PathEnum: Towards Real-Time Hop-Constrained s-t Path Enumeration

Shixuan Sun
National University of Singapore
Singapore
sunsx@comp.nus.edu.sg

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

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

Bryan Hooi
National University of Singapore
Singapore
dcsbhk@nus.edu.sg

ABSTRACT

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.

CCS CONCEPTS

• **Information systems** → *Information retrieval query processing*.

KEYWORDS

s-t path enumeration; hop-constrained; light-weight index

ACM Reference Format:
Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained s-t Path Enumeration. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457290

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. Real-time HcPE is very important for interactiveness of many of those 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. 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 [41]. 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 interests. 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 a 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 $\mathcal{I}$ 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 $\mathcal{I}$. The time complexity of constructing $\mathcal{I}$ 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 results based on $\mathcal{I}$. The DFS-based method recursively extends the partial result by one vertex at a step to find all results, whereas the join-based method first cuts the query into two sub-queries, then evaluates them with the DFS-based method, and finally join the intermediate results of the two sub-queries. 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 significantly outperforms the state-of-the-art method [29] in terms of both query time and 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 invalid vertices. Then, we design two index-based approaches, and an effective join order optimization method to reduce the search space of the enumeration.
- 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 complete version of this paper [37]. Our source code is publicly available at *https://github.com/shixuansun/PathEnum.*

2 BACKGROUND AND RELATED WORK

2.1 Preliminaries

$G = (V, E)$ 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^r 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 $S(v, v'|G)$, 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 $(v_0, v_1, ..., v_l)$ such that $\forall 1 \leqslant i \leqslant l, e(v_{i-1}, v_i) \in E$. $|W|$ 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 ith vertex in W where $0 \leqslant i \leqslant |W| - 1$. Given two distinct vertices s and t, we define a walk from s to t in Definition 2.1. $\mathcal{W}(s, t, k, G)$ represents all walks W from s to t in G that satisfy $L(W) \leqslant k$. 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. $\mathcal{P}(s, t, k, G)$ denotes all paths P from s to t such that $L(P) \leqslant k$. Apparently, given $P \in \mathcal{P}(s, t, k, G)$, P belongs to $\mathcal{W}(s, t, k, G)$.

Definition 2.1. A walk from s to t is a walk W such that (1) $W[0] = s \wedge W[|W| - 1] = t$; 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 $\mathcal{P}(s, t, k, G)$. The query is denoted by $q(s, t, k)$. We assume that $k \geqslant 2$ in this paper.

2.2 State-of-the-art Approaches

Algorithm 1 illustrates a generic depth-first search based framework to find $\mathcal{P}(s, t, k, G)$. M stores a sequence of vertices, which initially contains s (Line 1). Line 5 emits M if the last vertex of M is t. Otherwise, Lines 6-8 loop over $N(v)$ to extend M. Particularly, $B(v')$ stores the distance from v' to t. We can initialize it by performing a breadth-first search from t along G^r. Line 7 checks (1) whether v' belongs to M; and (2) whether we can extend M by adding v' to generate a path satisfying the hop constraint. If v' passes the check, then we add v' to M and continue the search. Otherwise, we skip v'. Therefore, the *Search* procedure can be viewed as performing a depth-first search in a search tree where each node is a partial result M and each edge is the action of adding a vertex to M.

Existing approaches [14, 29, 33] adopt the same backtracking strategy as Algorithm 1, but introduce different pruning techniques to achieve polynomial delay. They update $B(v)$ for a vertex during the enumeration because a path contains no duplicate vertices and the update of M can break the shortest path from v to t. Peng et al. [29] designed a *barrier*-based method, which dynamically maintains the distance from each vertex to t. Initially, they set the barrier for each $v \in V(G)$ as $S(v, t|G)$. During the enumeration, if they find that a sub-tree rooted at a node in the search tree contains no result, then they will increase the barrier to avoid falling into the same sub-tree again. T-DFS [33] and T-DFS2 [14] are two theoretical works. They achieve polynomial delay by ensuring that each search branch in the search tree leads to a result. For example, before extending M by adding v' in Algorithm 1, T-DFS checks whether there is a shortest path from v' to t without vertices in M whose length is bounded by $k - L(M) - 1$. Although all the three algorithms achieve $O(k \times |E(G)|)$ polynomial delay, Peng et al. showed that their method runs much faster than T-DFS and T-DFS2 in practice because their pruning strategy incurs lower overhead [29]. HPI [32] builds an index maintaining paths between vertices with a high degree to enumerate hop-constrained cycles triggered by incoming edges in dynamic graphs. However, the index consumes a large amount of memory due to the exponential number of paths.

2.3 Other Related Work

***s-t* Path (or Cycle) Enumeration.** Another kind of algorithms [7, 27, 42] 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, 40]. 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.

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

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, 43]. 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 systems such as EmptyHeaded [1], GraphFlow [26] and RapidMatch [39] 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 catalogs [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, t 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**.

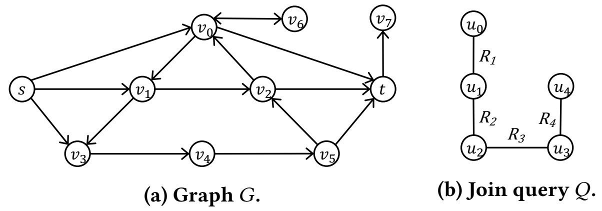
3.1 A Join-based Model

Although existing algorithms [14, 29, 33] provide comprehensive analysis to the HcPE problem in terms of the time complexity, we lack 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 $R(u,u') = \{(v,v')|e(v,v') \in E(G)\}$. 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 $1 < i < k$. For the ease of presentation, we use Q to represent the results of evaluating Q as well. To obtain $\mathcal{P}(s,t,k,G)$, 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 $R_k = \{(v,t)|e(v,t) \in E(G) \wedge v \neq s\}$;
(2) $R_i = \{(v,v')|e(v,v') \in E(G-\{s\}) \wedge v \neq t\}$ when $1 < i < k$;
(3) $R_i = R_i \cup \{(t,t)\}$ for $1 < i \leqslant k$.

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 $P \in \mathcal{P}(s,t,k,G)$ 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 complete version [37]

THEOREM 3.1. *Evaluating Q and eliminating tuples in Q having duplicate vertices except t results in $\mathcal{P}(s,t,k,G)$.*



(a) Graph G. **(b) Join query Q.**

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

Example 3.2. Given G in Figure 1a and a query $q(s,t,4)$, the join query Q is represented as a graph in Figure 1b where each edge is a relation and each node is an attribute. The relations of Q are shown in Figure 3a. The path (s,v_0,t) corresponds to the tuple (s,v_0,t,t,t) in Q. (s,v_0,v_6,v_0,t) belongs to Q as well. However, it is a walk from s to t, but not a path.

The cost function of evaluating Q is shown in Equation 1, which is the total number of intermediate results generated during the computation. If Q is a basic relation, the cost is the size of the relation as we need to read it. Otherwise, the cost is the sum of the cost of evaluating Q_1 and Q_2 and the number of results of Q where $Q = Q_1 \bowtie Q_2$. From the cost model, we can see that the cost of evaluating a query is closely related to (1) the number of edges involving in the enumeration; and (2) the join order (i.e., search order) of evaluating the query.

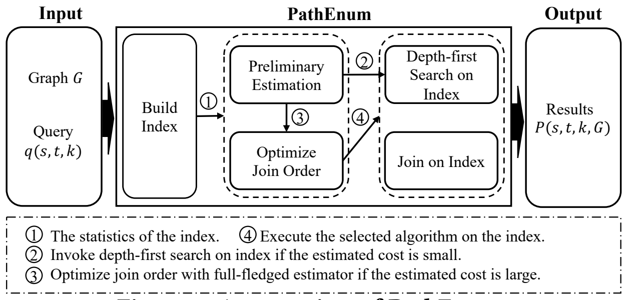

Input
PathEnum
Output
Graph G
Query q(s,t,k)
Build Index
Preliminary Estimation
Optimize Join Order
Depth-first Search on Index
Join on Index
Results P(s,t,k,G)
①
②
③
④
① The statistics of the index.
④ Execute the selected algorithm on the index.
② Invoke depth-first search on index if the estimated cost is small.
③ Optimize join order with full-fledged estimator if the estimated cost is large.

Figure 2: An overview of PathEnum.

$$T(Q) = \begin{cases} |R| & \text{If } Q \text{ is a base relation } R. \\ |Q| + T(Q_1) + T(Q_2) & \text{If } Q = Q_1 \bowtie Q_2. \end{cases} \quad (1)$$

3.2 An Overview of PathEnum

Figure 2 gives an overview of our PathEnum algorithm. Given a graph G and a HcPE query $q(s,t,k)$, we first build a light-weight index to reduce the number of edges involving in the subsequent search. Next, we generate a join order based on the statistics of the index. We observe that the running time of different queries varies greatly because of the diverse size of the search space. Therefore, we optimize the search order in two steps. In the first step, we use a preliminary cardinality estimator to estimate the size of the search space. If the estimated size is small, then we directly invoke a depth-first search based method on the index to find results. Otherwise, we optimize the join order with a full-fledged cardinality estimation. This method makes more accurate estimation than the coarse-grained one at higher cost. However, the overhead is negligible when the running time of queries is long. The query optimizer selects the method with a lower cost to evaluate the query.

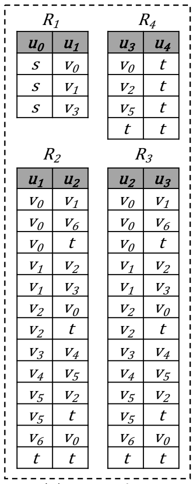
4 INDEX CONSTRUCTION

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 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.

Algorithm 2: Build Relations

```
   Input: a graph $G$, two distinct vertices $s, t$, hop constraint $k$;
   Output: a set of relations $R$;
   /* Initialize relations. */
1  $R_1 \leftarrow \{(s, v) | e(s, v) \in E(G)\}$;
2  $R_k \leftarrow \{(v, t) | e(v, t) \in E(G) \wedge v \neq s\} \cup \{(t, t)\}$;
3  for $i \leftarrow 2$ to $k - 1$ do
4      $R_i \leftarrow \{(v, v') | e(v, v') \in E(G - \{s\}) \wedge v \neq t\} \cup \{(t, t)\}$;
   /* Perform full reducer. */
5  for $i \leftarrow 1$ to $k - 1$ do
6      $C \leftarrow \{v' | (v, v') \in R_i\}$;
7      foreach $(v, v') \in R_{i+1}$ do
8          if $v \notin C$ then $R_{i+1} \leftarrow R_{i+1} - \{(v, v')\}$;
9  for $i \leftarrow k - 1$ to 1 do
10     $C \leftarrow \{v | (v, v') \in R_{i+1}\}$;
11     foreach $(v, v') \in R_i$ do
12         if $v' \notin C$ then $R_i \leftarrow R_i - \{(v, v')\}$;
13 return $\{R_1, ..., R_k\}$;
```



R_1

u_0	u_1
s	v_0
s	v_1
s	v_3

R_4

u_3	u_4
v_0	t
v_2	t
v_5	t
t	t

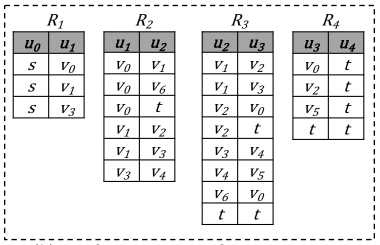
R_2

u_1	u_2
v_0	v_1
v_0	v_6
v_0	t
v_1	v_2
v_1	v_3
v_2	v_0
v_2	t
v_3	v_4
v_4	v_5
v_5	v_2
v_5	t
v_6	v_0
t	t

R_3

u_2	u_3
v_0	v_1
v_0	v_6
v_0	t
v_1	v_2
v_1	v_3
v_2	v_0
v_2	t
v_3	v_4
v_4	v_5
v_5	v_2
v_5	t
v_6	v_0
t	t

(a) Initial R**.**



R_1

u_0	u_1
s	v_0
s	v_1
s	v_3

R_2

u_1	u_2
v_0	v_1
v_0	v_6
v_0	t
v_1	v_2
v_1	v_3
v_3	v_4

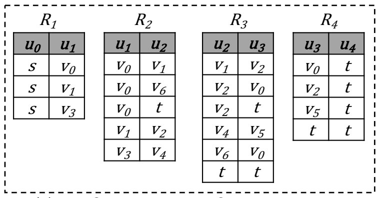
R_3

u_2	u_3
v_1	v_2
v_1	v_3
v_2	v_0
v_2	t
v_3	v_4
v_4	v_5
v_6	v_0
t	t

R_4

u_3	u_4
v_0	t
v_2	t
v_5	t
t	t

(b) R **after pruning from** R_2 **to** R_4**.**



R_1

u_0	u_1
s	v_0
s	v_1
s	v_3

R_2

u_1	u_2
v_0	v_1
v_0	v_6
v_0	t
v_1	v_2
v_3	v_4

R_3

u_2	u_3
v_1	v_2
v_2	v_0
v_2	t
v_4	v_5
v_6	v_0
t	t

R_4

u_3	u_4
v_0	t
v_2	t
v_5	t
t	t

(c) R **after pruning from** R_3 **to** R_1**.**

Figure 3: Build relations R **of** Q**.**

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_4. For example, (v_4, v_5) is removed from R_2 because v_4 does not appear in values of u_1 in R_1. Figure 3c presents the relations after filtering from R_3 to R_1. 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.

Proposition 4.2. *Each tuple in relations of 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.

Algorithm 3: Build Index

```
   Input: a graph $G$, two distinct vertices $s, t$, hop constraint $k$;
   Output: a light-weight index $\mathcal{I}$;
1  Set $v.s = S(s, v|G - \{t\})$ and $v.t = S(v, t|G - \{s\})$ for $v \in V(G)$;
2  $X \leftarrow$ a $(k + 1) \times (k + 1)$ matrix of sets;
3  foreach $v \in V(G)$ do
4      if $v.s + v.t \leqslant k$ then Add $v$ to $X[v.s, v.t]$;
5  $H \leftarrow$ a hash table;
6  foreach $v \in X - \{t\}$ do
7      Add a key-value pair $(v, \{\})$ to $H$;
8      foreach $v' \in N(v)$ do
9          if $v.s + v'.t + 1 \leqslant k$ then Add $v'$ to $H[v]$;
10 Add a key-value pair $(t, \{t\})$ to $H$;
11 Sort the values $v' \in H[v]$ of $v \in X$ by the ascending order of $v'.t$;
12 return $\mathcal{I}(X, H)$;
```

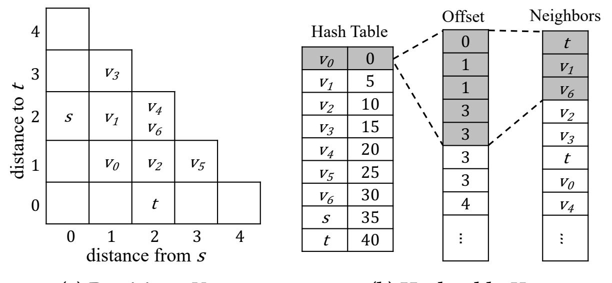
General Idea. Based on the definition of a path from s to t, we have the following proposition. Note that we set $S(v, v'|G - \{v''\})$ as $S(v, v'|G)$ if $v = v''$ or $v' = v''$.

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* $0 \leqslant i \leqslant |P| - 1$*, then* $S(s, v|G - \{t\}) \leqslant i$ *and* $S(v, t|G - \{s\}) \leqslant k - i$*.*

Let C_i denote the set of vertices $v \in V(G)$ that satisfy $S(s, v|G - \{t\}) \leqslant i$ and $S(v, t|G - \{s\}) \leqslant k - i$. 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_i. 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 $S(v', t|G - \{s\}) \leqslant b - 1$ to meet the hop constraint. 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 $0 \leqslant i \leqslant k$, retrieve C_i; and (2) given $v \in C_i$ and an integer b where $0 \leqslant b \leqslant k$, retrieve the neighbors v' of v such that $S(v', t|G - \{s\}) \leqslant b$ (or the in neighbors v' of v such that $S(s, v'|G - \{t\}) \leqslant b$).

Implementation. Algorithm 3 presents the details of building the index. For each $v \in V(G)$, Line 1 sets $v.s$ and $v.t$ as $S(s, v|G - \{t\})$ and $S(v, t|G - \{s\})$, respectively. We implement this by performing two breadth-first search from s and t, respectively. Lines 2-4 divide the vertices $v \in V(G)$ into disjoint sets based on $v.s$ and $v.t$. The partition considers the vertices v such that $v.s + v.t \leqslant k$ only. After that we build a hash table H to maintain the relationship between vertices in X and their neighbors (Lines 5-10). The key is the vertex v in X and the value is the set of neighbors v' of v that satisfy $v.s + v'.t + 1 \leqslant k$. Line 11 sorts the neighbors $v'.t$ of v in H by the ascending order of $v'.t$. Finally, we return the index $\mathcal{I}$ that 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 $v'.t$, the *Offset* array indexing the neighbor set of each vertex by the distance to t, and the *Hash Table* the key and value of 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 $\mathcal{I}$ on G and $q(s, t, 4)$ in Figure 1. Figure 4a shows the partitions X of $v \in V(G)$ based on $S(s, v|G - \{t\})$ and $S(v, t|G - \{s\})$. For example, $X[2, 2] = \{v_4, v_6\}$. Figure 4b demonstrates the implementation of H. Take v_0 as an example. It has three neighbors $\{t, v_1, v_6\}$, which are stored in the *Neighbors* array by the ascending order of the distance to t. As $k = 4$, v_0 has five slots in the *Offset* array to index $\{t, v_1, v_6\}$ based on the distance



(a) Partitions X. **(b) Hash table H.**

Figure 4: Build the index $\mathcal{I}$ on G.

to t. The value in H is 0, which points to the begin position of slots belonging to v_0 in the *Offset* array. Suppose that we want to retrieve the neighbors v of v_0 such that $S(v,t|G-\{s\}) \leqslant 2$. We first get the beginning position of the neighbor set of v_0 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 $\{t, v_1, v_6\}$.

Index Lookup Operations The index $\mathcal{I}$ supports two kinds of operations listed below.

- $\mathcal{I}(i)$: Retrieve C_i, i.e., the vertices $v \in V(G)$ satisfy that $S(s,v|G-\{t\}) \leqslant i$ and $S(v,t|G-\{s\}) \leqslant k-i$.
- $\mathcal{I}_t(v,b)$ (or $\mathcal{I}_s(v,b)$): Retrieve the neighbors v' of v such that $S(v',t|G-\{s\}) \leqslant b$ (or the in neighbors v' of v such that $S(s,v'|G-\{t\}) \leqslant b$).

$\mathcal{I}(i)$ and $\mathcal{I}_t(v,b)$ are implemented based on X and H. The time complexity of the two operations are both $O(1)$.

4.3 Analysis

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 $\mathcal{I}$ 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.

5 SEARCH ON INDEX

5.1 Depth-First Search on Index

Algorithm 4 presents the depth-first search method on the index $\mathcal{I}$. The *Search* procedure recursively enumerates all hop-constrained paths from s to t based on $\mathcal{I}$ (Lines 3-7). If the last vertex v in M is t, then we find a result and emit it (Line 5). Otherwise, we consider the neighbors v' of v such that $S(v',t|G-\{s\}) \leqslant k - L(M) - 1$ as the next vertex in M to meet the hop constraint. In particular, we loop over $\mathcal{I}_t(v, k-L(M)-1)$, add v' to M, and continue the search. The check at line 7 ensures that M contains no duplicate vertices.

Algorithm 4: Depth-First Search on Index

```
Input: two distinct vertices $s, t$, hop constraint $k$, index $\mathcal{I}$;
Output: all $k$ hop-constrained paths from $s$ to $t$;
1 $M \leftarrow (s)$;
2 Search($t, k, M, \mathcal{I}$);
3 Procedure Search($t, k, M, \mathcal{I}$)
4     $v \leftarrow$ the last vertex in $M$;
5     if $v = t$ then $emit(M)$, return;
6     foreach $v' \in \mathcal{I}_t(v, k - L(M) - 1)$ do
7         if $v' \notin M$ then Search($t, k, M \cup \{v'\}, \mathcal{I}$);
```

5.2 Analysis

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. Let $\mathcal{M}_i$ 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 $M \in \mathcal{M}_i$ corresponds to an invocation of the *Search* procedure. The cost of an invocation is $|\mathcal{I}_t(M[i], k-i-1)|$ (i.e., the *for* loop at Lines 6-7). Then, the running time T of Algorithm 4 is computed by Equation 2.

$$T = \sum_{0 \leqslant i \leqslant k-1} \sum_{M \in \mathcal{M}_i} |\mathcal{I}_t(M[i], k-i-1)|. \tag{2}$$

As Algorithm 4 generates partial results incrementally, $\mathcal{M}_{i+1}$ and $\mathcal{M}_i$ have the following relation where $0 \leqslant i \leqslant k-1$.

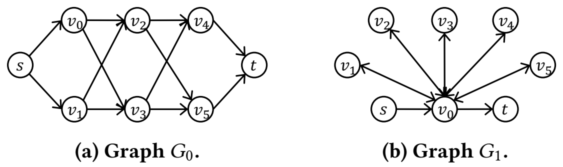
$$\mathcal{M}_{i+1} = \bigcup_{M \in \mathcal{M}_i} \{M \cup \{v\} | v \in \mathcal{I}_t(M[i], k-i-1) - M\}. \tag{3}$$

Because M is generated during the enumeration, it is hard to estimate the number of vertices $v \in \mathcal{I}_t(M[i], k-i-1)$ 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 $\widetilde{\mathcal{M}_i}$ denote the set of partial results containing $i+1$ vertices, which are generated by the algorithm after relaxation. According to Equation 3, $\widetilde{\mathcal{M}_{i+1}} = \bigcup_{M \in \widetilde{\mathcal{M}_i}} \{M \cup \{v\} | v \in \mathcal{I}_t(M[i], k-i-1)\}$ and $\mathcal{M}_i \subseteq \widetilde{\mathcal{M}_i}$. The algorithm after relaxation satisfies the following proposition.

Proposition 5.1. *Algorithm 4 without the check at line 7 finds all k hop-constrained walk $\mathcal{W}(s,t,G)$ from s to t in G. Given $M \in \widetilde{\mathcal{M}_i}$ where $0 \leqslant i \leqslant k$, 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, $|\widetilde{\mathcal{M}_i}| \leqslant \delta_W$ where $\delta_W = |\mathcal{W}(s,t,k,G)|$. Together with Equations 2 and 3, we get the following equation.

$$\begin{aligned} T &= \sum_{0 \leqslant i \leqslant k-1} \sum_{M \in \mathcal{M}_i} |\mathcal{I}_t(M[i], k-i-1)| \\ &\leqslant \sum_{0 \leqslant i \leqslant k-1} \sum_{M \in \widetilde{\mathcal{M}_i}} |\mathcal{I}_t(M[i], k-i-1)| \\ &= \sum_{0 \leqslant i \leqslant k-1} |\bigcup_{M \in \widetilde{\mathcal{M}_i}} \{M \cup \{v\} | v \in \mathcal{I}_t(M[i], k-i-1)\}| \\ &= \sum_{1 \leqslant i \leqslant k} |\widetilde{\mathcal{M}_i}| \leqslant k \times \delta_W. \end{aligned} \tag{4}$$



(a) Graph G_0. (b) Graph G_1.

Figure 5: Sample graphs.

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 when 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 δ_P and δ_W (i.e., $\frac{\delta_P}{\delta_W}$) 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. $|\mathcal{W}(s,t,4,G_0)|$ is equal to 8, and each walk in $\mathcal{W}(s,t,4,G_0)$ is a path in $\mathcal{P}(s,t,4,G_0)$, for example, $W = (s,v_0,v_2,v_4,t)$. 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 HcPE 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 $\sum_{1 \leqslant i \leqslant k} |\widetilde{\mathcal{M}}_i|$ where $\widetilde{\mathcal{M}}_0 = \{(s)\}$ and $\widetilde{\mathcal{M}}_i = \bigcup_{M \in \widetilde{\mathcal{M}}_{i-1}} \{M \cup \{v\} | v \in \mathcal{I}_t(M[i-1], k-i)\}$. Suppose that the average number of immediate partial results derived from partial results in $\widetilde{\mathcal{M}}_i$ is γ_i. Then, $|\widetilde{\mathcal{M}}_{i+1}| = \gamma_i \times |\widetilde{\mathcal{M}}_i|$.

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

$$\begin{aligned} \hat{T} &= \sum_{1 \leqslant i \leqslant k} |\widetilde{\mathcal{M}}_i| \approx \sum_{0 \leqslant i \leqslant k-1} \prod_{0 \leqslant j \leqslant i} \hat{\gamma}_j \\ &= \sum_{0 \leqslant i \leqslant k-1} \prod_{0 \leqslant j \leqslant i} \frac{1}{|C_j|} \sum_{v \in C_j} |\mathcal{I}_t(v, k - L(M) - 1)|. \end{aligned} \tag{5}$$

We compare $\hat{T}$ with a threshold τ. 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 τ such that the cost of the optimization is neglected compared with the search time when $\hat{T} > \tau$. In particular, given G, τ is measured by pre-executing some random queries with Algorithm 4 and testing τ from 10, 10^2,..., till the time finding τ results is longer than the join plan optimization time for most of queries. In our experiments, we execute 100 queries and set τ as 10^5, which works well in our workloads. This is because the optimization time on these graphs is generally shorter than the time of finding 10^5 results. Moreover, if the queries have fewer than 10^5 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 $Q' = R_{i+1}(u_i, u_{i+1}) \bowtie \cdots \bowtie R_j(u_{j-1}, u_j)$. We want to estimate $|Q[i:j]|$ based on the index.

We first consider a simple case $Q' = Q[i:i+1]$ where $0 \leqslant i < k$, i.e., Q' is a base relation $R_{i+1}(u_i, u_{i+1})$. Based on $\mathcal{I}$, we obtain that $R_{i+1} = \bigcup_{v \in C_i} \{(v, v') | v' \in \mathcal{I}_t(v, k-i-1)\}$. Thus, $|Q'| = |R_{i+1}| = \sum_{v \in C_i} |\mathcal{I}_t(v, k-i-1)|$. Let $c_{i+1}^i(v)$ denote the number of tuples starting with v in $Q[i:i+1]$. Given $Q' = Q[i-1:i+1]$, we have $|Q'| = |R_i \bowtie R_{i+1}| = \sum_{v \in C_{i-1}} c_{i+1}^{i-1}(v) = \sum_{v \in C_{i-1}} \sum_{v' \in \mathcal{I}_t(v, k-i)} c_{i+1}^i(v')$. Therefore, we compute $|Q[i:j]|$ as follows.

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

$$c_j^i(v) = \begin{cases} 1 & \text{If } i = j. \\ \sum_{v' \in \mathcal{I}_t(v, k-i-1)} c_j^{i+1}(v') & \text{If } i < j. \end{cases} \tag{7}$$

We estimate $|Q[0:k]|$ with a dynamic programming method based on the index $\mathcal{I}$. Given $0 \leqslant i \leqslant k$, we store $c_k^i(v)$ for each vertex $v \in \mathcal{I}(i)$. As such, the space cost of the full-fledged cardinality estimator is $\sum_{0 \leqslant i \leqslant k} |\mathcal{I}(i)|$. Based on Equation 7, we first set $c_k^k(v)$ to 1 for each $v \in \mathcal{I}(k)$. Given $0 \leqslant i \leqslant k$, $|Q[i:k]|$ is equal to

Algorithm 5: Join Order Optimization

```
   Input: two distinct vertices $s, t$, hop constraint $k$, index $\mathcal{I}$;
   Output: the cut position $i^*$;
   /* Estimate the number of paths to $t$.                 */
1  Set $c_k^i(v)$ as 0 for each $v \in \mathcal{I}(i)$ where $0 \leqslant i \leqslant k-1$;
2  Set $c_k^k(v)$ as 1 for each $v \in \mathcal{I}(k)$;
3  for $i \leftarrow k-1$ to 0 do
4      foreach $v \in \mathcal{I}(i), v' \in \mathcal{I}_t(v, k-i-1)$ do
5          $c_k^i(v) \leftarrow c_k^i(v) + c_k^{i+1}(v')$;
   /* Estimate the number of paths from $s$.               */
6  Set $c_i^0(v)$ as 0 for each $v \in \mathcal{I}(i)$ where $1 \leqslant i \leqslant k$;
7  Set $c_0^0(v)$ as 1 for each $v \in \mathcal{I}(0)$;
8  for $i \leftarrow 1$ to $k$ do
9      foreach $v \in \mathcal{I}(i), v' \in \mathcal{I}_s(v, k-i-1)$ do
10         $c_i^0(v) \leftarrow c_i^0(v) + c_{i-1}^0(v')$;
   /* Find the cut position $i^*$.                         */
11 $i^* \leftarrow arg \min_{0 \leqslant i \leqslant k}(\sum_{v \in \mathcal{I}(i)} c_i^0(v) + \sum_{v \in \mathcal{I}(i)} c_k^i(v))$;
12 return $i^*$;
```

$\sum_{v \in I(i)} c_k^i(v)$ where $c_k^i(v) = \sum_{v' \in \mathcal{I}_t(v,k-i-1)} c_k^{i+1}(v')$ according to Equations 6 and 7. Then, the cost of estimating $|Q[i:k]|$ based on $Q[i{+}1:k]$ is $\sum_{v \in I(i)} |\mathcal{I}_t(v, k-i-1)|$. With the method, we calculate $|Q[0:k]|$ along the order from $k-1$ to 0. Therefore, the cost is equal to $\sum_{0 \leqslant i \leqslant k-1} \sum_{v \in I(i)} |\mathcal{I}_t(v, k-i-1)|$. As $\mathcal{I}(i) \leqslant |V(G)|$, the space complexity is $O(k \times |V(G)|)$. Given $v \in \mathcal{I}(i)$, $|\mathcal{I}_t(v, k-i-1)| \leqslant d(v)$. As such, $\sum_{v \in \mathcal{I}(i)} |\mathcal{I}_t(v, k-i-1)| \leqslant \sum_{v \in \mathcal{I}(i)} d(v) \leqslant |E(G)|$, and the time complexity is $O(k \times |E(G)|)$. The number of edges in the index is generally smaller than $|E(G)|$ 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.

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' of $Q[0:i]$ (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 $v \in \mathcal{I}(i)$ (i.e., C_i) where $0 \leqslant i \leqslant k$, 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 $O(k \times |E(G)|)$ and the space complexity is $O(k \times |V(G)|)$.

Algorithm 6: Join On Index

```
   Input: two distinct vertices $s, t$, hop constraint $k$, cut position $i^*$, index $\mathcal{I}$;
   Output: all $k$ hop-constrained paths from $s$ to $t$;
1  $R_a \leftarrow \{\}, R_b \leftarrow \{\}$;
2  Search($M \leftarrow (s), t, 0, k, k-i^*, \mathcal{I}, R_a$);
3  $C \leftarrow \{r[i^*] | r \in R_a\}$;
4  foreach $v \in C$ do
5      Search($M \leftarrow (v), t, i^*, k, k-i^*+1, \mathcal{I}, R_b$);
6  $R \leftarrow R_a \bowtie_{HJ} R_b$;
7  foreach $r \in R$ do
8      if $r$ is a $k$ hop-constrained path from $s$ to $t$ then emit($r$);
9  Procedure Search($M, t, i, k, l, \mathcal{I}, R$)
10     if $|M| = l$ then $R \leftarrow R \cup \{M\}$, return;
11     $v \leftarrow$ the last vertex in $M$;
12     foreach $v' \in \mathcal{I}_t(v, k-i-L(M)-1)$ do
13         Search($M \cup \{v'\}, t, i, k, l, \mathcal{I}, R$);
```

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 $T_{DFS} = \sum_{1 \leqslant i \leqslant k} |Q[0:i]|$ based on Equation 1. In contrast, the cost of the new order is $T_{JOIN} = |Q| + T(Q[0:i^*]) + T(Q[i^*:k]) = |Q| + \sum_{1 \leqslant i \leqslant i^*} |Q[0:i]| + \sum_{i^* < i \leqslant k} |Q[i:k]|$. Based on the intermediate results in Algorithm 5, we can get that $T_{DFS} = \sum_{1 \leqslant i \leqslant k} \sum_{v \in \mathcal{I}(i)} c_i^0(v)$, while $T_{JOIN} = \sum_{v \in \mathcal{I}(0)} c_k^0(v) + \sum_{1 \leqslant i \leqslant i^*} \sum_{v \in \mathcal{I}(i)} c_i^0(v) + \sum_{i^* \leqslant i \leqslant k} \sum_{v \in \mathcal{I}(i)} c_k^i(v)$. 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_a and R_b 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 l 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 $S(v', t|G - \{s\}) \leqslant k - i - L(M) - 1$, 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 $\mathcal{W}(s, t, k, G)$.*

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 $|R| = |\mathcal{W}(s, t, k, G)|$, 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 $R = R_a \bowtie_{HJ} R_b$ 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 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 (δ_W). As a result, if the gap between the number of walks (δ_W) and the number of paths (δ_P) is small, then our method can give an accurate estimation. Otherwise, the method can introduce some errors. Our extensive experiment results in the complete version [37] 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 1 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' and V'' based on the vertex degrees: (1) V' is the set of vertices within top 10% in the descending order of their degrees; and (2) V'' is the remaining ones $V(G)$, excluding V'. Then, we have four settings according to the locations of s and t: $\{V', V''\} \times \{V', V''\}$. 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' 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 $s, t \in V'$ and $k = 6$ by default.

Table 1: Properties of real-world graphs.

Name	Dataset	|V|	|E|	d_{avg}	Type
up	US Patents[1]	4M	17M	8.8	Citation
db	DBpedia[2]	4M	14M	6.5	Miscellaneous
gg	Web-google[1]	876K	5M	11.1	Web
st	Web-standford[1]	282K	2.3M	16.4	Web
tw	Twitter-social[2]	465K	835K	3.6	Miscellaneous
bk	Baidu-baike[2]	416K	3M	15.8	Web
tr	Wiki-trust[2]	139K	740K	10.7	Interaction
ep	Soc-Epinsion1[1]	75K	508K	13.4	Social
uk	Web-uk-2005[2]	121K	334K	181.2	Web
wt	WikiTalk[2]	2M	5M	4.2	Miscellaneous
sl	Soc-Slashdot0922[1]	82K	948K	21.2	Social
lj	LiveJournal[1]	5M	69M	28.3	Social
da	Rec-dating[2]	169K	17M	205.7	Recommendation
ye	Bio-grid-yeast[2]	6K	314K	104.5	Biological
tm	Twitter-mpi[2]	52M	1.96B	74.7	Miscellaneous

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 (1.2×10^5 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.

- 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.

The comparison between IDX-DFS/IDX-JOIN and PathEnum is to demonstrate the effectiveness of our cost-based selection. Also note, Peng et al. [29] showed that BC-DFS and BC-JOIN outperform T-DFS [33], T-DFS2 [14], KRE [13], KPJ [8] and HPI [32] by orders of magnitude.

7.2 Comparison with Existing Algorithms

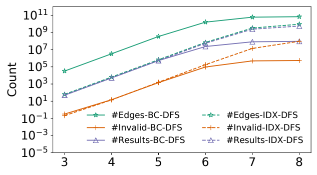
Overall Comparison. Table 2 gives an overall comparison of competing algorithms on different graphs. We only report the response time of BC-DFS and IDX-DFS because the join-based methods have to obtain the results of each sub-query before computing the final results, which have a long response time. As shown in the table, the query time on different graphs varies greatly, which ranges from less than one millisecond to tens of seconds.

[1]http://snap.stanford.edu/data/, Last accessed on 2021/03/29

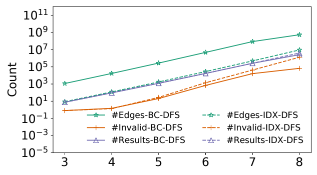
[2]http://networkrepository.com/networks.php, Last accessed on 2021/03/29

Table 2: 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	Query Time (ms)					Throughput (#Results Per Second)					Response Time (ms)	
	BC-DFS	BC-JOIN	IDX-DFS	IDX-JOIN	PathEnum	BC-DFS	BC-JOIN	IDX-DFS	IDX-JOIN	PathEnum	BC-DFS	IDX-DFS
up	5.75e+0	4.26e+0	2.75e-1	2.41e+1	**2.28e-1**	1.46e+3	1.97e+3	3.06e+4	3.50e+2	**3.68e+4**	5.75e+0	**2.75e-1**
db	1.13e+1	1.05e+1	7.97e-1	4.06e+1	**6.22e-1**	1.14e+4	1.24e+4	1.63e+5	3.20e+3	**2.09e+5**	1.13e+1	**7.97e-1**
gg	1.83e+2	6.64e+1	**9.67e-1**	8.08e+0	1.16e+0	9.38e+4	2.58e+5	**1.77e+7**	2.12e+6	1.48e+7	4.65e+1	**6.67e-1**
st	3.67e+3	4.05e+2	4.44e+0	4.95e+0	**3.28e+0**	4.98e+4	5.29e+5	4.82e+7	4.32e+7	**6.52e+7**	1.21e+2	**1.32e+0**
tw	3.35e+2	4.16e+2	**1.72e+0**	2.96e+0	1.78e+0	5.60e+1	4.51e+1	**1.09e+4**	6.33e+3	1.05e+4	3.35e+2	**1.72e+0**
bk	7.08e+3	2.63e+3	9.19e+1	**7.57e+1**	9.29e+1	5.25e+4	7.29e+5	1.88e+8	**2.29e+8**	1.87e+8	4.68e+2	**2.14e+0**
tr	9.88e+4*	1.17e+4	2.39e+2	1.00e+2	**9.83e+1**	1.15e+4	8.33e+5	5.26e+7	1.26e+8	**1.28e+8**	1.07e+3	**1.76e+1**
ep	1.06e+5*	2.34e+4	6.55e+2	**2.78e+2**	3.79e+2	1.34e+4	1.04e+6	8.45e+7	**2.00e+8**	1.46e+8	7.35e+2	**1.28e+1**
uk	4.87e+4*	4.47e+4*	3.88e+3	4.68e+3	**3.84e+3**	7.95e+5	9.85e+5	3.21e+8	2.45e+8	**3.24e+8**	1.61e+1	**4.23e-1**
wt	1.05e+5*	3.23e+4	1.70e+3	5.14e+2	**4.79e+2**	5.33e+3	6.20e+5	3.49e+7	1.17e+8	**1.26e+8**	1.08e+4	**1.58e+2**
sl	1.20e+5*	6.10e+4*	2.76e+3	7.51e+2	**7.18e+2**	1.43e+4	1.02e+6	5.02e+7	1.85e+8	**1.93e+8**	1.42e+3	**3.81e+1**
lj	1.20e+5*	1.20e+5*	8.50e+2	6.39e+2	**4.99e+2**	1.35e+3	2.38e+4	1.69e+7	2.24e+7	**2.88e+7**	1.57e+5	**4.38e+2**
da	1.20e+5*	1.20e+5*	1.26e+4	3.84e+3	**3.32e+3**	2.10e+3	4.14e+5	2.88e+7	1.19e+8	**1.36e+8**	4.13e+4	**5.78e+2**
ye	1.20e+5*	1.20e+5*	7.88e+4*	1.18e+5*	**6.46e+4**	6.67e+4	9.40e+5	1.87e+8	4.44e+7	**2.34e+8**	3.86e+2	**1.01e+1**



(a) *ep*.



(b) *gg*.

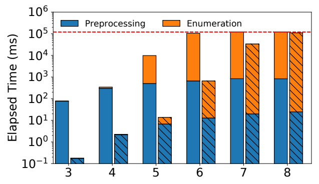
Figure 6: Comparison of detailed metrics with k varied.

PathEnum vs. BC-DFS/BC-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.

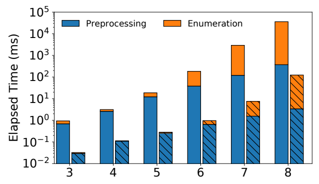
IDX-DFS vs. IDX-JOIN. IDX-DFS outperforms IDX-JOIN on graphs with short queries, but generally runs slower on graphs with long queries. For example, IDX-DFS achieves up to two orders of magnitude speedup over IDX-JOIN on *up* because there is a small number of results (i.e., the search space is small) 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. 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 runs 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.



(a) *ep*.



(b) *gg*.

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

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 $\mathcal{P}(s, t, k, G)$; (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 extra pruning power compared with simply using the distance to t 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.

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.

Query Time Distribution. Table 3 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

Table 3: Query time distribution on *ep* and *gg*.

	ep				*gg*			
	BC-DFS		IDX-DFS		BC-DFS		IDX-DFS	
k	<60s	>120s	<60s	>120s	<60s	>120s	<60s	>120s
3	1.00	0.00	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	1.00	0.00
5	0.959	0.016	1.00	0.00	1.00	0.00	1.00	0.00
6	0.130	0.813	1.00	0.00	1.00	0.00	1.00	0.00
7	0.003	0.997	0.877	0.066	0.994	0.003	1.00	0.00
8	0.001	0.999	0.231	0.674	0.749	0.138	1.00	0.00

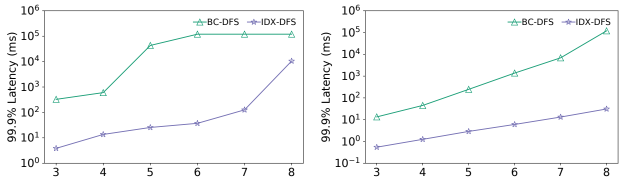
120 seconds. We can see that the number of queries running out of time increases with k varied from 3 to 8 on *ep*. IDX-DFS significantly outperforms BC-DFS, especially when k is large. For example, IDX-DFS completes 23.1% queries within 60 seconds, whereas BC-DFS only completes 0.1% queries. Furthermore, IDX-DFS completes all queries on *gg* within 60 seconds.

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 4 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 4: Performance for queries with different query time on *ep* with $k = 8$.

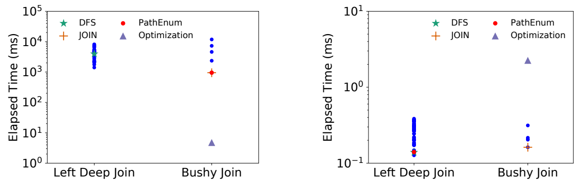
	Throughput		Response Time (ms)	
Method	<60s	>120s	<60s	>120s
BC-DFS	1.52e+04	8.65e+03	6.11e+02	3.03e+03
IDX-DFS	7.83e+06	5.13e+07	2.12e+01	2.52e+01

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 $e(v, v')$, we set v' and v as s and t, 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.



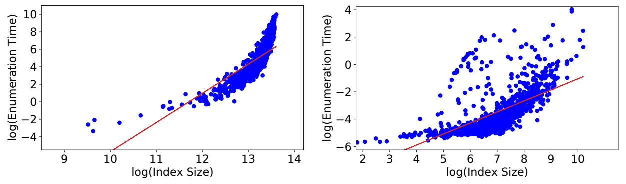
(a) ***ep*.** **(b)** ***gg*.**

Figure 8: Comparison of 99.9% latency with k varied.



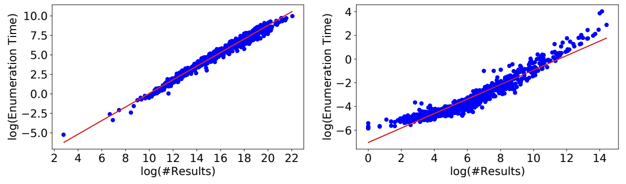
(a) ***ep*.** **(b)** ***gg*.**

Figure 9: Spectrum analysis of join plan optimization.



(a) ***ep*.** **(b)** ***gg*.**

Figure 10: Impact of index size on enumeration time.



(a) ***ep*.** **(b)** ***gg*.**

Figure 11: Impact of #results on enumeration time.

7.3 Evaluation of Individual Techniques

Spectrum Analysis. We conduct the spectrum analysis to study the effectiveness of our join plan optimization method. Particularly, given Q, we categorize all join plans into the left deep tree and the bushy tree based on the shape of join trees. The left deep tree extends partial results by a vertex at a step (e.g., Algorithm 4), whereas the bushy tree performs the join on partial results of two sub-queries of Q (e.g., Algorithm 6). We enumerate all left deep trees of Q without the Cartesian product. In contrast, for the bushy tree, we consider all cut positions i where $0 < i < k$ that divides Q into two sub-queries $Q[0 : i]$ and $Q[i : k]$ and evaluate $Q[0 : i]$ and $Q[i : k]$ with the depth-first search method because the join of $Q[0 : i]$ and $Q[i : k]$ dominates the cost.

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. The index size is measured by the number of edges in the index. As

Table 5: 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	3.00e+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

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.

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 5 presents the experiment results. The star symbol denotes that we cannot enumerate all results within 120 seconds and report the value found by IDX-DFS within the time limit. We can see that the number of results significantly increases with k varied from 3 to 8, and the number of results on *ep* is much more than taht *gg*. Therefore, the query time on *ep* is longer than that on *gg*, and the query time significantly increases with the increasing of k as shown in Figure 7.

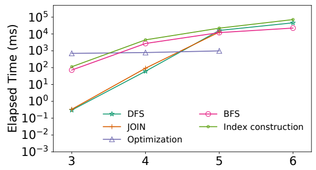
Memory Cost. Table 6 presents the maximum memory consumption on indexes and partial results of IDX-JOIN with k varied. The index consumes a small amount of memory space as the space complexity of the index is $O(|E(G)| + k \times |V(G)|)$ and the filtering can effectively prune some vertices and edges. The partial results of IDX-JOIN on *ep* consume much more space than that on *gg* because there are more results on *ep* than *gg* as shown in Table 5.

Table 6: Maximum memory consumption (MB) on *ep* and *gg*.

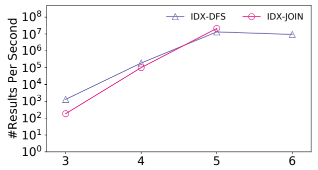
	k	3	4	5	6	7	8
Index	*ep*	0.15	1.68	3.28	4.60	5.45	5.91
	gg	0.01	0.10	0.11	0.21	0.44	0.63
Partial Results	*ep*	0.03	0.66	26.51	138.32	4561.59	21479.32
	gg	0.01	0.02	0.37	1.05	17.80	55.70

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 $k = 5$.



(a) Execution time.



(b) Throughput.

Figure 12: Scalability evaluation on *tm* with k varied.

7.5 Discussions

Although PathEnum significantly accelerates HcPE queries, there still leaves interesting future work. First, our join optimizer can be further improved by searching the optimal plan in a larger plan space and considering more metrics such as the cost of materializing partial results. Second, developing algorithms having a short response time on very large graphs is an interesting research direction because building the index from scratch on very large graphs can take a long time (e.g., tens of seconds on *tm*). A promising approach is to build a global index in an offline preprocessing step to reduce the cost of construing the query-dependent index. However, designing an effective global index is challenging because (1) such an index has to maintain the global statistics of G to serve all queries and therefore must balance the cost of the index and query efficiency; and (2) the index needs to support efficient update operations to serve dynamic graphs.

Additionally, we observe some opportunities for graph database systems [1, 26] to explore. The query-dependent index can reduce elements involved in the computation and provide accurate statistics to the query optimizer. This gives graph databases an alternative way of evaluating queries by dividing the evaluation into two phases: (1) builds a query-dependent index; and (2) generates the query plan and computes based on the index. Moreover, the systems can adopt an adaptive query optimizer to process queries because the query time of different queries can vary greatly.

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 online applications such as fraud detection in massive transactions in future digital finance. We design a light-weight index, and two index-based approaches for efficient enumeration. 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.

ACKNOWLEDGMENTS

This project is supported by the grant "Asian Institute of Digital Finance" awarded by National Research Foundation, Singapore and administered by the Infocomm Media Development Authority under its Smart Systems Strategic Research Programme in 2020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. This project is also in part supported by Alibaba Group through Alibaba Innovative Research (AIR) Program 2021.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 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 Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. 2013. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1884–1896.

[7] Kateřina Böhmová, Luca Häfliger, Matúš Mihalák, Tobias Pröger, Gustavo Sacomoto, and Marie-France Sagot. 2018. Computing and listing st-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 Eppstein. 1998. Finding the k shortest paths. *SIAM Journal on computing* 28, 2 (1998), 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 paths 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 st-paths in graphs. In *Latin American Symposium on Theoretical Informatics*. Springer, 544–557.

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

[16] Czeslaw Jedrzejek, J Bak, and M Falkowski. 2009. Graph mining for detection of a large class of financial crimes. In *17th International Conference on Conceptual Structures, Moscow, Russia*, Vol. 46.

[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, Mihalis 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, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: 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), 1441–1453.

[22] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang 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 Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[24] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. 2020. FlowScope: Spotting 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 Salihoglu. 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] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-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 st simple path enumeration. *Proceedings of the VLDB Endowment* 13, 4 (2019), 463–476.

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

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

[34] Baoxu 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 Ciampaglia. 2017. Finding streams in knowledge graphs to support fact checking. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[36] Avadhesh Pratap Singh and Dhirendra Pratap Singh. 2015. Implementation of K-shortest path algorithm in GPU using CUDA. *Procedia Computer Science* 48 (2015), 5–13.

[37] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained s-t Path Enumeration (Complete Version). *arXiv preprint arXiv:2103.11137* (2021).

[38] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An Indepth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[39] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

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

[41] Hongwei Wang, Hongyu Ren, and Jure Leskovec. 2020. Entity Context and Relational Paths for Knowledge Graph Completion. *arXiv preprint arXiv:2002.06757* (2020).

[42] 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.

[43] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *management Science* 17, 11 (1971), 712–716.