≤ 2. We first get the beginning position of the neighbor set of v0 from the first slot of the Offset array, which is 0. Next, we get the end position of the neighbors satisfying the distance constraint from the fourth slot, which is 3. Then, we get the results from the Neighbors array, which are {t, v1, v6}.

Index Lookup Operations The index I supports two kinds of operations listed below.
- \( I(i) \): Retrieve \( C_i \), i.e., the vertices \( v \in V(G) \) satisfy \( S(v, t) (G − \{t\}) ≤ i \) and \( S(v, t) (G − \{s\}) ≤ k − i \).
- \( I(t, v) \) (or \( I(v, t) \)): Retrieve the neighbors \( v' \) of \( v \) such that \( S(v', t) (G − \{s\}) ≤ b \) (or the in neighbors \( v' \) of \( v \) such that \( S(v, v') (G − \{t\}) ≤ b \)).

\( I(i) \) and \( I(t, v) \) are implemented based on X and H, respectively. The time complexity of the two operations are both \( O(1) \).

4.3 Analysis

We compare the pruning power of Algorithm 3 with Algorithm 2 in the appendix. In this following, we analyze the space and time complexities of Algorithm 3.

Space. The space complexity of X is \( O((k+1)^2+|V(G)|) \) because X contains all vertices of G at most. The space complexity of H is \( O(|E(G)|+k \times |V(G)|) \) because we store all edges in E(G) at most and each vertex has k + 1 slots in the Offset array. Therefore, the space complexity of constructing I is \( O(|E(G)|+k \times |V(G)|) \).

Time. Line 1 in Algorithm 3 takes \( O(|E(G)|+|V(G)|) \) time because we perform two breadth-first searches. Lines 2-4 takes \( |V(G)| \) time. Since Lines 6-9 loop over the neighbors of each vertex in X, the cost is \( O(|E(G)|) \). We implement the sort at Line 11 with the counting sort because \( k \) is small. Therefore, the cost is \( O(|E(G)|) \) as well. In summary, the time complexity of Algorithm 3 is \( O(|E(G)|+|V(G)|) \). The cost in practice is small because many vertices and edges are ruled out by the distance constraint.

Algorithm 4: Depth-First Search on Index

Input: two distinct vertices s, t, hop constraint k, index I.
Output: all k hop-constrained paths from s to t.

1. \( M \leftarrow \{s\} \);
2. Search \( (t, k, M, I) \);
3. Procedure Search \( (t, k, M, I) \)
   4. \( v \leftarrow \) the last vertex in \( M \);
   5. if \( v = t \) then emit(M), return;
   6. foreach \( v' \in I(t, k-1) \) do
      7. if \( v' \notin M \) then Search \( (t, k, M \cup \{v'\}, I) \);

5 SEARCH ON INDEX

5.1 Depth-First Search on Index

Algorithm 4 presents the depth-first search method on the index I. The Search procedure recursively enumerates all hop-constrained paths from s to t based on I (Lines 3-7). If the last vertex \( v \) in M is t, then we find a result and emit it (Line 4). Otherwise, we consider the neighbors \( v' \) of \( v \) such that \( S(v', t) (k-1) ≤ k-1 \) as the next vertex in M to meet the hop constraint. In particular, we loop over \( I(t, k-1) \), add \( v' \) to M, and continue the search. The check at line 7 ensures that there are no duplicate vertices in M.

5.2 Analysis

Algorithm 4 can be easily extended to support HePE queries with variant constraints such as accumulative values and a sequence of actions, which is detailed in the appendix. In the following, we focus on the space consumption and time complexity.

Space. Algorithm 4 spends \( O(k) \) space to store partial results because it maintains one partial result at a time during the search.

Time. Algorithm 4 performs a DFS on the search tree where nodes are partial results M and edges are operations of adding a vertex to M. We analyze the time complexity based on the search tree. Let \( M_i \) represent the nodes at depth i in the search tree, which are the set of partial results M containing i+1 vertices. Each internal node \( M_i \) in \( M_i \) corresponds to an invocation of the Search procedure. The cost of an invocation is \( |I(t, k-1)| \) (i.e., the for loop at Lines 6-7). Then, the running time \( T \) of Algorithm 4 is computed by Equation 2.

\[
T = \sum_{0 \leq i \leq k-1} \sum_{M \in M_i} |I_t(M[i], k-i-1)|. \tag{2}
\]

As Algorithm 4 generates partial results incrementally, \( M_{i+1} \) and \( M_i \) have the following relation where \( 0 \leq i \leq k-1 \).

\[
M_{i+1} = \bigcup_{M \in M_i} \{M \cup \{v\} | v \in I_t(M[i], k-i-1) - M\}. \tag{3}
\]

Because M is generated during the enumeration, it is hard to estimate the number of vertices \( v \in I_t(M[i], k-i-1) \) belonging to M. For the ease of analysis, we relax the constraint of Algorithm 4 by removing the check at line 7 because \( k \) is small and M contains a few vertices. Let \( \tilde{M}_i \) denote the set of partial results containing \( i+1 \) vertices, which are generated by the algorithm after relaxation. According to Equation 3, \( \tilde{M}_{i+1} = \bigcup_{M \in \tilde{M}_i} \{M \cup \{v\} | v \in I_t(M[i], k-i-1)\} \) and \( M_i \subseteq \tilde{M}_i \). The algorithm after relaxation satisfies the following proposition.
Algorithm 4 enumerates all results through extending the partial result from $s$ to $t$ in $G$. Given $M \in \tilde{M}_t$ where $0 \leq i \leq k$, $M$ must appear in a walk $W \in \mathcal{W}(s, t, k, G)$.

The proposition shows that each leaf in the search tree of Algorithm 4 after relaxation is a walk $W \in \mathcal{W}(s, t, k, G)$. Then, $|\tilde{M}_t| \leq \delta_W$ where $\delta_W = |\mathcal{W}(s, t, k, G)|$. Together with Equations 2 and 3, we get the following equation.

$$T = \sum_{0 \leq i < k-1} \sum_{M \in \tilde{M}_i} |I_t(M[i], k-i-1)|$$

$$\leq \sum_{0 \leq i < k-1} \sum_{M \in \tilde{M}_i} |I_t(M[i], k-i-1)|$$

$$= \sum_{0 \leq i < k-1} \sum_{M \in \tilde{M}_i} \left\{ |\mathcal{W}(s, t, k, G)| \right\}$$

$$= \sum_{0 \leq i < k-1} |\tilde{M}_i| \leq k \times \delta_W.$$

In summary, given $G$ and $q(s, t, k)$, the running time of Algorithm 4 is $O(k \times \delta_W)$. The analysis indicates that most of walks in $\mathcal{W}(s, t, k, G)$ belong to $\mathcal{P}(s, t, k, G)$, Algorithm 4 generates a few invalid partial results, i.e., the partial results do not exist in any final results, and its running time is very close to the lower bound $\Omega(\delta_P)$ for the problem where $\delta_P = |\mathcal{P}(s, t, k, G)|$. In contrast, when most walks in $\mathcal{W}(s, t, k, G)$ are not paths, the algorithm can result in a large number of invalid partial results. Example 5.2 presents an example. From the example, we can also see that the gap between $\delta_P$ and $\delta_W$ (i.e., $\left(\frac{\delta_W}{\delta_P}\right)$) depends on the query and the graph topology.

Example 5.2. We execute a query $q(s, t, 4)$ on $G_0$ and $G_1$ in Figure 5, respectively. $\mathcal{W}(s, t, 4, G_0)$ is a path in $\mathcal{P}(s, t, 4, G)$, for example, $W = (s, v_0, v_2, u_4, t)$. In contrast, $|\mathcal{W}(s, t, 4, G_1)|$ is equal to 6, and only $W = (s, v_0, t)$ belongs to $\mathcal{P}(s, t, 4, G_1)$.

6 QUERY OPTIMIZATION

6.1 General Idea

Algorithm 4 enumerates all results through extending the partial result from $s$ by one vertex at a step. It is equivalent to evaluating $Q$ along the join order $(R_1, R_2, ..., R_k)$, which is a left-deep join tree. Because the join order has an important impact on the cost of the enumeration, we want to optimize it to further accelerate the query. On the other hand, an important observation we made on the HePE problem is that the running time of different queries varies greatly. We can easily answer the query with a small search space regardless of join orders. Consequently, the benefit of optimizing join orders on these queries is limited. Even worse the optimization time dominates the query time if the optimization method is complex.

In order to reveal the problem, we propose an optimizer generating join orders based on the index in two phases. In particular, we first use a preliminary cardinality estimator to roughly but quickly estimate the size of the search space. If the search space size is small, then we directly invoke Algorithm 4. Otherwise, we generate a join order with a full-fledged cardinality estimator, which provides more accurate estimation but at a higher cost. The optimizer selects the method (Algorithm 4 versus. Algorithm 6) with lower cost to evaluate the query.

6.2 Cardinality Estimator

Preliminary Cardinality Estimator. Given $q(s, t, k)$ on $G$, the preliminary estimator aims to estimate the size of the search space roughly but quickly. Based on the analysis in Section 5.2, the size of the search space can be estimated as $\sum_{0 \leq i < k} |\tilde{M}_i|$ where $\tilde{M}_0 = (s)$ and $\tilde{M}_i = \bigcup_{M \in \tilde{M}_{i-1}} \{M \cup \{v\} | v \in I_t(M[i-1], k-i)\}$. Suppose that the average number of immediate partial results derived from partial results in $\tilde{M}_i$ is $\gamma_i$. Then, $|\tilde{M}_{i+1}| = \gamma_i \times |\tilde{M}_i|$.

To calculate the size of the search space, we would like to estimate $\gamma_i$. Given $M \in \tilde{M}_i$, the number of immediate partial results derived from $M$ is $|I_t(M[i], k-L(M)-1)|$. $M[i]$ must belong to $C_i$ based on Proposition 4.3. Then, an intuitive method for estimating $\gamma_i$ is to estimate it as the average number of neighbors $\forall'$ of vertices $v$ in $C_i$ that satisfy $\forall'(\forall', G - (s)) \leq k - L(M) - 1$, i.e., $\frac{1}{|\forall'|} \sum_{v \in C_i} |I_t(v, k - L(M) - 1)|$. The estimated value is denoted by $\hat{\gamma}_i$. Total, the estimated size $\hat{T}$ of the search space is computed by Equation 5. The value of $\hat{\gamma}_1$ is a basic statistics of $\forall'$, which is collected during the index construction. Then, the time complexity of computing the equation is $O(k^2)$, which incurs a small cost.

$$\hat{T} = \sum_{1 \leq i < k} |\tilde{M}_i| \approx \sum_{0 \leq i < k-1} \frac{1}{|\forall'_i|} \sum_{v \in \forall'_i} |I_t(v, k - L(M) - 1)|$$

We compare $\hat{T}$ with a threshold $\tau$. If $\hat{T} > \tau$, then we optimize the join order with the full-fledged cardinality estimator. Otherwise, we evaluate the query with Algorithm 4. Therefore, we set $\tau$ such that the cost of the optimization is neglected compared with the search time when $\hat{T} > \tau$. In particular, given $G$, $\tau$ is measured by pre-executing some random queries with Algorithm 4 and testing $\tau$ from $10, 10^2, ...,$, till the time finding $\tau$ results is longer than the join plan optimization time for most of queries. In our experiments, we execute 100 queries and set $\tau$ as $10^2$, which works well in our workloads. This is because the optimization time on these graphs is generally shorter than the time of finding $10^2$ results. Moreover, if the queries have fewer than $10^3$ results, then the enumeration time is small (several milliseconds), which makes the gain of optimization limited. As such, we directly use Algorithm 4 to answer them.

Full-fledged Cardinality Estimator. The full-fledged estimator gives an accurate estimation on the size of the search space. To avoid performing the Cartesian product of two relations, we require that the two relations in a join operation must have common attributes. Therefore, each sub-query $Q'$ is a sub-chain of $Q$, $Q[i : j]$ denotes a sub-query $Q' = R_{i,j}(u_i, u_{i+1}) \times \cdots \times R_j(u_{j-1}, u_j)$. We want to estimate $|Q[i : j]|$ based on the index.
We first consider a simple case $Q' = Q[i : i+1]$ where $0 \leq i < k$, i.e., $Q'$ is a base relation $R_{i+1}(M_i, R_{i+1})$. Based on $I$, we obtain that $R_{i+1} = \bigcup_{v \in I(i)} (v, v')$ for each $v \in I(i)$. Thus, $\{Q'[i : i+1]\} = \bigcup_{v \in I(i)} R_{i+1}(v, v')$. Let $c_k(v)$ denote the number of tuples starting with $v$ in $Q[i : i+1]$. Given $Q' = Q[i - 1 : i+1]$, we have $\{Q'[i : i+1]\} = R_{i} \leftarrow R_{i+1} = \sum_{v \in I(i)} c_k(v)$. Therefore, we compute $|Q[i : j]|$ as follows.

$$|Q[i : j]| = \sum_{v \in I(i)} c_j(v). \quad (6)$$

$$c_j(v) = \begin{cases} 1 & \text{if } i = j \\ \sum_{v' \in I(i) (v, v' - 1)} c_i(v') & \text{if } i < j \end{cases} \quad (7)$$

We estimate $|Q[i : k]|$ with a dynamic programming method based on the index $I$. Given $0 \leq i < k$, we store $c_k(v)$ for each vertex $v \in I(i)$. As such, the space cost of the full-fledged cardinality estimator is $\sum_{v \in I(i)} |I(i)| \times |V(G)|$. Based on Equation 7, we first set $c_k(v) = 1$ for each $v \in I(k)$. Given $0 \leq i < k$, $|Q[i : k]|$ is equal to $\sum_{v \in I(i)} c_k(v)$. Similar to Equation 6, the total space cost is $O((k \times |V(G)|) \times \sum_{v \in I(i)} |I(i)|)$, where the time complexity is $O(k \times |V(G)|)$. The number of edges in the index is generally smaller than $|V(G)|$ because of the filtering. The time complexity can be met when $G$ is a clique. The implementation of the estimator will be introduced in Algorithm 5.

**Algorithm 5: Join Order Optimization**

**Input:** two distinct vertices $s, t$, hop constraint $k$, index $I$.
**Output:** the cut position $i^*$.

```
/* Estimate number of paths from to t. */
1 set $c_k(v) = 0$ for each $v \in I(i)$ where $0 \leq i \leq k - 1$.
2 set $c_k(v) = 1$ for each $v \in I(k)$.
3 for $i \leftarrow k - 1$ to $0$ do
4    foreach $v \in I(i)$, $v' \in I((v, k - i - 1))$
5        $c_k(v') = c_k(v') + c_k(v)$
/* Estimate number of paths from s. */
6    set $c_k(v) = 0$ for each $v \in I(k)$ where $1 \leq i \leq k - 1$.
7    $c_k(v) = 1$ for each $v \in I(k)$.
8 for $i \leftarrow 1$ to $k$ do
9    foreach $v \in I(i)$, $v' \in I((v, k - i - 1))$
10       $c_k(v') = c_k(v') + c_k(v)$
/* Find the cut position $i^*$. */
11 $i^* \leftarrow \arg\min_{k \in \mathbb{N}} |\sum_{v \in I(i)} c_k(v) + \sum_{v' \in I(i)} c_k(v')|$.
12 return $i^*$.
```

6.3 Join On Index

**Join Order Optimization**. The cost of a join order can be estimated by the cost model in Equation 1 with the full-fledged cardinality estimator. Because it is prohibitively expensive to enumerate all orders, the number of which is exponential to the number of relations, to minimize the cost, we design a greedy optimization method. In particular, we minimize Equation 1 in a top-down manner by (1) cutting $Q$ into two sub-queries $Q[0 : i]$ and $Q[i : k]$ such that the sum of $|Q[0 : i]|$ and $|Q[i : k]|$ is minimized; and (2) cutting $Q[0 : i]$ and $Q[i : k]$ into smaller sub-queries, respectively, and continuing the process until each sub-query is a base relation.

However, the benefit of optimizing the orders of evaluating $Q[0 : i]$ and $Q[i : k]$ is limited for a query with a large search space. Specifically, a large search space indicates that the number of partial results grows exponentially with the length of the path increasing. Given any sub-query $Q'[0 : i]$ (or $Q[i : k]$), $|Q'|$ is much less than $|Q[0 : i]|$ (or $|Q[i : k]|$). Consequently, the last join operation $Q = Q[0 : i] \bowtie Q[i : k]$ dominates the evaluation cost. Therefore, we simplify the join order optimization as follows: (1) find a cut position $i^*$ of $Q$ such that the sum of $|Q[0 : i^*]|$ and $|Q[i^* : k]|$ is minimized; (2) evaluate $Q[0 : i^*]$ and $Q[i^* : k]$ with the depth-first search method, respectively; and (3) perform $Q[0 : i^*] \bowtie Q[i^* : k]$ to find final results.

Algorithm 5 illustrates the method finding the cut position $i^*$. Given $v \in I(i)$ (i.e., $C_i$) where $0 \leq i < k$, Lines 1-5 estimate the number of paths from $v$ to $t$ based on the full-fledged cardinality estimator. Similarly, Lines 6-10 estimate the number of paths from $s$ to $v$. Finally, Lines 11-12 find the cut position $i^*$ such that the sum of $|Q[0 : i]|$ and $|Q[i : k]|$ is minimized, and return it. The time complexity of Algorithm 5 is $O(k \times |V(G)|)$ and the space complexity is $O(k \times |V(G)|)$.

After finding the cut position, we compare the cost of the new order with that of Algorithm 4. In particular, Algorithm 4 is equivalent to the left-deep join along the order $(R_1, R_2, \ldots, R_k)$. The cost is $T_{DFS} = \sum_{i < k} |Q[i : i]|$ based on Equation 1. In contrast, the cost of the new order is $T_{JOIN} = |Q| + \sum_{i < k} |Q[i : i^*]| + T(Q[i^* : k]) = |Q| + \sum_{i < k} |Q[0 : i]| + \sum_{i < k} |Q[i^* : k]|$. Based on the intermediate results in Algorithm 5, we can get that $T_{DFS} = \sum_{i < k} \sum_{v \in I(i)} c_i(v)$, while $T_{JOIN} = \sum_{i < k} c_i(v) + \sum_{i < k} \sum_{v' \in I(k)} c_i(v')$. If $T_{DFS} < T_{JOIN}$, then we adopt Algorithm 4. Otherwise, we evaluate the query with the join-based method, which is introduced in Algorithm 6.

**Join Implementation**. Algorithm 6 presents the join-based method on the index. $R_0$ and $R_k$ store the results of evaluating $Q[0 : i^*]$ and $Q[i^* : k]$, respectively (Line 1). We first find the results of $Q[0 : i^*]$ with a depth-first search from $s$ (Line 2). $M$ is a sequence of vertices. If $M$ contains $t$ vertices, then we add it to $R$ and return (Line 10). Otherwise, we loop over the neighbors $v'$ of the last vertex $v$ in $M$ such that $S(v', t - |G - v'|) \leq k - i - L(M) - 1$, add $v'$ to $M$, and continue the search (Lines 11-13). After that, Line
3 collects all vertices appearing in the last position of tuples in \(R_a\), i.e., the values of the join key \(Q[i^+]\). Next, we find the results of \(Q[i^+ : k]\) with a depth-first search from each \(v \in C\) (Lines 4-5). Finally, we perform the hash join of \(R_a\) and \(R_b\) and output the valid path (Lines 6-8). In practical implementation, we check whether a result is a valid path when performing the join operation.

6.4 Analysis
Algorithm 6 satisfies Proposition 6.1. Based on the proposition, we analyze its space and time complexities.

**Proposition 6.1.** Each partial result \(M\) generated by the Search procedure appears in a tuple in \(R\). Each tuple in \(R\) corresponds to a walk in \(W(s, t, k, G)\).

**Space.** Algorithm 6 maintains intermediate results of evaluating \(Q[0 : i^+]\) and \(Q[i^+ : k]\), which are \(R_a\) and \(R_b\), respectively. Based on Proposition 6.1, each tuple in \(R_a\) (or \(R_b\)) appears in a result of \(R\). Therefore, the sizes of \(R_a\) and \(R_b\) are less than or equal to \(|R| = |W(s, t, k, G)|\), and the space complexity is \(O(k \times \delta_W)\).

**Time.** Because each partial result \(M\) appears in \(R_a\) (or \(R_b\)), the time complexity of evaluating \(Q[0 : i^+]\) and \(Q[i^+ : k]\) are \(O(|R_a| \times (i^+ + 1))\) and \(O(|R_b| \times (k - i^+ + 1))\), respectively. The time complexity of a hash join is \(O(|IN| + |OUT|)\) where \(|IN|\) and \(|OUT|\) are the sizes of the input and output, respectively. Therefore, the cost of evaluating \(R = R_a \bowtie_H R_b\) is \(O(|R_a| \times (i^+ + 1) + |R_b| \times (k - i^+ + 1)) + |R| \times k\). As both \(|R_a|\) and \(|R_b|\) are less than or equal to \(|R|\), the cost is \(O(k \times |R|) = O(k \times \delta_W)\). In summary, the time complexity of Algorithm 6 is \(O(k \times \delta_W)\).

**Discussion.** As discussed in Section 2.3, existing cardinality estimation methods generally take catalogs or relations as the input [28], which cannot be directly applied to our light-weight index. Our full-fledged estimator works closely with the query-dependent index that rules out many invalid edges that cannot appear in any of the results of the given query. Therefore, it is expected to give more accurate estimation than working on the original graph or the global statistics of \(G\). The method estimates the cardinality based on Equations 6 and 7, which calculates the number of walks from \(s\) to \(t\) (\(\delta_W\)). As a result, if the gap between the number of walks (\(\delta_W\)) and the number of paths (\(\delta_P\)) is small, then our method can give an accurate estimation. Otherwise, the method can introduce some errors. Our extensive experiment results in the appendix show that the estimation method works well in practice.

7 EXPERIMENTS
7.1 Experimental Setup
All experiments are conducted in a Linux machine equipped with two Intel Xeon E5-2660 v2 CPUs and 64GB RAM. The graph is first loaded entirely into the main memory from the disk, and we focus on the scenario of queries on in-memory graphs. Thus we exclude the time on disk I/O.

**Datasets.** Table 2 lists the details of the 15 real-world graphs, most of which are used in previous work [29]. These graphs are from a variety of categories such as social networks, web graphs and biology graphs. The number of vertices ranges from thousands to tens of millions, and the number of edges varies from hundreds of thousands to billions. We use \(tm\), a graph with billions of edges, to evaluate the scalability of our algorithm.

**Queries.** For each graph \(G\), we generate four query sets each of which contains 1,000 queries. Different query sets vary in the number of query results as well as search space in enumeration. Specifically, we divide \(V(G)\) into two disjoint sets \(V^t\) and \(V^s\) based on the vertex degrees: (1) \(V^t\) is the set of vertices within top 10% in the descending order of their degrees; and (2) \(V^s\) is the remaining ones \(V(G)\), excluding \(V^t\). Then, we have four settings according to the locations of \(s\) and \(t\): \((V^t, V^s)\times(V^t, V^s)\). For each setting, we generate 1,000 queries by choosing \(s\) and \(t\) uniformly at random. We vary the hop-constraint \(k\) from 3 to 8 in our experiments. To guarantee that there exists at least one result, we ensure that the distance between \(s\) and \(t\) is no larger than 3. We add this constraint because the query is terminated by a breadth-first search if these is no result, which makes the enumeration problem trivial. The query set where both \(s\) and \(t\) belong to \(V^s\) is generally more challenging than the other three query sets because there are more paths between vertices with large degrees. Therefore, we report the experiment results on the query set where \((s, t)\in V^s\) and \(k=6\) by default.

**Metrics.** For each algorithm, we measure the query time, throughput and response time to process a query. The query time is the elapsed time from the beginning of a query to its end. The response time is the elapsed time from the beginning of a query to finding the first 1000 results. Both of them are measured in milliseconds (ms). The throughput is the number of results found per second. We report the arithmetic mean of these metrics on a query set unless otherwise specified. To complete our experiments in a reasonable time, we set the time limit for a query as two minutes (1.2 \(\times 10^5\) ms). If the query cannot be completed within the time limit, we terminate it and set its query time as two minutes. The throughput is calculated based on the number of results found when the query is terminated.

**Comparisons.** We study the following algorithms in comparison with PathEnum. We obtain the source code of BC-DFS and BC-JOIN from their original authors [29]. All the competing algorithms are implemented in C++. We compile the code with g++ 7.3.1 with -O3 enabled.

---

Table 2: Properties of real-world graphs.

| Name   | Dataset          | \(|V|\) | \(|E|\) | \(d_{avg}\) | Type          |
|--------|------------------|-------|-------|----------|---------------|
| up     | US Patents       | 4M    | 17M   | 8.8      | Citation      |
| db     | DBpedia          | 4M    | 14M   | 6.5      | Miscellaneous |
| gg     | Web-google      | 876K  | 5M    | 11.1     | Web           |
| st     | Web-standford    | 282K  | 2.3M  | 16.4     | Web           |
| tw     | Twitter-social   | 465K  | 835K  | 3.6      | Miscellaneous |
| bk     | Baidu-baike      | 416K  | 3M    | 15.8     | Web           |
| tr     | Wiki-trust       | 139K  | 740K  | 10.7     | Interaction   |
| ep     | Soc-Epinson      | 75K   | 508K  | 13.4     | Social        |
| uk     | Web-uk-2005      | 121K  | 334K  | 181.2    | Web           |
| wt     | WikiTalk         | 2M    | 5M    | 4.2      | Social        |
| sl     | Soc-Slashdot0922 | 82K   | 948K  | 21.2     | Social        |
| lj     | LiveJournal      | 5M    | 69M   | 28.3     | Social        |
| da     | Rec-dating       | 169K  | 17M   | 205.7    | Recommendation|
| ye     | Bio-grid-yeast   | 6K    | 314K  | 104.5    | Biological    |
| tm     | Twitter-mpi      | 52M   | 1.96B | 74.7     | Miscellaneous |

---

1http://snap.stanford.edu/data/
2http://networkrepository.com/networks.php
Table 3: Overall comparison of competing algorithms on different graphs. The star symbol besides the query time denotes that the algorithm runs out of time on > 20% queries. The algorithm performing the best on each graph is marked in bold.

| Dataset | BC-DFS (Time) | BC-JOIN (Time) | IDX-DFS (Time) | IDX-JOIN (Time) | PathEnum (Time) |
|---------|---------------|----------------|----------------|-----------------|-----------------|
| up      | 5.75e+0       | 4.26e+0        | 2.73e-1        | 2.41e-1         | 2.28e-1         |
| db      | 1.13e+1       | 1.05e+1        | 7.97e-1        | 4.06e+1         | 6.22e-1         |
| gg      | 1.83e+2       | 6.64e+1        | 9.67e-1        | 8.08e+0         | 1.16e+0         |
| st      | 3.67e+3       | 4.05e+2        | 4.44e-0        | 4.95e+0         | 3.28e+0         |
| tw      | 3.35e+2       | 4.16e+2        | 1.72e+0        | 2.96e+0         | 1.78e+0         |
| bk      | 7.08e+3       | 2.63e+3        | 9.19e+1        | 7.57e+1         | 9.29e+1         |
| tr      | 9.88e+4*      | 1.17e+4        | 2.39e+2        | 1.00e+2         | 9.83e+1         |
| ep      | 1.66e+5*      | 2.34e+4        | 6.55e+2        | 2.78e+2         | 3.79e+2         |
| uk      | 4.67e+4*      | 4.47e+4*       | 3.88e+3        | 4.65e+3         | 3.84e+3         |
| wt      | 1.05e+5*      | 3.23e+4        | 1.70e+3        | 5.14e+2         | 4.79e+2         |
| sl      | 1.20e+5*      | 6.10e+4*       | 2.76e+3        | 7.51e+2         | 7.18e+2         |
| lj      | 1.20e+5*      | 1.20e+5*       | 8.50e+2        | 6.39e+2         | 4.99e+2         |
| da      | 1.20e+5*      | 1.20e+5*       | 1.26e+4        | 3.84e+3         | 3.32e+3         |
| ye      | 1.20e+5*      | 1.20e+5*       | 7.88e+4*       | 1.18e+5*        | 6.46e+4         |
| #Results-BC-DFS| 7.57e+1        | 1.72e+0        | 2.96e+0        | 7.57e+1         | 9.29e+1         |
| #Results-IDX-DFS| 5.26e+1        | 2.78e+2        | 3.79e+2        | 5.14e+2         | 4.79e+2         |
| #Invalid-BC-DFS| 1.00e+2        | 1.00e+2        | 1.00e+2        | 1.00e+2         | 1.00e+2         |
| #Invalid-IDX-DFS| 3.84e+3        | 3.84e+3        | 3.84e+3        | 3.84e+3         | 3.84e+3         |
| #Edges-BC-DFS| 5.14e+2        | 4.79e+2        | 4.79e+2        | 4.79e+2         | 4.79e+2         |
| #Edges-IDX-DFS| 1.18e+5*       | 6.46e+4         | 6.46e+4         | 6.46e+4         | 6.46e+4         |

- BC-DFS [29]: The state-of-the-art polynomial delay method.
- BC-JOIN [29]: A join-oriented algorithm based on BC-DFS.
- IDX-DFS: The proposed depth-first search method.
- IDX-JOIN: The proposed join method on the index.
- PathEnum: IDX-JOIN achieves more than two times speedup over IDX-DFS on graphs with long responses. As shown in the table, the number of queries running out of time on ye.

For example, PathEnum reduces both the query time and the number of queries running out of time on ye. The results prove the effectiveness of our query optimizer. In a small number of cases, PathEnum can run slightly slower than IDX-DFS or IDX-JOIN. This is because our cost model only considers the impact of the number of partial results, whereas some other factors (e.g., the overhead of materialization and the cost of checking whether a vertex belongs to M) can affect the practical performance.

In the following, we select ep and gg as representative graphs to demonstrate experiment results. ep takes long query time, while gg takes short query time.

**Detailed Metrics.** To compare the pruning techniques, we examine the detailed metrics of BC-DFS and IDX-DFS, which includes (1) the number of invalid partial results (#Invalid), which are the partial results that do not appear in any path in the graph; (2) the number of edges accessed (#Edges) during the enumeration; and (3) the number of results reported (#Results). Figure 6 presents the experiment results. We make the following observations. First, the number of edges accessed by BC-DFS is around 100 times as many as that by IDX-DFS, which shows the effectiveness of our index. The gap narrows on ep with k varied from 6 to 8 because BC-DFS runs out of time on most queries and finds fewer results than IDX-DFS. Second, the number of invalid partial results generated by the studied approaches is very close, which indicates that the pruning techniques in BC-DFS provide limited pruning power compared with simply using the distance to 0 in our method. Moreover, the number of invalid partial results accounts for a small portion of results. This implies that the benefit of adopting complex pruning techniques during the enumeration to reducing the invalid partial results is limited.

Impact of cost optimizer. PathEnum generally outperforms both IDX-DFS and IDX-JOIN, especially on graphs with long query time.

7.2 Comparison with Existing Algorithms

**Overall Comparison.** Table 3 gives an overall comparison of competing algorithms on different graphs. We only report the response time of BC-DFS and IDX-DFS because the join-based methods have to obtain the results of each sub-query before computing the full results, which have a long response time. As shown in the table, the query time on different graphs varies greatly, which ranges from less than one millisecond to tens of seconds.

PathEnum vs. BC-DFS/IDX-JOIN. Our algorithms significantly outperform counterparts on all graphs, especially those with long query time. For example, IDX-DFS runs 61X faster than BC-DFS on wt in terms of query time and achieves 6547X speedup in terms of throughput. The performance gap is different in terms of query time and throughput because BC-DFS runs out of time on a number of queries. Additionally, we can see that BC-DFS runs out of time on more than 20% queries on a number of graphs, while our algorithms complete most of queries. IDX-DFS spends less than 1 second to find 1000 results on all graphs, and achieves more than one order of magnitude speedup over BC-DFS in terms of response time.

IDX-DFS vs. IDX-JOIN. IDX-DFS outperforms IDX-JOIN on graphs with short queries, but generally runs slower on graphs with long queries. For example, IDX-DFS achieves up to two orders of magnitude speedup over IDX-JOIN on up because there is a small number of results (i.e., the search space is small) and the time spent on generating join orders can dominate the query time. In contrast, IDX-JOIN achieves more than two times speedup over IDX-DFS on tr and ep. The results demonstrate that optimizing the join order can significantly accelerate the query.

Impact of cost optimizer. PathEnum generally outperforms both IDX-DFS and IDX-JOIN, especially on graphs with long query time.
Query Time Breakdown. Figure 7 presents the query time breakdown of BC-DFS and IDX-DFS on $ep$ and $gg$. The preprocessing time is the time on building index, while the enumeration time is that on enumerating results. As shown in the figure, the preprocessing dominates the query time when $k$ is small. IDX-DFS runs much faster than BC-DFS on both the preprocessing and enumeration (Note that y-axis is log-scale and we terminate a query when it runs out of time). The elapsed time of BC-DFS and IDX-DFS is close on $ep$ when $k = 8$ because a number of queries run out of time.

![Figure 7: Query time breakdown of BC-DFS (without hatches) and IDX-DFS (with hatches) with $k$ varied.](image)

Query Time Distribution. Table 4 counts the percentage of queries that can be completed within 60 seconds ($<60s$) and that run out of time ($>120s$). Others can be finished between 60 seconds and 120 seconds. We can see that the number of queries running out of time increases with $k$ varied from 3 to 8 on $ep$. IDX-DFS significantly outperforms BC-DFS, especially when $k$ is large. For example, IDX-DFS completes 23.1% queries within 60 seconds, whereas BC-DFS only completes 0.1% queries. Furthermore, IDX-DFS completes all queries on $gg$ within 60 seconds.

![Figure 8: Comparison of 99.9% latency with $k$ varied.](image)

### Table 4: Query time distribution on $ep$ and $gg$.

| $k$ | $<60s$ | $>120s$ | $<60s$ | $>120s$ | $<60s$ | $>120s$ |
|-----|---------|---------|--------|---------|--------|---------|
| 3   | 1.00    | 0.00    | 1.00   | 0.00    | 1.00   | 0.00    |
| 4   | 1.00    | 0.00    | 1.00   | 0.00    | 1.00   | 0.00    |
| 5   | 0.959   | 0.016   | 1.00   | 0.00    | 1.00   | 0.00    |
| 6   | 0.130   | 0.813   | 1.00   | 0.00    | 1.00   | 0.00    |
| 7   | 0.003   | 0.997   | 0.877  | 0.066   | 0.994  | 0.003   |
| 8   | 0.001   | 0.999   | 0.231  | 0.674   | 0.749  | 0.138   |

Performance on Outlier Queries (queries running out of time). Moreover, we evaluate the performance of BC-DFS and IDX-DFS on short running queries ($<60s$) and long running queries ($>120s$), respectively. Table 5 presents throughput and response time on $ep$ with $k = 8$. IDX-DFS runs much faster than BC-DFS in terms of both throughput and response time. The response time of IDX-DFS on short and long running queries is very close and the value is small. Moreover, IDX-DFS has a high throughput on both short and long running queries, which indicates that the enumeration is efficient. Therefore, IDX-DFS cannot complete the outlier queries because these queries have a large number of results.

### Table 5: Performance for queries with different query time on $ep$ with $k = 8$.

| Method   | Throughput ($<60s$) | Throughput ($>120s$) | Response Time (ms) ($<60s$) | Response Time (ms) ($>120s$) |
|----------|---------------------|----------------------|-----------------------------|-----------------------------|
| BC-DFS   | 1.52e+04            | 8.65e+03             | 6.11e+02                    | 3.03e+03                    |
| IDX-DFS  | 7.83e+06            | 5.13e+07             | 2.12e+01                    | 2.52e+01                    |

Performance on Dynamic Graphs. We compare the performance of BC-DFS and IDX-DFS on dynamic graphs. Following experiments in [29], we randomly select $10\%$ edges of $ep$ and $gg$ as updates and keep subgraphs on remaining edges as initial graphs. For each selected edge $e(a, v')$, we set $a'$ and $v$ as $s$ and $t$, respectively, and enumerate the hop-constrained paths. As the index is built for each query online, our method can directly process dynamic graphs. We examine the 99.9% latency of BC-DFS and IDX-DFS in terms of the response time. Figure 8 presents the results on $ep$ and $gg$ with $k$ varied. As show in the figure, IDX-DFS significantly outperforms BC-DFS. The 99.9% latency of IDX-DFS on $ep$ with $k$ varied from 3 to 7 is within 0.1s. The 99.9% latency of IDX-DFS on $gg$ is less than 0.1s.

![Figure 9: Spectrum analysis of join plan optimization.](image)

Figure 9 presents the results of a query with $k = 6$ on $ep$ and $gg$. “DFS” and “JOIN” denotes the enumeration time of Algorithms 4 and 6, respectively. “Optimization” represents the time spent on optimizing the join order (Algorithm 5). “PathEnum” denotes the sum of the enumeration time and the query optimization time of PathEnum. Each blue point denotes the time on enumerating all results based on the index with a join plan. In Figure 9a, the optimization time is much shorter than the enumeration time and the
optimal plan is a bushy tree. In Figure 9b, the optimization time is longer than the enumeration time. PathEnum takes shorter time than the optimization because the preliminary estimator decides to use IDX-DFS directly. So our join optimizer is effective. Nevertheless, the query optimizer can be further improved by considering a larger plan space because our method considers only one plan with the left-deep tree (i.e., the order from s to t) and the optimal plan can fall outside of our plan space.

Factors on Query Efficiency. We examine the impact of the index size and the number of results on the enumeration time, respectively. The index size is measured by the number of edges in the index. As the enumeration time, the index size and the number of results vary greatly on different queries, we perform the linear regression analysis on the logarithm values of these metrics. Figures 10 and 11 present the results of IDX-DFS on ep and gg with $k = 6$. A blue point represents the result of a query and the red line denotes the underlying relationship obtained by the linear regression model. The enumeration time increases with the index size and #results increasing. Moreover, the enumeration time has a closer relationship with #results than the index size.

![Figure 10: Impact of index size on enumeration time.](image)

![Figure 11: Impact of #results on enumeration time.](image)

Average and Maximum Number of Results. Moreover, we examine the average and maximum number of results reported on ep and gg with $k$ varied. Table 6 presents the experiment results. The star symbol denotes that we cannot enumerate all results within 120 seconds.

Table 6: The average and maximum number of results reported on ep and gg. The star symbol denotes that we cannot enumerate all results within 120 seconds.

| $k$ | 3   | 4   | 5   | 6   | 7   | 8   |
|-----|-----|-----|-----|-----|-----|-----|
| ep  | avg | 5.06e+1 | 5.16e+3 | 5.34e+5 | 5.54e+7 | 3.08e+9 | 1.24e+10 |
|     | max | 3.30e+3 | 3.48e+5 | 3.64e+7 | 3.79e+9 | 2.74e+10 | 2.28e+10 |
| gg  | avg | 7.58e+0 | 9.33e+1 | 1.22e+3 | 1.71e+4 | 2.59e+5 | 4.03e+6 |
|     | max | 3.78e+2 | 6.69e+3 | 1.13e+5 | 1.81e+6 | 3.40e+7 | 7.24e+8 |

Because there are more results on ep than gg as shown in Table 6. In summary, the index takes small space, while the partial results of IDX-JOIN can consume a large amount of space due to the large number of results.

Table 7: Maximum memory consumption (MB) on ep and gg.

|       |   | 3   | 4   | 5   | 6   | 7   | 8   |
|-------|---|-----|-----|-----|-----|-----|-----|
| Index | ep | 0.15 | 1.68 | 3.28 | 4.60 | 5.45 | 5.91 |
|       | gg | 0.01 | 0.10 | 0.11 | 0.21 | 0.44 | 0.63 |
| Partial | ep | 0.03 | 0.66 | 26.51 | 138.32 | 4561.59 | 21479.32 |
| Results | gg | 0.01 | 0.02 | 0.37 | 1.05 | 17.80 | 55.70 |

Supplement Experiments. The appendix presents more experiment results including the comparison of throughput, query time and response time with $k$ varied, the cumulative distribution function of query time, the time efficiency of individual techniques (e.g., index construction) with $k$ varied, and the effectiveness of cardinality estimators.

7.4 Scalability Evaluation

We evaluate the scalability of IDX-DFS and IDX-JOIN with tm that has around two billion edges. Figure 12 presents the execution time of each individual technique and the throughput with $k$ varied from 3 to 6. IDX-JOIN runs out of memory when $k = 6$. Therefore, we omit its results on this case. “Index construction” denotes the time spent on building the index (Algorithm 3). Additionally, we report the time of computing the distance of each vertex to s, t, which is denoted by BFS. BFS is included in Index construction. As shown in Figure 12a, Algorithm 3 spends tens of seconds on the index construction, which is dominated by BFS. The time spent on building the index and generating join orders is more than that on enumerating results when $k$ varied from 3 to 4. Despite the long preprocessing time, the throughput of both IDX-DFS and IDX-JOIN is up to $10^7$ when $k = 5$, which demonstrates the efficiency of the index-based enumeration.

![Figure 12: Scalability evaluation on tm with $k$ varied.](image)

7.5 Discussions

Although PathEnum significantly accelerates HcPE queries, there still leaves interesting future work. First, our join optimizer can be
We design a light-weight index, and two index-based approaches per query basis. We conduct extensive experiments with a variety for efficient enumerations. We further develop a query optimizer addressing real-time requirements from many on-line applications. In this paper, we study the hop-constrained

8 CONCLUSIONS

In this paper, we study the hop-constrained $s$-$t$ path enumeration problem, and propose PathEnum, an efficient algorithm towards addressing real-time requirements from many on-line applications. We design a light-weight index, and two index-based approaches for efficient enumerations. We further develop a query optimizer to optimize the join order and decide which approach to use at per query basis. We conduct extensive experiments with a variety of real-world graphs, and show that PathEnum achieves orders of magnitude speedup over the state-of-the-art approaches.

REFERENCES

[1] Christopher R Aberg, Andrew Lamb, Susan Tu, Andres Nietzli, Kunle Olukotun, and Christopher Re. 2017. Emptyheaded: A relational engine for graph processing. ACM Transactions on Database Systems (TODS) 42, 4 (2017), 1–44.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. Foundations of databases. Vol. 8. Addison-Wesley Reading.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. 349–360.

[4] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. 2015. A new approach to incremental cycle detection and related problems. ACM Transactions on Algorithms (TALG) 12, 2 (2015), 1–22.

[5] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2509–2521.

[6] Etienne Burmele, Rui Ferreira, Roberto Grossi, Andrea Marin, Nadia Pisanti, Romeo Rizzi, and GustavoSacomoto. 2013. Optimal listing of cycles and $s$-$t$-paths in undirected graphs. In Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete Algorithms. SIAM, 1884–1896.

[7] Katerina Bohnová, Luca Fiallger, Matúš Mihaláik, Tobias Pröger, GustavoSaco-moto, and Marie-France Sagot. 2018. Computing and listing $s$-$t$-paths in public transportation networks. Theory of Computing Systems 62, 3 (2018), 600–621.

[8] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Jian Pei. 2015. Efficiently computing top-$k$ shortest path join. In EDBT 2015-18th International Conference on Extending Database Technology. Proceedings.

[9] Jiefeng Cheng and Jeffrey Xu Yu. 2009. On-line exact shortest distance query processing. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. 481–492.

[10] 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.

[11] David Epstein. 1998. Finding the $k$ shortest paths. SIAM Journal on computing 28, 3 (1999), 652–673.

[12] Financial Action Task Force. 2013. FATF Report. Money Laundering and Terrorist Financing Vulnerabilities of Legal Professionals. Paris: FATF (2013).

[13] Jun Gao, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. 2010. Fast-top-$k$ simple shortest path discovery in graphs. In Proceedings of the 19th ACM international conference on Information and knowledge management. 509–518.

[14] Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Efficient algorithms for listing $k$ disjoint $s$-paths in graphs. In Latin American Symposium on Theoretical Informatics. Springer, 544–557.

[15] Bernhard Haeseler, Telikepalli Kivitha, Rogers Mathew, Siddhartha Sen, and Robert E Tarjan. 2012. Incremental cycle detection, topological ordering, and strong component maintenance. ACM Transactions on Algorithms (TALG) 8, 2 (2012), 1–33.

[16] Czeslaw Jedrzejek, J B, and M Falkowski. 2009. Graph mining for detection of a large class of financial crimes. In 17th International Conference on Conceptual Structures, Moscow, Russia d. 4.

[17] Ruoming Jin, Zhen Peng, Wendell Wu, Feodor Dragan, Gagan Agrawal, and Bin Ren. 2019. Pruned Landmark Labeling Meets Vertex Centric Computation: A Surprisingly Happy Marriage? arXiv preprint arXiv:1906.12018 (2019).

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

[19] David S Johnson, Mihalin Yannakakis, and Christos H Papadimitriou. 1988. On generating all maximal independent sets. Inform. Process. Lett. 27, 3 (1988), 119–123.

[20] Kyoungmin Kim, In Seo, Woo-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Huyungyu Shin, and Geonhwa Jeong. 2018. Turboldx: A fast continuous subgraph matching system for streaming graph data. In Proceedings of the 2018 International Conference on Management of Data. 411–426.

[21] Rohit Kumar and Toon Calders. 2018. 2SCENT: an efficient algorithm for enumerating all simple temporal cycles. Proceedings of the VLDB Endowment 11, 11 (2018), 1441–1455.

[22] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhenqin Lai, Ran Wang, Kongzhao Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. Proceedings of the VLDB Endowment 12, 10 (2019), 1099–1112.

[23] Feifei Li, Bin Wu, Ke Yi, and Zhiyue Zhao. 2016. Wander join: Online aggregation via random walks. In Proceedings of the 2016 International Conference on Management of Data. 615–629.

[24] Xianglei Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hosi, He Huang, and Xueqi Cheng. 2020. FlowScope: Spouting Money Laundering Based on Graphs. In AAAI. 4731–4738.

[25] Ernesto QV Martins and Marta MB Pascoal. 2003. A new implementation of Yen’s ranking loopless paths algorithm. Quarterly Journal of the Belgian, French and Italian Operations Research Societies 1, 2 (2003), 121–133.

[26] Amine Mhedhbi and Semih Sahinoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. arXiv preprint arXiv:1903.02076 (2019).

[27] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. 2017. Compiling graph substructures into sentential decision diagrams. In Proceedings of the Thirty-First AAAI conference on artificial intelligence. 1213–1221.

[28] Yeonwoo Park, Seongyoon Ko, Soura S Bhowmick, Kyoungmin Kim, Kjaje Hong, and Woo-Shin Han. 2020. GRE-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 1099–1114.

[29] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained $s$ $t$ path enumeration. Proceedings of the VLDB Endowment 13, 4 (2019), 463–476.

[30] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Arístides Gionis. 2009. Fast shortest path distance estimation in large networks. In Proceedings of the 18th ACM conference on Information and knowledge management. 867–876.

[31] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. 2012. Approximate shortest distance computing: A query-dependent local landmark scheme. IEEE Transactions on Knowledge and Data Engineering 26, 1 (2012), 55–68.

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

[33] Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. 2014. Efficiently listing bounded length $s$-$t$-paths. In International Workshop on Combinatorial Algorithms. Springer, 318–329.

[34] Baobao Shi and Tim Weninger. 2016. Discriminative predicate path mining for fact checking in knowledge graphs. Knowledge-based systems 104 (2016), 123–133.

[35] Prashant Shiralkar, Alessandro Flammini, Filippo Menczer, and Giovanni Luca Cimabbage. 2017. Finding streams in knowledge graphs to support fact checking. In 2017 IEEE International Conference on Data Mining (ICDM). IEEE, 859–864.
The theorem is proved. Theorem 3.1 as follows.

Thus, the lemma is proved.

Hongwei Wang, Hongyu Ren, and Jure Leskovec. 2020. Entity Context and Relational Paths on a graph for counting and enumeration. In Advanced Methodologies for Bayesian Networks. 129–140.

Norihito Yasuda, Teruji Sugaya, and Shin-Ichi Minato. 2017. Fast compilation of st paths on a graph for counting and enumeration. In Advanced Methodologies for Bayesian Networks. 129–140.

Jin Y. Yen. 1971. Finding the k shortest loopless paths in a network. Management Science 17, 11 (1971), 712–716.

A CORRECTNESS OF JOIN-BASED MODEL

In this section, we prove that the join-based model in Section 3.1 is correct. Given a graph G and a HcPE query q(s, t, k), suppose that the relations of Q is generated based on the method in Section 3.1. Given a tuple r ∈ Q, r[1] represents the vertex at position i, and l[i] denotes the first position that t appears in r. Let r[0 : i] be the vertices from positions 0 to i. We first prove that Q satisfies the following lemma.

Lemma A.1. Given a tuple r ∈ Q, r[0 : l[i]] is a walk from s to t, and each vertex v ∈ r[l[i] : k] is t.

Proof. Based on the first property of relations, r[0] and r[k] must be s and t, respectively. Given 0 < i < l, (r[i − 1], r[i]) is an edge in E(G) because it is a tuple belonging to R(i). Thus, r[0 : l[i]] is a walk from s to t. Next, we prove that each vertex v ∈ r[l[i] : k] is t by contradiction. Without loss of generality, assume that r[i] is the first vertex not equal to t where l[i] < i < k. Then, (t, r[i]) belongs to R(i). However, the second property guarantees that there is no tuple starting from t except (t, t), which contradicts the assumption. Thus, the lemma is proved.

Moreover, the following lemma holds.

Lemma A.2. Given a walk W ∈ W(s, t, k, G), there is a tuple r ∈ Q such that W is equal to r[0 : l[i]].

Proof. Given W ∈ W(s, t, k, G), we first construct a tuple r as follows: r[0 : l[i]] = W and set each vertex in r[l[i] : k] as t. Next, we prove that r belongs to Q. Given 1 ≤ i < l, (r[i − 1], r[i]) exists in R(i) according to the generation method of relations. Let Q’ be R_1 ⊔ ⋯ ⊔ R_l. Then, r[0 : l[i]] belongs to Q’. As (t, t) ∈ R_l where l[i] < i < k, r[0 : l[i]] will be extended by adding t when performing the join operation on Q’ and the remaining relations. Therefore, r appears in the final results. Moreover, as the join operation satisfies the commutative and associative laws, r belongs to Q regardless of the join order. Thus, the lemma is proved.

Based on Lemmas A.1 and A.2, we can prove the correctness of Theorem 3.1 as follows.

Proof. Q has no duplicate results because tuples in each relation of Q are distinct. According to Lemmas A.1 and A.2, Q contains all walks in W(s, t, k, G). Therefore, we can obtain all paths from s to t by eliminating results in Q contain duplicate vertices except t. The theorem is proved.

B PRUNING POWER COMPARISON

We compare the pruning power of Algorithm 2 with that of Algorithm 3 in this section. Let R_i(u_{i−1}, u_i) be the relation generated by Algorithm 2. Given 1 ≤ i < k, R_i(u_{i−1} : v, u_i) represents the neighbors of v in R_i and C(u_{i−1}) denotes all values of u_{i−1} in R_i, i.e., [v, v’) ∈ R_i]. Because v ∈ C(u_{i−1}), there exists a walk W ∈ W(s, t, k, G) such that W[1 : j] = v according to Proposition 4.2. Therefore, v belongs to X in I according to Proposition 4.3. Next, we prove that given v ∈ C(u_{i−1}) where v ≠ t, R_i(u_{i−1} : v, u_i) is equal to I_i(v, k − i) by contradiction.

Assume that v’ ∈ R_i(u_{i−1} : v, u_i) but v’ ∈ I_i(v, k − i). Then, S(v’, t|G − {s}) ≤ k − i because there exists W ∈ W(s, t, k, G) such that W[1 : j] = v according to Proposition 4.2. Based on Algorithm 3, if v’ ∈ N(v) and S(v’, t|G − {s}) < k − i, then v’ belongs to I_i(v, k − i), which contradicts the assumption. Therefore, R_i(u_{i−1} : v, u_i) ⊆ I_i(v, k − i) by contradiction. Now, assume that v’ ∈ I_i(v, k − i) but v’ ∉ R_i(u_{i−1} : v, u_i). Therefore, there is a walk W from v’ to t such that L(W) < k − i because S(v’, t|G − {s}) < k − i according to Algorithm 3. Moreover, there is a walk W’ from s to each such that W[1 : j] = v because v ∈ C(u_{i−1}). Then, we construct a walk from s to t by concatenating W’ and W. According to Proposition 4.2, (v, v’) ∈ M, which contradicts the assumption. So we have I_i(v, k − i) ⊆ R_i(u_{i−1} : v, u_i). Based on the analysis, we can see that our index provides competitive pruning power with relations generated by Algorithm 2.

C CORRECTNESS OF OUR ALGORITHMS

In the following, we first prove the correctness of Algorithms 4.

Proposition C.1. Algorithm 4 finds all k hop-constrained paths P(s, t, k, G) from s to t in G.

Proof. We first prove that M emitted by the algorithm is a path P ∈ P(s, t, k, G). Lines 1 and 4 ensure that M begins with s, while ends with t. The check at Line 7 keeps that M contains no duplicate vertices. Based on the construction method of I, Line 6 guarantees that L(M) ≤ k and there is an edge in E(G) between any two successive vertices in M. So M reported by Algorithm 4 is a path P ∈ P(s, t, k, G). Additionally, Algorithm 4 does not report duplicate results because I_i(s, v) returns a set of vertices.

Next, we show that given any path P ∈ P(s, t, k, G), Algorithm 4 can find it. We prove this by induction. Without loss of generality, suppose that P = (v_0 = s, v_1, ..., v_j = t) where 1 ≤ j ≤ k. Initially, M = {v_0} holds (Line 1). Assume that M = {v_0, v_1, ..., v_i} where 1 < i < j − 1 is constructed by the algorithm. We prove that the algorithm can generate M’ = (v_0, v_1, ..., v_i, v_{i+1}) from M. As v_{i+1} appears in P, S(v_{i+1}, t|G − {s}) < k − i − 1. Based on the construction method of I, v_{i+1} must belong to I_i(v_i, k − i − 1). Therefore, M’ can be generated from M by the for loop (Lines 6-7). Thus, the algorithm can find P, and the proposition is proved.

Algorithm 6 first evaluates Q[0 : l[i]] and Q[l[i] : k], respectively. Then, it joins them to find final results. The analysis in Section 3.1 guarantees the correctness of Algorithm 6.

Proposition C.2. Algorithm 6 finds all k hop-constrained paths P(s, t, k, G) from s to t in G.
Next, we prove Proposition 5.1, which shows that (1) Algorithm 4 without the check at Line 7 finds all $k$ hop-constrained walk $W(s,t,k,G)$ from $s$ to $t$ in $G$; and (2) Given $M \in \mathcal{M}_i$ that 0 $\leq i \leq k$, $M$ must appear in a walk $W \in W(s,t,k,G)$.

Proof. We can prove that Algorithm 4 after relaxation finds $W(s,t,k,G)$ with the same proof method of Proposition C.1. Therefore, we omit a detailed proof for brevity, and focus on the second part of the proposition. We prove this by contradiction.

Assume that there exist $M \in \mathcal{M}_i$ that does not appear in any walk in $W(s,t,k,G)$. Let $v$ and $v'$ represent the last two vertices in $M$. Then, $v' \neq t$ and $v' \in T_i(a,k-L(M-\langle v' \rangle)-1)$, which means $S(v',t|G-(s)) \leq k-L(M-\langle v' \rangle)-1$. Let $P'$ denote the shortest path from $v'$ to $t$ in $G-(s)$. We construct a walk $W$ by concatenating $M$ and $P'$. $L(W) = L(M) + S(v',t|G-(s)) < k$. Then, $W$ belongs to $W(s,t,k,G)$, which contradicts the assumption. Thus, the proposition is proved.

We can prove the correctness of Proposition 6.1 with the same method as that in Proposition 5.1. For brevity, we omit the details.

D COMPARISON WITH BC-DFS/BC-JOIN [29]

Comparison with BC-DFS. The barrier optimization technique in BC-DFS [29] extracts a subgraph $G'$ of $G$ based on the rule: if $v$ belongs to a result, then $S(s,v|G) + S(v,t|G) <= k$. Then, it enumerates all results on $G'$ with Algorithm 1. In contrast, we construct a light-weight index based on Proposition 4.3, which puts a constraint on which vertices appear at position $i$ of a result, and finds all results with the assistance of the index, which can efficiently get $v$ in $N(v)$ such that $S(v',t|G-(s)) <= b$. The index accelerates the enumeration because the index reduces the number of edges accessed at each step and eliminates the distance check (see the difference between Algorithms 1 and 4). Benefiting from the index, Algorithm 4 achieves a good time complexity $O(k \times \delta_W)$. The following is an example demonstrating the difference.

Example D.1. Given $G$ and $q$ in Figure 1, $G'$ contains all vertices except $v_7$ based on the barrier optimization technique [29]. Suppose that $M = (s,v_0,v_1)$. Algorithm 1 loops over $v_2$ and $v_3$ and checks their distances to $t$, whereas Algorithm 4 directly gets $v_2$ from the index and continues the search.

Both Algorithms 1 and 4 adopt the backtracking search to enumeration all results. The exploration process can be viewed as conducting a depth-first search in a search tree. Suppose that the average cost of expanding a node in the search tree is $\alpha$ and the search tree has $\beta$ nodes. The cost of exploring the search tree is $T = \alpha \times \beta$. Existing algorithms [14, 29, 33] directly traverse on $G$ to enumerate results. Given the last vertex $v$ of a partial result $M$, they visit all neighbors $v'$ of $v$ in $G$, check whether $S(v',t|G-M) <= k-L(M)-1$ and dynamically update $S(v,t|G-M)$ at each step. In contrast, our algorithm explores the index constructed in a preprocessing step to find results. At each step, we only consider the neighbors $v'$ of $v$ such that $S(v',t|G-(s)) <= k-L(M)-1$ with the assistance of the index, but eliminate the explicit distance verification and complex filtering techniques. In other words, the difference between existing algorithms and ours is the trade-off between $\alpha$ and $\beta$. Exiting algorithms optimize the search by decreasing $\beta$ at the cost of increasing $\alpha$, while we make the $Search$ procedure simple and efficient to reduce $\alpha$, but can generate more invalid partial results. However, it is challenging to make a direct comparison of our algorithm and existing algorithms in terms of the time complexity $O(k \times \delta_W)$ versus $O(k \times |E(G)| \times \delta_P)$ because the value of $\delta_P$ depends on each query instance. Instead, we conduct extensive experiments with a variety of real-world graphs to compare them (see Figure 6). Our experiment results show that our method significantly outperforms BC-DFS.

Comparison with BC-JOIN. Join is a common methodology for graph queries [1, 26]. The key factors leading to the performance differences are strategies reducing partial results, e.g., invalid elements filtering and join order optimization. BC-JOIN, which is a join-oriented algorithm proposed in [29], first computes the set $V$ of vertices appearing in the middle position $([\frac{k}{2}]$) of paths in $\mathcal{P}(s,t,k,G)$. Then, it computes the paths from $s$ to $v \in V$, and that from $v \in V$ to $t$ with Algorithm 1. Finally, it joins the intermediate results to find the final results. In contrast, our algorithm first adopts a light-weight index to prune invalid candidates and then performs the join on the basis of the index. Moreover, we design a cost-based query optimizer with the preliminary and full-fledged estimators to process queries with variant execution time.
equal to $\bigoplus_{e \in P} a(e)$. The constraints on accumulative values require that the accumulative value $\beta$ of $P$ satisfies a user-defined boolean function $f_{\beta}(\cdot)$, i.e., $f_{\beta}(\beta)$ returns true in addition to the length constraint. For example, if we require that the sum of the transaction risky factors along the path is above a threshold [12], then we can define $\bigoplus$ as a plus operation.

Algorithm 7: Depth-First Search on Index with Constraints on Accumulative Values

Input: two distinct vertices $s, t$, hop constraint $k$, index $I$;
Output: all $k$ hop-constrained paths from $s$ to $t$ that satisfy constraints on accumulative values;
1 $M \leftarrow (s)$;
2 $\beta \leftarrow$ an initial value;
3 Search($t, k, M, I, \beta$);
4 Procedure Search($t, k, M, I, \beta$)
5 \hspace{1em} $\sigma \leftarrow$ the last vertex in $M$;
6 \hspace{1em} if $\sigma = t$ and $f_{\beta}(\beta)$ is true then emit($M$), return;
7 \hspace{1em} foreach $\alpha' \in I_{\sigma}(k, k - L(M) - 1)$ do
8 \hspace{2em} if $\alpha' \notin M$ then Search($t, k, M \cup \{\alpha'\}, I, \beta \oplus a(\alpha(\alpha, \alpha'))$);

Algorithm 7 extends the depth-first search on the index to support constraints on accumulative values. Line 2 assigns an initial value based on the binary operation $\bigoplus$. For example, if $\bigoplus$ is the sum operation, then we set $\beta$ to 0, while setting it to 1 if $\bigoplus$ is the multiply operation. If the accumulative value of $P$ satisfies the constraint $f_{\beta}(\cdot)$, then Line 6 outputs the path. When adding a new vertex to the partial result $M$, Line 8 updates the accumulative value with the binary operation. In some cases, we can check whether the accumulative value cannot lead to a valid solution and can be pruned when extending the partial results to reduce search space; for example, if the constraint is that the accumulative edge weight must be below a threshold, and all edge weights are nonnegative. However, if edge weights are negative, we cannot add the check when extending $M$ because the accumulative value is not monotonic.

We can extend the join on index with the same method as Algorithm 7 because the binary operation satisfies commutative law and associative law and the accumulative value is independent of the sequence of performing the binary operations. We omit the detail for brevity.

Constraints on a Sequence of Actions. $l(e)$ is the edge label, which represents an action. Given a sequence of actions, the constraint requires that the edge label sequence along a result path satisfies the given sequence. For example, if we require that the flow of transactions pass through at least two high-risk counties [12], then we can define a sequence schema.

We can model the given sequence of actions as an automata where nodes are states and edges define the transition relationship among states based on actions. The automata can be represented by a matrix $A$. Given a state $a$ and an action (i.e., label) $l(e)$, $A[a][l(e)]$ returns the next state. If the action is invalid for the given state, then $A[a][l(e)]$ returns null. Algorithm 8 presents the depth-first search on index with constraints on sequences of actions, which is based on the automata $A$. Line 2 sets the initial state as the start state in the automata $A$. When trying to extend $M$ with a vertex, Line 8 retrieves the next state $a'$ based on $A$. If $a'$ is null, then skip the vertex. Otherwise, move to the state $a'$ and continue the search.

Line 6 outputs the result path if the vertex is $t$ and the transition reaches an end state in $A$.

Algorithm 8: Depth-First Search on Index with Constraints on Sequences of actions

Input: two distinct vertices $s, t$, hop constraint $k$, index $I$;
Output: all $k$ hop-constrained paths from $s$ to $t$ that satisfy constraints on sequences of Actions;
1 $M \leftarrow (s)$;
2 $a \leftarrow$ the start state in the automata $A$;
3 Search($t, k, M, I, a$);
4 Procedure Search($t, k, M, I, a$)
5 \hspace{1em} $\sigma \leftarrow$ the last vertex in $M$;
6 \hspace{1em} if $\sigma = t$ and $\alpha$ is an end state then emit($M$), return;
7 \hspace{1em} foreach $\alpha' \in I_{\sigma}(k, k - L(M) - 1)$ do
8 \hspace{2em} if $\alpha' \notin M$ and $\alpha' \neq$ null then
9 \hspace{3em} Search($t, k, M \cup \{\alpha'\}, I, \alpha'$);

As the join on index can evaluate the query with different orders, we use the automata to check whether a path $P$ can meet the constraint after $P$ is generated by the join on index method. Therefore, the depth-first search method can terminate the invalid search path at an earlier stage than the join method.

In summary, our method on the HcPE problem can be easily extended to support variant constraints. Adding the constraints can accelerate the query because the constraints can reduce the size of the search space.

F SUPPLEMENT EXPERIMENT RESULTS

Varying Hop Constraint $k$. Figure 13 shows the experiment results on query time with $k$ varied. When $k = 8$, BC-JOIN runs out of memory on ep due to the maintenance of a large amount of intermediate results. Thus, we omit the results of BC-JOIN on this case. As shown in the figures, PathEnum significantly outperforms BC-DFS and BC-JOIN. The preliminary estimation makes the overhead of PathEnum very small. Its time complexity is $O(k^2)$, and it takes less than 0.01 ms in experiments. Because there are a small number of results on gg with $k$ varied from 3 to 6, PathEnum directly invokes IDX-DFS after the preliminary estimation, the cost of which is negligible. Thus, PathEnum is close to IDX-DFS despite that IDX-DFS dominates IDX-JOIN. If the join order optimization is executed, the preliminary estimation ensures that the optimization time accounts for a small portion of query time. The benefit of the optimization can offset the overhead, and PathEnum can beat both IDX-DFS and IDX-JOIN.

Figure 13: Comparison of query time with $k$ varied.

Figure 14 shows the experiment results on throughput with $k$ varied. Although PathEnum runs out of time on most queries
against \( ep \) when \( k = 8 \), it has much higher throughput than BC-DFS and BC-JOIN. The throughput of our three algorithms increases on \( ep \) with \( k \) varied from 3 to 6, but keeps steady with \( k \) increasing from 6 to 8 because the time spent on building the index and optimizing join orders is neglected compared with the time spent on enumeration when \( k \) is large. Moreover, the results imply that the value of \( k \) has little impact on the enumeration speed of our algorithms. In contrast, the throughput of BC-DFS decreases with \( k \) varied from 5 to 8 because the increasing of \( k \) results in more overhead of dynamically updating the distance of vertices to \( t \).

**Individual Query Performance.** We demonstrate the cumulative distribution function of the query time in Figure 16 to examine the performance of completing individual query. The query time of different queries varies greatly. IDX-JOIN performs better than IDX-DFS on \( ep \), but worse on \( gg \). In contrast, PathEnum performs well on both of the two graphs. Nevertheless, our algorithms significantly outperform BC-DFS and BC-JOIN. For example, BC-DFS and BC-JOIN run out of time on more than 80% and 10% queries on \( ep \), respectively, while our algorithms complete all queries within around 10 seconds.

**Summary.** First, the light-weight index can significantly accelerate the enumeration by reducing the number of edges accessed and eliminating the distance check at each step. Moreover, as the index is query dependent, which prunes many invalid edges, it is expected to have a significant impact on query performance. However, the overhead of dynamically updating the distance to \( t \) is a disadvantage that needs to be addressed in future work.

**Time Efficiency of Individual Technique.** Figure 17 illustrates the execution time of each individual technique with \( k \) varied. "Index construction" denotes the time spent on building the index (Algorithm 3). "Optimization" represents the time spent on generating join orders (Algorithm 5). "DFS" and "JOIN" denotes the enumeration time of Algorithms 4 and 6, respectively. Therefore, the query time of IDX-DFS is the sum of Index construction and DFS, while the query time of IDX-JOIN is the sum of Index construction, Optimization and JOIN. Additionally, we report the time of computing the distance of each vertex to \( s, t \), which is denoted by BFS. BFS is included in Index construction. We omit the execution time of the preliminary cardinality estimator because its value is negligible (less than 0.01 ms).

As shown in the figure, BFS generally dominates the execution time of Algorithm 3. The cost of optimizing join orders can be greater than the enumeration for the short running queries. The DFS on the index runs faster than the join on the index when \( k \) is small, but slower when \( k \) is large. Nevertheless, the absolute value of Index construction and Optimization is very small, which demonstrates the efficiency of our index construction and query optimization.

More importantly, the benefits of these operations are far higher than the overhead, showing significant performance speedup even for short queries over existing approaches (BC-DFS/BC-Join).

**Cardinality Estimation.** We evaluate the preliminary and full-fledged cardinality estimators by comparing the number of results estimated to the actual value. Figure 18 illustrates the results with \( k \) varied. The gap between our estimation and the actual value widens with \( k \) varied from 3 to 8 because it is more challenging to estimate queries with long paths and the estimation error propagates with the increase of \( k \). We omit the result on \( ep \) when \( k = 8 \), because more than 50% of queries run out of time, and we cannot get their actual number of results of these queries.
to provide more accurate statistics for the query optimizer than that of the original graph. Second, a cost-based query optimizer is essential for reducing the cost of evaluating HcPE queries. Our query optimizer that closely works with the light-weight index is effective. Furthermore, the optimizer with cardinality estimation methods having different time complexities is necessary because the query time of different queries varies greatly even on the same graph. Third, our solution has a high enumeration speed (i.e., throughput) even on the graphs with billions of edges. As the index construction is efficient, our solution can directly handle dynamic graphs in general. However, when the graphs are very large, our method can have a long response time because it builds the index from the scratch. For example, the index construction takes tens of seconds on \( tm \) with billions of edges in Figure 12a. Fourth, the number of results has a significant impact on HcPE query efficiency. Some queries have a long running time because they have a huge number of results.