Efficient Distributed Hop-Constrained Path Enumeration on Large-Scale Graphs

Anonymous Author(s)

ABSTRACT

The enumeration of hop-constrained simple paths is a building block in many graph-based areas. Due to the enormous search spaces in large-scale graphs, a single machine can hardly satisfy the requirements of both efficiency and memory, which causes an urgent need for efficient distributed methods. In practice, it is inevitable to produce plenty of intermediate results when directly extending centralized methods to the distributed environment, thereby causing a memory crisis and weakening the query performance. The state-of-the-art distributed method HybridEnum designed a hybrid search paradigm to enumerate simple paths. However, it makes massive exploration for the redundant vertices not located in any simple path, thereby resulting in poor query performance. To alleviate this problem, we design a distributed approach **DistriEnum** to optimize query performance and scalability with well-bound memory consumption. Firstly, DistriEnum adopts a graph reduction strategy to rule out the redundant vertices without satisfying the constraint of hop number. Then, a core search paradigm is designed to simultaneously reduce the traversal of shared subpaths and the storage of intermediate results. Moreover, DistriEnum is equipped with a task division strategy to theoretically achieve workload balance. Finally, a vertex migration strategy is devised to reduce the communication cost during the enumeration. The comprehensive experimental results on 10 real-world graphs demonstrate that DistriEnum achieves up to 3 orders of magnitude speedup than HybridEnum in query performance and exhibits superior performances on scalability, communication cost, and memory consumption.

KEYWORDS

distributed computing, graph traversal, path enumeration

ACM Reference Format:

Anonymous Author(s). 2024. Efficient Distributed Hop-Constrained Path Enumeration on Large-Scale Graphs. In *Proceedings of Proceedings of the 2024 International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Given a vertex pair (s, t) and a hop constraint k, hop-constrained path enumeration (HcPE) returns all simple paths between s and t from a given graph G, where the length of each path is no more than k. HcPE is one of the fundamental problems in graph analytics and serves as a building block in many graph-based areas [11, 23], including E-commerce networks [14, 22], knowledge graphs [27, 28], and biological networks [16, 22]. Several representative scenarios are listed as follows.

(1) **E-commerce Networks**. In E-commerce networks, the online shopping process can be represented as a graph. In this graph, individual users such as sellers and buyers are depicted as vertices, while online transactions like online payment and shipment of goods are represented by edges [23, 25]. By analyzing this graph, we can identify circles that indicate potential fraudulent activities among the users involved [23]. To calculate all the new cycles created by inserting an edge (s, t), one can obtain all the simple paths between s and t. Furthermore, introducing the constraint of the hop number can help reduce the occurrence of false alarms [23, 25].

(2) **Knowledge Graphs**. Knowledge graphs play a vital role in recommendation systems, but their incompleteness can significantly impact the user experience. To address this issue, path enumeration techniques can be employed to augment the edges between different entities by enumerating simple paths connecting them. These supplemented edges can then serve as features to train models and predict missing relationships [29]. The constraint of hop number can be utilized to enhance the connectivity between entities, thereby improving the overall quality of the knowledge graph [27, 28]. It should be noted that long paths are generally less useful in capturing strong relationships between two entities, as their corresponding relation strength tends to be weak [23].

Motivation. In real-life scenarios, the data graphs tend to be large and grow exponentially [14]. However, processing such large graphs becomes challenging for a single machine due to the limited memory and computation resources, rendering centralized algorithms unsuitable and non-scalable for these scenarios [14]. To overcome this challenge, designing efficient distributed approaches that offer good query efficiency and scalability is crucial [14]. While the literature contains numerous excellent centralized techniques, implementing them in the distributed setting is often not feasible. For R3M7
instance, state-of-the-art centralized methods like PathEnum [29] and EVE [6] have successfully constructed condensed graphs to reduce the search space of simple paths. However, when deployed in the distributed setting, the enumeration process can meet memory bottleneck caused by intermediate results, especially for large values of k, resulting in substantial memory consumption.

As the pioneering work in studying distributed HcPE, HybridEnum [14] introduced a novel hybrid search paradigm that ensures low memory overhead. It also employed a divide-and-conquer strategy to reduce unnecessary computations during the enumeration process. Moreover, the algorithm incorporated mechanisms such as work-stealing and caching to handle unbalanced workloads and optimize communication costs, respectively. Although HybridEnum

SIGMOD '24, June 11 - June 16, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

adopted effective strategies to prune computations, it still encountered exploration for redundant vertices not involved in any simple paths. Additionally, frequent message exchanges among different machines adversely impacted its query performance. Inspired by these limitations, our goal is to develop an efficient and scalable distributed algorithm to address HcPE. The proposed algorithm aims to optimize the performance of query efficiency, scalability, and memory cost for large-scale graphs. R1O3

Challenges. Developing an algorithm that simultaneously achieves the aforementioned goals presents several challenges. Improving query performance necessitates minimizing the exploration of redundant vertices that are not located in any simple path and reducing the frequency of message exchanges between machines. R1O3 To optimize scalability, it is crucial to design an effective task division strategy that achieves workload balance with reasonable time costs. Additionally, constraining the number of intermediate results, such as subpaths and communication messages, becomes essential to decrease memory consumption and enhance scalability.

Our Approach. Based on the aforementioned analysis, we propose an efficient distributed approach called DistriEnum that simultaneously optimizes query performance and scalability, whilst providing a well-bound memory consumption. DistriEnum consists of four modules, each contributing to optimizing the overall performance of the approach. The core components of DistriEnum are the first two modules, while the latter two modules further enhance its performance. The detailed descriptions of these modules are as follows:

- **Graph reduction.** This module not only reduces the exploration by ruling out vertices that cannot satisfy the hop constraint but also provide pruning operations to early terminate the search.
- **Core search paradigm.** This module adopts a depth-first search (DFS) procedure to traverse the backbone of paths and the join-oriented method to acquire the final results, respectively. This search paradigm reduces the memory cost of subpath storage and minimizes the computational cost of concatenating subpaths.
- **Task division.** In this module, we theoretically analyze the upper bound of the searching scope of query tasks and propose an effective task division strategy to achieve workload balance, thereby largely improving scalability.
- **Vertex migration.** This module implements a vertex-cut partitioning strategy to migrate candidate vertices to corresponding machines, which helps to avoid frequently exchanging messages and reduce intermediate results. Additionally, it optimizes workload balance by addressing the uneven distribution of candidate vertices.

Furthermore, our method can be easily extended to solve the problem of path enumeration with other constraints, such as edge labels [10], distance [19], and time series [8, 23], which are relevant to various real-life applications. For example, edge labels can be utilized to identify specific types of enzymes or reactions in biological networks. Time series can be employed to monitor epidemic situations [23].

Contributions. In this paper, we make the following principal contributions:

- We propose an efficient solution, **DistriEnum**, which employs a core search paradigm on a sketch graph to enable fast enumeration of simple paths while maintaining well-bound memory consumption.
- To further enhance query efficiency and scalability, we introduce the modules of task division and vertex migration into DistriEnum.
- We conduct extensive experiments with various workloads to demonstrate the superior performance of our proposed method.

Roadmap. The rest of the paper is organized as follows. Section 2 reviews important related work. Section 3 presents the problem of HcPE and analyzes existing methods, and Section 4 provides a detailed introduction to the DistriEnum framework. Section 5 evaluates the performance of our method, and finally, Section 6 concludes the paper.

2 RELATED WORK

In this section, we discuss the related work on path enumeration, specifically focusing on hop-constrained path enumeration (HcPE). We also provide a brief review of other path-related problems, including reachability queries and distance queries.

2.1 Path Enumeration

We begin with the simple path enumeration problem and next discuss the hop-constrained path enumeration, i.e., HcPE, the focus of this paper.

Simple Path (or Cycle) Enumeration. Several existing works [5, 21, 25, 33] focus on efficiently listing the simple paths or cycles between source and target nodes. These methods leverage representation structures to avoid explicitly storing each individual result, but they can incur significant memory costs. However, it is time-consuming for these methods to resolve HcPE [4, 7, 17] since the search space of each query cannot be pruned when neglecting the hop constraints. R3M1 Additionally, some works [2, 3, 13] focus on detecting the existence of cycles in dynamic graphs rather than enumerating the results.

Hop-constrained Path Enumeration (HcPE). In [23], the authors proposed a barrier-based pruning technique to accelerate the performance of HcPE. This strategy leverages failure enumeration results to avoid exploring non-promising search branches in the future. Moreover, a join-based method, called JOIN, is constructed to further optimize the query efficiency with large k values by reducing the redundant traversal of shared subpaths while taking expensive space overhead.

PathEnum [29] constructed a lightweight index to exclude the vertices which cannot satisfy the constraint of hop numbers, thereby avoiding exploring non-promising search branches. They further propose a DFS-based approach and an effective join-oriented method to enumerate all simple paths. EVE [6] improves upon PathEnum by establishing a more condensed graph, reducing exploration spaces, and accelerating enumeration.

Although these algorithms provide effective strategies to prune redundant computation and enhance query efficiency, they may encounter performance issues, such as intermediate results and frequent communication, when directly applied to distributed environments.

2.2 Reachability and Distance Queries

Reachability Queries. Reachability queries involve confirming the existence of directed paths between two vertices in a graph. Many works [12, 23, 30, 34, 35] improve query efficiency by building effective indexes. In [9, 23, 31], the reachability queries can be resolved based on the 2-hop index without the data graph. However, the simple paths of each vertex pair are not recorded in the 2-hop index. Therefore, it is impossible to only use this index to enumerate all simple paths between the given two vertices. More details have been summarized in [36]. R3M2

Distance Queries. A distance query asks about the distance between two vertices in a graph, which receives a lot of research interests [1, 9, 19, 20, 24]. In [1], the authors constructed a landmark-based index to serve all queries and evaluated the query with the pre-computed results. In [19], the authors proposed a parallel method to accelerate the construction of the 2-hop index. However, these methods or indexes cannot be used for path enumeration.

3 PRELIMINARY

In this section, we begin by presenting the problem of HcPE. Subsequently, we thoroughly examine the state-of-the-art approaches. Table 1 summarizes frequently used notations in this paper.

Table 1: Notations and meanings.

Notations	Meanings
G=(V, E)	an undirected graph
(u, v)	a vertex pair
$N(v, G)$	the neighbor set of v in G
$deg(v)$	the degree of v
$w(s, t)$	a walk from s to t
$p(s, t)$	a simple path from s to t
$hop(p)$	the hop number of the simple path p
$v.s$ and $v.t$	the minimal hop numbers from v to s and t
$W(s, t, k, G)$	a set of walks from s to t
$P(s, t, k, G)$	a set of simple paths from s to t
SP_s and SP_t	sets of source and target paths
SP_m	sets of backbone paths

3.1 Problem Description.

Let G=(V, E) be an undirected graph where V is a set of n vertices and $E \subseteq V \times V$ is a set of m edges. We use $N(v, G)$=$\{u | e(u, v) \in E\}$ as the neighbor set of v in G. $deg(v)$=$|N(v, G)|$ denotes the degree of v. Given a pair of vertices (s, t), we represent a walk between s and t with potentially duplicate vertices as $w(s, t) = \langle v_0 = s, v_1, \ldots, v_{k-1}, v_k = t \rangle$, where $e(v_i, v_{i+1}) \in E$ for $i \in [0, k-1]$. Comparably, $p(s, t)$ denotes a simple path between s and t with no duplicate vertices. $hop(p)$ denotes the hop number (the number of edges) of this path p. We say a path p is a k-hop-constrained path if $hop(p) \leq k$, where k is a pre-defined hop number. For presentation simplicity, we use a hc-s-t path to denote a hop-constrained s-t simple path.

Graph Partitioning. In the distributed environment, the data graph is divided into partitions based on vertices and distributed across multiple machines. The entire adjacency list of the graph is spread across different machines. It is ensured that for each vertex, its corresponding adjacency list is stored in the same memory space. By default, we utilize the commonly adopted hash partitioning method [18] to divide the data graph.

Problem Definition. Given an undirected graph G, a hop constraint k, and two vertices s and t, the query task $q(s, t, k)$ aims to find all simple paths between s and t, where the length of each simple path is no more than k.

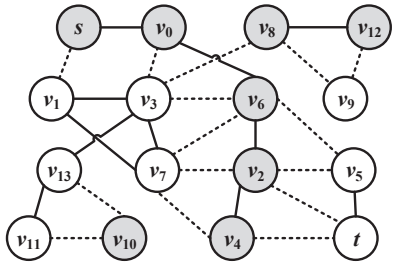


Figure 1: Example of graph G stored in two machines

Example 1. *Fig. 1 depicts a graph G where the vertices represented by white and grey in G are stored in two machines respectively. The dotted line means that the vertices of this edge are placed in different machines. Given an HcPE query task $q(s, t, 4)$, two hc-s-t paths can be found, namely $\langle s, v_1, v_7, v_4, t \rangle$ and $\langle s, v_1, v_7, v_2, t \rangle$.*

3.2 State-of-the-art Approaches

PathEnum [29]. PathEnum introduces a lightweight index structure to identify candidate vertices and employs two index-based search approaches, IDX-DFS and IDX-JOIN, for enumeration. IDX-DFS utilizes a depth-first search (DFS) strategy to generate all simple paths, while IDX-JOIN incorporates path concatenation to optimize the traversal of shared subpaths. However, IDX-DFS often traverses numerous identical subpaths, resulting in extensive redundant computations [29], and IDX-JOIN requires additional memory to store these subpaths. Despite its effectiveness, the distributed extension of PathEnum exhibits two limitations: R1O3

- *Memory consumption.* The DFS-based strategy utilizes memory efficiency in a centralized setting by maintaining a data list to track the visited vertices. Directly applying the DFS-oriented approach in IDX-DFS results in significant intermediate result generation, especially for large values of k. Compared to IDX-DFS, IDX-JOIN adopts a join-oriented operation to minimize redundant traversal of shared subpaths. However, this method incurs significant memory overhead due to the storage of all subpaths, and the time overhead of concatenating subpaths is also expensive. R1O3
- *Poor scalability.* PathEnum faces challenges in achieving workload balance and scalability in the distributed setting. Theoretically, the generation of subtasks with comparable workloads is difficult to achieve. Moreover, the uneven distribution of candidate vertices can lead to significant differences in computational overhead among machines. Specifically, DFS is inherently sequential and challenging to parallelize effectively in distributed systems [14], limiting the exploitation of computational resources of distributed systems. Furthermore, the uneven distribution of subpaths across machines can lead to R1O3

memory overflow in certain machines, adversely affecting scalability and practicality.

EVE [6]. EVE improves upon PathEnum by further compressing the search space of query tasks through the elimination of redundant vertices and edges that do not participate in any simple path. However, EVE still relies on the DFS-based and JOIN-based paradigms for obtaining the final results. As the reduced graph is distributed among different machines, it is necessary to exchange messages when visiting the graph placed in other machines, thus producing many communication costs and seriously damaging the query efficiency. R2O1

HybridEnum [14]. HybridEnum adopts a divide-and-conquer strategy to minimize memory consumption and enhance enumeration efficiency. It introduces work-stealing and caching mechanisms to address workload imbalances and optimize communication costs. However, HybridEnum has the following three shortcomings:

- *Redundant computations.* Although HybridEnum employs effective strategies to reduce search spaces, it does not directly eliminate redundant vertices that do not meet the hop number constraint. As a result, query efficiency is negatively impacted.
- *Frequent communication.* HybridEnum necessitates frequent message exchanges among machines due to the random distribution of candidate vertices. This communication overhead can hamper query performance. Although a cache mechanism is deployed in HybridEnum to store a part of common vertices and edges, this strategy heavily relies on the size of the cache graph and cannot avoid the production of intermediate results. R1O3
- *Workload imbalance.* While HybridEnum incorporates a work-stealing mechanism to migrate subtasks to idle machines and enhance query efficiency, achieving theoretical workload balance across all machines during enumeration is challenging since this strategy needs to be executed frequently during enumeration and is not equipped with a theoretical guarantee to keep workload balance. R1O3

Based on the aforementioned analysis, there is a need for a more effective distributed method that optimizes query efficiency, scalability, and memory utilization.

Example 2. *Take $q(s,t,6)$ in Fig 1 as an example. The reduced graphs in PathEnum and EVE are the same as in Fig. 3(a), as every edge belongs to at least one simple path. For a simple path $p(s,t) = \langle s, v_1, v_7, v_4, t\rangle$ of this query, each dashed line in this path represents a message exchange. PathEnum and EVE need to make three message exchanges to get this simple path. Based on this calculation, the total number of message exchanges over all paths of $q(s,t,6)$ is 186. Apparently, the frequent communication heavily impairs the query efficiency.* R2O1

By contrast, HybriEnum needs to make a DFS-based paradigm to visit all vertices where the hop number away from s is no more than 6. Therefore, it is inevitable for this method to visit the redundant vertices v_8 to v_{13}. R3M3

4 OUR FRAMEWORK OVERVIEW

In this section, we first give an overview of the framework of DistriEnum. Then, we introduce each module deployed in DistriEnum in detail.

4.1 Overview

According to the analysis in Section 3.2, we aim to overcome the difficulties of existing methods in the distributed setting. The overview of our DistriEnum approach is illustrated in Fig. 2. When given a query task $q(s,t,k)$ in the graph G, we first apply the graph reduction module to eliminate redundant vertices and generate a sketch graph. This sketch graph serves as the basis for subsequent procedures. Then, we design a new search paradigm to efficiently enumerate all simple paths with limited memory overhead. Furthermore, we design two optimization modules of task division and vertex migration to optimize query efficiency and scalability. The task division module establishes the theoretical upper bound on the search space for each query subtask and employs an efficient method to achieve workload balance among machines. The vertex migration module focuses on migrating candidate vertices from the sketch graph to their respective machines, which helps maintain workload balance and reduces communication costs.

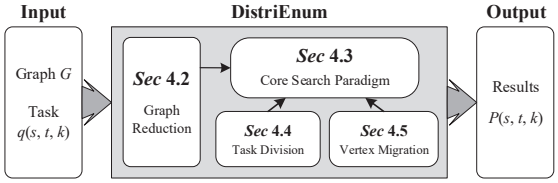


Figure 2: An overview of DistriEnum

4.2 Graph Reduction

Based on the analysis in "Challenge" of Sec. 1, the key to improving query efficiency is to reduce unnecessary computation caused by traversing redundant vertices and edges. R2M1 HybriEnum adopted a backpropagation mechanism to maintain the barrier levels of vertices, thereby avoiding falling into the same unnecessary exploration. However, it is inevitable for this strategy to explore the vertices which cannot satisfy the constraint of hop numbers, thus seriously impairing the query efficiency. Inspired by the lightweight index adopted in PathEnum, we propose a vertex-centric graph reduction method to generate a sketch graph $G^*(V^*, E^*)$ where each vertex $v \in V^*$ satisfies the constraint of hop number. The definition of this sketch graph is listed as follows.

Definition 1 (**Sketch Graph**). *Given a graph $G(V,E)$ and a query task $q(s,t,k)$, the sketch graph $G^*(V^*,E^*)$ is an induced subgraph of G, which satisfies:*

- $V^* \subseteq V$ *and* $E^* = E \cap (V^* \times V^*)$;
- $\forall v \in V^*$, *it satisfies* $v.s + v.t \le k$;
- $\forall u \in N(v, G^*)$, *it satisfies* $v.s + u.t \le k-1$;
- $\forall v \in V^*$, *each node* $u \in N(v, G^*)$ *is sorted in an ascending order by* $u.t$,

Here, $v.s$ and $v.t$ denote the minimal hop numbers from v to s and t, respectively. The individual components of the satisfied properties are explained below: (1) the first one ensures the sketch graph is an induced subgraph; (2) the second (resp., third) means that there is at least one walk via v (resp., $e(v,u)$), so v (resp., $e(v,u)$) should be included; and (3) the fourth enables an effective pruning strategy in Algorithm 3 (to be explained). R1O1 After constructing the sketch

graph, each hc-s-t path can be obtained when the corresponding walk does not contain any repeated vertices.

Algorithm 1 presents the details of generating the sketch graph. Given a query task $q(s,t,k)$, we first compute the minimal hop numbers of each vertex v to s and t (Line 1) by invoking the procedure `Compute()` (Lines 8-18) twice. `Compute(tgt)` performs the breadth-first search (BFS) via vertex-centric message passing. Initially, the target vertex tgt sends the current hop number to its neighbors (Lines 10-11) when $superstep$=0. During the next supersteps, each activated vertex executes three steps, including receiving messages, updating the minimal hop number, and transmitting messages to the neighbors (Lines 12-18). This procedure requires k iterations to finish at most. Next, we select the candidate vertices and edges based on their minimal hop numbers (Lines 3-5) and maintain a two-dimensional matrix Adj to construct the sketch graph G^* (Line 6).

Algorithm 1: `GraphReduction`

```
Input: G(V,E), q(s,t,k)
Output: G*(V*,E*)
1  Execute Compute(s) and Compute(t) to get v.s and v.t for v∈V, respectively
2  Initialize a two-dimensional array Adj
3  foreach v ∈ V with v.s + v.t ≤ k do
4      foreach u ∈ N(v,G) with v.s + u.t + 1 ≤ k do
5          Adj[v].push(u)
6      Sort Adj[v] based on a ascending order of u.t
7  Construct G* based on Adj and return it.
8
9  procedure Compute(vertex tgt)
10 Initial v.tgt←∞ for v∈V−{tgt}
11 if superstep = 0 then
12     Send msg to v for v∈N(tgt,G) with msg.dis ← 1
13 for superstep ∈ [1,k] do
14     foreach vertex v ∈ V do
15         foreach received message msg do
16             v.tgt←msg.dis with v.tgt > msg.dis
17         if v.tgt is changed and v ∉ {s,t} then
18             msg.dis ← v.tgt + 1
19             Send msg to u for u∈N(v,G)
```

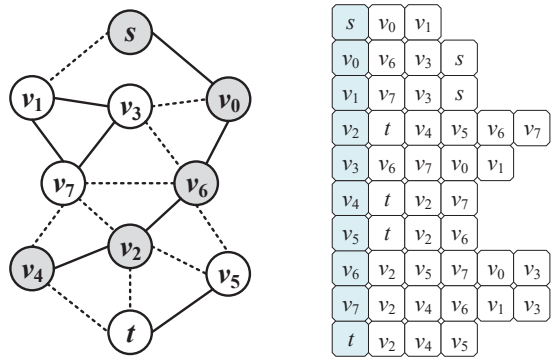
Time complexity. In the worst case, Algorithm 1 takes $O(|V|+|E|)$ time to construct the sketch graph, assuming that all vertices satisfy the hop number constraint.

Example 3. *Consider a query task $q(s,t,6)$ on the graph shown in Fig. 1. We illustrate the execution of Algorithm1 to generate the sketch graph.*

First, the algorithm computes the minimum hop numbers of all vertices to s (or t) and records the results in Table 2. Next, redundant vertices (v_8 to v_{13}) that violate the hop number constraint $v.s+v.t \le 6$ are ruled out. For each candidate vertex v, its neighbors u are sorted in ascending order based on u.t. The resulting adjacency structure, shown in Fig. 3(b), depicts the candidate vertices in blue and their corresponding sorted neighbors in white. Finally, the sketch graph G^* shown in Fig. 3(a) is generated based on this adjacency structure.

Table 2: A two dimensional matrix which records the minimal hop numbers of each vertex to s and t

Vertex	s	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	t
$v.s$	0	1	1	3	2	3	3	2	2	3	4	4	4	4	3	4
$v.t$	4	3	3	1	3	1	1	2	2	4	5	5	5	5	4	0



(a) (b)

Figure 3: (a) the sketch graph G^* of $q(s,t,6)$ and (b) the adjacency structure Adj of G^*.

4.3 Core Search Paradigm

After constructing the sketch graph G^*, one straightforward approach is to utilize IDX-DFS and IDX-JOIN from PathEnum to enumerate all simple paths. However, as introduced in Section 3.2, these two strategies have certain drawbacks that limit their applicability.

Algorithm 2: `PathCollect`

```
Input: G*, q(s,t,k)
Output: SP_s, SP_t, and SrcV
1  foreach v ∈ N(s,G*) do
2      if v = t then
3          Output the simple path ⟨s,t⟩
4      foreach u ∈ N(v,G*) with v ≠ t do
5          if u = t then
6              Output the simple path ⟨s,v,t⟩
7          else
8              Add the subpath ⟨s,v,u⟩ to SP_s
9              SrcV.push(u)
10 foreach v̄ ∈ N(t,G*) with v̄ ≠ s do
11     if v̄ ∈ SrcV and k ≥ 3 then
12         foreach ⟨s,v,v̄⟩ ∈ SP_s do
13             Output the simple path ⟨s,v,v̄,t⟩
14     Add the subpath ⟨v̄,t⟩ to SP_t
```

To alleviate these issues, we design a triple concatenation-based search paradigm, named `TCBSearch`, that combines the advantages of these two strategies to efficiently enumerate simple paths with reduced memory consumption. Given a query task $q(s,t,k)$ in the sketch graph G^*, each simple path is split into three parts: source subpaths (SP_s), backbone subpaths (SP_m), and target subpaths (SP_t). SP_s consists of subpaths from s to its 2-hop neighbors, while SP_t

contains subpaths from t to its 1-hop neighbors. The backbone subpaths $p_m(v_1, v_2) \in SP_m$ satisfy the following conditions: 1) v_1 is a 2-hop neighbor of s; 2) $v_2 \in N(t, G^*)$; and 3) $hop(p_m) \leq k-3$. All simple paths can be obtained by concatenating subpaths from these three parts.

Algorithm 3: TCBSearch

```
Input: G*, q(s,t,k), SP_s, SP_t, SrcV
Output: all simple paths between s and t
1  foreach v ∈ SrcV and k > 3 do
2      Initial a stack Stk to collect vertices
3      Search(Stk, v, k − 3)
4  ──────────────────────────────────────────
5  procedure Search(stack S, vertex u, hop number khop)
6  S.push(u) and v̄ ← S[0]
7  if p_t=⟨u,t⟩ ∈ SP_t then
8      foreach p_s ∈ SP_s with p_s ∩ S = {v̄} do
9          Output the simple path p(s,t) = p_s ∪ S ∪ p_t
10 foreach v̇ ∈ N(u, G*) do
11     if len(S) + v̇.t > khop then break;
12     if v̇ ∉ S and v̇ ≠ t then Search(S, v̇, khop) ;
13 u is unstacked from S
```

Specifically, given a query task $q(s, t, k)$, we first execute Algorithm 2 to establish SP_s and SP_t, and collect the 2-hop candidate neighbors of s (called $SrcV$). Next, all simple paths $p(s, t)$ with $hop(p) \leq 3$ can be directly obtained during the procedure PathCollect() in Algorithm 2.

- $hop(p)$=1. The simple path $\langle s, t\rangle$ is unique when satisfying $t \in N(s, G^*)$ (Lines 2-3).
- $hop(p)$=2. Each simple path $\langle s, v, t\rangle$ can be confirmed when satisfying $v \in N(s, G^*) \cap N(t, G^*)$ (Lines 4-6).
- $hop(p)$=3. We first confirm the middle vertex $\overline{v}$ with $\overline{v} \in SrcV \cap N(t, G^*)$ (Line 11). Then, each simple path $\langle s, v, \overline{v}, t\rangle$ can be obtained by concatenating $\langle s, v, \overline{v}\rangle$ and $\langle \overline{v}, t\rangle$ which are stored in SP_s and SP_t, respectively (Lines 12-13).

Algorithm 3 presents the pseudo-code of TCBSearch. After collecting SP_s and SP_t, for each vertex $v \in SrcV$, we first execute a DFS-based search to obtain the backbone subpaths (Lines 1-3). Then, we adopt a triple concatenation-based method to output the final results. The details are shown in the procedure of Search() in Algorithm 3. Similar to IDX-DFS, this search procedure can prune many redundant computations based on the reordered neighbors of candidate vertices (Line 11). Here, the stack Stk is established to avoid traversing the repeated vertices (Line 12). For each backbone subpath $p_m(v_1, v_2)$, the corresponding simple path $p(s, t)$ can be obtained by combining $p_s(s, v_1)$, $p_m(v_1, v_2)$, and $p_t(v_2, t)$, where $p_s \cap p_m = \{v_1\}$ and $p_m \cap p_t = \{v_2\}$ (Lines 7-9).

Table 3: The elements in SP_s, SP_t, and $SrcV$ of $q(s, t, 6)$

Symbols	Elements
SP_s	$\langle s, v_0, v_3\rangle, \langle s, v_0, v_6\rangle, \langle s, v_1, v_3\rangle, \langle s, v_1, v_7\rangle$
SP_t	$\langle v_2, t\rangle, \langle v_4, t\rangle, \langle v_5, t\rangle$
$SrcV$	v_3, v_6, v_7

Example 4. *Consider an HcPE query task $q(s, t, 6)$ in Fig. 3(a), where the adjacency structure of each vertex is shown in Fig. 3(b). As shown in Table 3, we first collect the subpaths in SP_s and SP_t, respectively. Then, given an activated vertex $v \in SrcV$, we execute the* Search() *procedure to get the corresponding backbone subpaths. Take the vertex v_7 as an example. Fig 4 lists a total number of 14 backbone paths from v_7 to the 1-hop neighbors of t which are v_2, v_4, and v_5. Furthermore, we execute a triple concatenation-based method for all backbone paths to output the final results. For example, according to concatenate $p_s = \langle s, v_1, v_7\rangle$, $p_m = \langle v_7, v_2\rangle$, and $p_t = \langle v_2, t\rangle$, we can obtain the simple path $p(s, t) = p_s \cup p_m \cup p_t = \langle s, v_1, v_7, v_2, t\rangle$.*

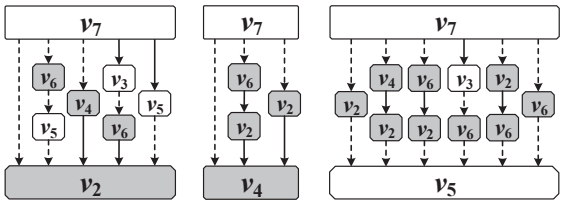


Figure 4: Backbone subpaths of the vertex v_7 of $q(s, t, 6)$ in G^*

Comparison with existing methods. TCBSearch offers three key advantages for path enumeration:

- *Reduced memory consumption:* TCBSearch only collects subpaths from SP_s and SP_t, and employs a DFS-based method to traverse subpaths from SP_m instead of storing them in advance. This significantly reduces memory overhead compared to IDX-JOIN.
- *Decreased time cost:* TCBSearch is faster than IDX-DFS since decreasing the hop number (from k to $k-3$) indeed prunes the traversal of shared subpaths. Moreover, for each backbone subpath $p_m(v_1, v_2)$ in SP_m, the time complexity of path concatenation can be optimized to $O(c)$, where $c = |SP_s[v_1]|$ is the number of simple paths from s to v_1. R2O2
- *Enhanced scalability:* Similar to IDX-JOIN, TCBSearch divides the query task into a series of finer-grained independent subtasks. This division allows for better load balancing and resource utilization in the distributed environment, leading to enhanced scalability.

Example 5. *Take the query $q(s, t, 6)$ in Fig. 3 (a) as an example. The performance comparison of three methods is listed as follows.*

Memory consumption. *To resolve this query, TCBSearch collects 7 subpaths in SP_s and SP_t. By contrast, IDX-JOIN needs to get a total number of 46 subpaths for path concatenation, and IDX-DFS only maintains a queue whose length is no more than 6. Therefore, the memory cost of TCBSearch is less than that of IDX-JOIN whilst larger than that of IDX-DFS.* R2M3

Time cost. *IDX-DFS needs to repeatedly traverses 7 subpaths from v_3 to t to obtain the simple paths originating from the source subpaths $\langle s, v_0, v_3\rangle$ and $\langle s, v_1, v_3\rangle$. By contrast, repeated path traversal can be avoided in DistriEnum by executing path concatenation. Note that, the time complexity of path concatenation of each source subpath in TCBSearch is $O(1)$, whilst the complexity of that in IDX-JOIN is $O(k)$.*

Scalability. *TCBSearch can simultaneously activate the search processes originated from the subpaths in SP_s. Similarly, the path*

concatenation in DFS-Join can also be executed in parallel. By contrast, it is difficult for IDX-DFS to be performed in parallel, thus impairing the scalability.

Time complexity. In the worst case, Algorithm 2 takes $O(|E^*|)$ to generate a series of auxiliary structures. Algorithm 3 takes $\sum_{v \in SrcV} |SP_s[v]| \cdot |SP_m[v]|$ time to get the simple paths. Here, $|SP_s[v]|$ and $|SP_m[v]|$ are the numbers of source and backbone subpaths, respectively.

4.4 Task division

While the first two modules offer effective pruning strategies to improve query efficiency, there is a challenge when it comes to distributing the dynamically confirmed candidate vertices across multiple machines. As a result, workload imbalance may occur, leading to uneven processing times. To address this issue, we first analyze the maximum potential search space of each candidate vertex under various hop numbers, relaxing the non-duplicate requirement. Then, we design a model to assess the upper-bound workload of each subtask, enabling us to estimate the computational effort required for processing. Finally, we develop an efficient strategy to divide tasks, ensuring workload balance and providing a theoretical guarantee. In addition, the vertex migration strategy can reduce the cutting edges caused by the skewed distribution of
R3O1 data graphs in vertex-centric distributed systems, thus reducing communication cost. Here, we use the upper bound of the path count as the estimated measure of the search space. According to the following lemma, we can use the walk count as the upper bound of the path count.

Lemma 1 (Upper Bound of Path Count). *Given a task $q(s,t,k)$, the k-hop-constrained path count $|P(s,t,k,G^*)|$ can be upper bounded by $\sum_{1 \le i \le k} \sum_{u \in N(s,G^*)} |\overline{W}(u,t,i-1)|$, where $\overline{W}(v,t,i)$ denotes the set of walks with $hop(w(v,t)) = i$.*

Proof. Assuming that $W(s,t,k,G^*) = \{w(s,t) | hop(w) \le k\}$ and $P(s,t,k,G^*) = \{p(s,t) | hop(p) \le k\}$ are the sets of walks and simple paths based on k and G^*, respectively. Since $p(s,t)$ is a special case of $w(s,t)$, we have $P(s,t,k,G^*) \subseteq W(s,t,k,G^*)$ such that $|P(s,t,k,G^*)| \le |W(s,t,k,G^*)|$. Given a candidate vertex v, we have $|W(v,t,k,G^*)| = \sum_{i \in [0,k]} |\overline{W}(v,t,i)|$ where $\overline{W}(v,t,i)$ is denoted as the set of walks with $hop(w(v,t)) = i$. Therefore, the upper bound of $|P(s,t,k,G^*)|$ can be computed by

$$\begin{aligned} |P(s,t,k,G^*)| \le |W(s,t,k,G^*)| &= \sum_{0 \le i \le k} |\overline{W}(s,t,i)| \\ &= \sum_{1 \le i \le k} \sum_{u \in N(s,G^*)} |\overline{W}(u,t,i-1)| \quad (1) \\ \text{s.t. } |\overline{W}(t,t,0)| &= 1. \end{aligned}$$

□

Referring [15], a dynamic programming-based method is pro-
R2M4 posed based on Equation 1 to obtain the walk count, i.e., the upper bound of the path count. The state transition equation can be expressed by $VS(v,i) = \sum_{u \in N(v,G^*)} VS(u,i-1)$, where $VS(v,i)$ denotes the walk count $|\overline{W}(v,t,i)|$, i.e., the walk count from v to t with i hops. The pseudo-code is presented in Algorithm 4. Based on the state transition equation, we can gradually compute the walk count of each candidate vertex on different hop numbers (Lines 2-5). It is noted that the redundant computation of search space can be ruled out by introducing distance judgment (Line 3) and avoiding vertex repetition (Line 4).

Algorithm 4: `SearchSpace`

```
Input: G*, q(s,t,k)
Output: VS
1 Initialize VS[t][0]←1 and VS[v][1]←1 with v ∈ N(t,G*)
2 foreach d∈[2,k] do
3     foreach v∈V*(G) with v.s + d ≤ k do
4         foreach vertex u ∈ N(v,G*) with u ∉ {s,t} do
5             VS[v][d]←VS[v][d] + VS[u][d − 1]
6 return VS
```

Table 4 records the walk counts (denoting search spaces) of all candidate vertices for $q(s,t,6)$ in Fig. 1. Firstly, we have $VS[v_2][1]$=1, $VS[v_4][1]$=1, and $VS[v_5][1]$=1 since v_2, v_4, and v_5 are the neighbors of t. Then, the walk counts of other candidate vertices can be gradually obtained based on the accumulation of walk counts of neighbors. Finally, each value in VS represents the walk count of the candidate vertex in a corresponding hop number. For example, $VS[s][6]$=70 represents that the upper bound of path count from s to t is 70 when $|p(s,t)| = 6$.

Lemma 2 (Workload Model). *Given the walk count array* VS *and the set of source paths SP_s, for each vertex $v \in SrcV$, the workload of v is $W_v = \sum_{i \in [0,k-2]} VS[v][i] \cdot |SP_s[v]|$, where $|SP_s[v]|$ is the number of source paths between s and v.*

Proof. Based on the `TCBSearch` method, for each vertex $v \in SrcV$, we aim to explore all backbone paths from v and execute a join-oriented method to obtain the corresponding simple paths. Assuming that W_v denotes the workload of v, we have

$$\begin{aligned} W_v = |SP_m[v]| \cdot |SP_s[v]| &\le |W(v,t,k-2,G^*)| \cdot |SP_s[v]| \\ &= \sum_{i \in [0,k-2]} VS[v][i] \cdot |SP_s[v]|, \quad (2) \end{aligned}$$

where $|SP_s[v]|$ is the number of source paths between s and v. □

After obtaining the walk count array VS, we employ an efficient task division strategy to achieve workload balance. The pseudo-code is presented in Algorithm 5. Firstly, each candidate vertex $v \in SrcV$ is decomposed to a series of subtasks $p(v,\overline{v})$ whose workloads are less than $W_{avg} = \sum_{i \in [0,k]} VS[s][i]/c$, where c is the number of machines (Lines 1-6). During the task decomposition, a

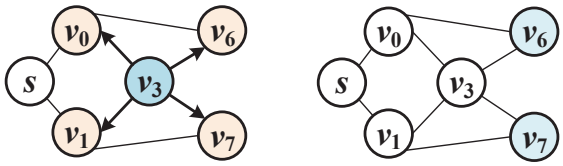
Table 4: A two-dimensional array *VS* of $q(s,t,6)$

hop	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	s	t
0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	1	1	0	0	0	0
2	0	0	2	0	1	1	2	2	0	0
3	2	2	6	4	4	4	5	5	0	0
4	9	9	0	14	0	0	21	21	4	0
5	35	35	0	0	0	0	0	0	18	0
6	0	0	0	0	0	0	0	0	70	0

forward search is executed to obtain the corresponding subpaths $p(v,\overline{v})$ until the workload constraint is satisfied. The workload of $p(v,\overline{v})$ is expressed as $\sum_{i\in[0,k-2-hop(p)]} VS[\overline{v}][i]\cdot|SP_s[v]|$. This strategy helps mitigate the impact of significant workload differences among candidate vertices. A workload list W is established by collecting the corresponding search space of each subtask, denoted as $p(v,\overline{v})\in SrcV^*$ (Lines 7-10). Finally, the subtasks are assigned to machines based on their workload, one-by-one, with preference given to machines with minimal workload (Lines 11-15).

Algorithm 5: `TaskDivision`

Input: *SrcV, VS, c*
Output: *TList*
1 Initialize $SrcV^*$ and $W_{avg} \leftarrow \sum_{i\in[0,k]} VS[s][i]/c$
2 **foreach** $v \in SrcV$ *with* $p=\{v\}$ **do**
3 Obtain W_v of v via Equation 2
4 **if** $W_v > W_{avg}$ **then**
5 Get all subtasks $p(v,\overline{v})$ with $W_{\overline{v}} < W_{avg}$
6 $SrcV^*.push(p)$
7 Initialize W and *TList*
8 **foreach** *element* $p(v,\overline{v}) \in SrcV^*$ **do**
9 Initialize $e \leftarrow \langle VS[\overline{v}][k-2-hop(p)], p(v,\overline{v})\rangle$
10 $W.push(e)$
11 Sort W ▷ the descending order
12 **foreach** *element* $e \in W$ **do**
13 Select the i-th machine with the minimal workload
14 $p(v,\overline{v}) \leftarrow e.second$
15 $TList[i].push(p)$
16 **return** *TList*



(a) Task decomposition of v_3 (b) Task decomposition of v_6 and v_7

Figure 5: The process of task decomposition with $c=3$. Here, each candidate vertex $v \in SrcV$ is represented by bule while orange stands for the tail node of each subtask.

Example 6. *Take the query task $q(s,t,6)$ in Fig. 3(a) as an example, where the number of machines is 2 and $SrcV=\{v_7,v_6,v_3\}$. Based on Equation 2, the workload of v_7 is $W_{v_7}=(VS[v_7][4]+VS[v_7][3]+VS[v_7][2])\cdot|SP_s[v_7]|=28$. Similarly, we have $W_{v_6}=28$ and $W_{v_3}=36$. Considering that W_{v_6} and W_{v_3} are less than $W_{avg}=(VS[s][6]+VS[s][5]+VS[s][4])/2=46$, the final results of Algorithm 5 are $TList[0]=\{v_7,v_6\}$ and $TList[1]=\{v_3\}$.*

Reconsider the query task $q(s,t,6)$ with the machine number as 3 and $W_{avg}=31$. Fig. 5 illustrates the corresponding task division process. In this case, only the vertex v_3 is involved in the decomposition because the workloads of v_6 and v_7 are already less than 31. Based on a forward exploration, we obtain 4 subtasks originated from v_3 which are $p_1=\langle v_3,v_0\rangle$, $p_2=\langle v_3,v_1\rangle$, $p_3=\langle v_3,v_6\rangle$, and $p_4=\langle v_3,v_7\rangle$. Assuming that $p_5=\langle v_6\rangle$ and $p_6=\langle v_7\rangle$, the final task allocation is as follows: $TList[0]=\{p_6,p_2\}$, $TList[1]=\{p_5,p_1\}$, and $TList[2]=\{p_3,p_4\}$. The total workloads of the three machines are 32, 32, and 28, respectively, achieving a good workload balance.

Time complexity. In the worst case, Algorithm 4 takes $O(k\cdot|E^*|)$ time to construct the two-dimensional walk count array VS. Similarly, Algorithm 5 takes $O(|V^*|)$ time to construct the task array *TList*. Hence, the overhead for performing the task division is very low.

4.5 Vertex migration

To minimize memory consumption, `TCBSearch` utilizes a DFS-based search strategy to traverse backbone paths. This approach requires only $O(k)$ space to maintain the current backbone paths, resulting in a reduced memory footprint compared to IDX-JOIN. However, `TCBSearch` has two disadvantages:

- *Uneven distribution of candidate vertices.* The dynamic generation of candidate vertices based on the query task makes it challenging to evenly distribute these vertices among machines. This imbalance can negatively impact query performance.
- *Production of intermediate results.* Although `TCBSearch` reduces the number of hops in path traversal, it may still require message exchange between different machines, resulting in a significant number of intermediate results.

Example 7. *As shown in Fig 4, the dashed lines connecting any two vertices represent the process of exchanging messages between different machines. There are 22 messages exchanged during the traversal of backbone paths originating from the vertex v_7.*

To address these issues, we design a vertex migration strategy to optimize the communication cost and reduce the production of intermediate results.

Algorithm 6: `VertexMigration`

1 **foreach** $v \in V^*-\{s,t\}$ **do**
2 **if** $v.s=1$ *with* $v \notin SrcV$ **then**
3 **if** $\exists u_1,u_2\in N(v,G^*)$ *with* $u_1,u_2\notin\{s,t\}$ **then**
4 **if** $u_1.s+2+u_2.t \le k$ **then**
5 Duplicate v to the machine of u_1
6 **else if** $v \in SrcV$ **then**
7 Initialize $hop \leftarrow 1$
8 **foreach** $u \in SrcV-\{v\}$ **do**
9 $hop \leftarrow 2$ with $u \notin N(v,G^*)$
10 **if** $2+hop+v.t \le k$ **then**
11 Duplicate v to the machine of u
12 **else if** $v.s>2$ **then**
13 Duplicate v to the machine of u with $u \in SrcV$

Algorithm 6 presents the pseudo-code of the vertex migration strategy, which can be classified into the following three cases:

(1) $v \notin SrcV$ with $v.s=1$. The vertex v is duplicated to the machine of u_1 when v is possibly located in a walk from u_1 to t (Lines 2-5).

(2) $v \in SrcV$. The vertex v is duplicated to the machine of u if it satisfies $2+hop+v.t \leq k$ (Lines 6-11), where hop is the minimal distance between u and v. To reduce time cost, hop is set as 2 if $u \notin N(v, G^*)$ (Line 9).
(3) $v.s > 2$. It is time-consuming to make massive judgments between the rest of the vertices and $SrcV$. Therefore, the rest of the vertices are duplicated to all machines to reduce time cost (Lines 12-13).

Example 8. *Given a query $q(s,t,6)$, Table 5 lists all migrated vertices for each candidate vertex $v \in SrcV$. Specifically, the vertex v_0 is moved to the machine of v_3 since $v_3.s+2+v_6.t = 6$, where v_3 and v_6 are the neighbors of v_0. Similarly, the vertex v_1 is also migrated to the machine of v_3. In addition, the vertex v_3 is migrated to the machine of v_6 (v_7) since $v_6.s + 1 + v_3.t = 6$ ($v_7.s + 1 + v_3.t = 6$) which means that two vertices v_3 and v_6 (v_7) are located in a walk $w(s,t)$. Finally, the rest of the vertices v_2, v_4, and v_5 are migrated to all machines.*

Table 5: The migrated vertices which correspond to three cases in Algorithm 6

	v_3	v_6	v_7
case 1	v_0, v_1	$\emptyset$	$\emptyset$
case 2	v_6, v_7	v_3, v_7	v_3, v_6
case 3		v_2, v_4, v_5	

Although the vertex migration strategy incurs additional memory consumption to store these vertices, it efficiently avoids the generation of intermediate results and optimizes the communication cost, leading to improved query performance.
Time complexity. In the worst case, Algorithm 6 takes $O(|V^*|)$ time to complete the migration of all candidate vertices in G^*.

4.6 Overall algorithm

Combining the above modules, our distributed path enumeration approach, **DistEnum**, is shown in Algorithm 7. Given a query task $q(s,t,k)$, Algorithm 7 first constructs the sketch graph G^* to rule out all redundant vertices (Line 1). Then, a series of auxiliary structures, including SP_s, SP_t, and $SrcV$, is generated to support the following procedures (Line 2). Next, the subtasks in $SrcV$ are decomposed, and evenly distributed to all machines (Line 4) based on the pre-computed search space array VS (Line 3). Furthermore, we adopt the vertex migration strategy to reduce the production of intermediate results and optimize the communication cost (Line 5). Finally, the procedure `TCBSearch` is executed in all machines in parallel to output all simple paths.

R3M6 **Proof of correctness.** Given a query task $q(s,t,k)$, the query results of DistriEnum consist of two parts: (1) all simple paths P_1 whose hop numbers are no more than 3. This part can be obtained based on Algorithm 2; (2) the rest of paths P_2 whose hop numbers are between 4 and k. Based on the analysis of `TCBSearch`, we have $P_2 = \sum_{v \in SrcV} \sum_{p_1 \in SP_s[v]} \sum_{p_2 \in SP_m[v]} p_1 \cup p_2$ with $p_1 \cap p_2 = \{v\}$. Therefore, the query result $P(s,t,k,G) = P_1 \cup P_2$ is complete.
Space Complexity. Assuming that the number of machines is c, for a given query task $q(s,t,k)$, the memory consumption mainly consists of three parts: (1) the memory used for the sketch graph G^*, whose space cost can be represented as $O(c \cdot |G^*|)$; (2) the memory consumption of `TCBSearch` concentrates on maintaining the vertices on the current traversal path, whose space cost is $O(c \cdot k)$; (3) the constant memory overhead of auxiliary structures, whose space cost is $O(\Delta) = O(|SP_s|+|SP_t|+|SrcV|+|TList|)$. Therefore, the total memory consumption is $O(c \cdot |G^*| + \Delta)$. By contrast, the space cost of IDX-DFS is $O(c \cdot |G^*| + \Delta^*)$ where Δ^* is the intermediate result size induced by message exchange. Similarly, IDX-JOIN takes R2O2 $O(c \cdot |G^*| + \Delta^{\#})$ space where $\Delta^{\#}$ is the size sum of intermediate results and subpaths. Here, the reduced graph size $|G^*|$ is similar among the three methods. However, our Δ is much less than Δ^* and $\Delta^{\#}$, especially for the queries with large-scale search spaces.
Time Complexity. Based on the time complexity analysis of the above modules, for a given query task $q(s,t,k)$, Algorithm 7 takes $O(|V|+|E|+\sum_{v \in SrcV} |SP_s[v]| \cdot |SP_m[v]| + k \cdot |E^*| + |V^*| + |V^*|)$ which can be simplified as $O(\sum_{v \in SrcV} |SP_s[v]| \cdot |SP_m[v]|)$.
Message complexity. The message complexity of DistriEnum is $O(c \cdot |G^*|)$ which are produced by constructing the sketch graph and vertex migration. By contrast, the communication costs of IDX-DFS and HybridEnum are $O(|P(s,t,k,|G^*|)|)$ in the worst R2O2 case. In addition, IDX-JOIN takes $O(\sum_{v \in V_M} (|P(v,s,\lceil k/2 \rceil, |G^*|)| + |P(v,t,\lfloor k/2 \rfloor, |G^*|)|)$, where V_M is a set of middle vertex. Clearly, DistriEnum exhibits the lowest communication cost, especially when dealing with queries spanning large-scale search spaces.

Algorithm 7: DistriEnum

Input: G, $q(s,t,k)$
Output: all simple paths between s and t
1 $G^* \leftarrow$ `GraphReduction(G, q)` ▷ Get the sketch graph by Algorithm 1 in Section 4.2
2 SP_s, SP_t, $SrcV \leftarrow$ `PathCollect(G*, q)` ▷ Collect auxiliary structures by Algorithm 2 in Section 4.3
3 $VS \leftarrow$ `SearchSpace(G*, q)` ▷ Confirm the upper bound of search space by Algorithm 4 in Section 4.4
4 $TList \leftarrow$ `TaskDivision(SrcV, VS)` ▷ Divide tasks evenly by Algorithm 5 in Section 4.4
5 Execute `VertexMigration()` ▷ Migrate candidate vertices by Algorithm 6 in Section 4.5
6 Execute `TCBSearch(G*, q, SPs, SPt, TList)` ▷ Output all simple paths by Algorithm 3 in Section 4.3

5 EXPERIMENTS

In this part, we conduct extensive experiments to evaluate the performance of our methods. Section 5.1 introduces the setup of our experiments, followed by the experimental results in Section 5.2.

5.1 Experimental Setup

Datasets. In the experiments, we employ 10 real-life datasets (Table 6) including social networks and web graphs that are downloaded from Stanford Large Network data set Collection[1] and Network Repository[2]. All directed data graphs have been converted to undirected graphs.
Algorithms. In this part, we compare the following algorithms:

- **HybridEnum [14]**. The state-of-the-art distributed method based on a join-oriented method.

[1] http://snap.stanford.edu/data/
[2] http://networkrepository.com/index.php

Table 6: Statistic of Real-world Graphs

Alias	Dataset	$\|V\|$	$\|E\|$	Type
TK	WikiTalk	2M	5M	Miscellaneous
DP	DBpedia	4M	14M	Miscellaneous
SP	SocPokec	0.6M	12M	Social Network
SJ	SocLiveJ	4.8M	42.8M	Social Network
UK	UK2005	39.4M	783.1M	Web Graph
WB	Webbase	0.1B	725.4M	Web Graph
IT	IT2004	41.3M	1.03B	Social Network
TW	Twitter	52.6M	1.96B	Social Network
SK	SK2005	50.6M	1.81B	Web Graph
FD	Friender	65.6M	1.80B	Web Graph

- **IDX-DFS [29]**. The distributed extension of the DFS-based method.
- **IDX-JOIN [29]**. The distributed extension of the join-oriented method.
- **DistriEnum**. Our distributed simple path enumeration method in Algorithm 7.

Distributed extensions of IDX-DFS and IDX-JOIN. Given a query task $q(s, t, k)$, the lightweight index is first built based on a vertex-centric computing paradigm and stored in the corresponding machines. Secondly, the DFS-based search process is activated from s to obtain the visited queues which represent the simple paths. Thirdly, when finishing the search process in the current machine, these queues are exchanged to other machines where the next candidate vertices are placed. The last two steps are repeated until all simple paths are obtained. Similarly, in IDX-JOIN, the indexes are first distributed on each machine and the middile vertices are confirmed. Then, the DFS-based search process is activated from the middle vertex to collect subpaths. Finally, the concatenation of subpaths executes in parallel.

Note that, the search on the subgraph in each machine and the process of path concatenation are executed in parallel. In addition, the search process across different machines inevitably involves the data exchange between machines.

Environment. All algorithms are implemented in the distributed graph computing system Blogel [32] which performs computational tasks in a superstep fashion. Blogel is deployed in a local cluster with 10 computing nodes with AMD 2.6 GHz and 64 GB memory. The communication among machines is achieved by MPI. All algorithms are achieved by C++ and compiled with O3-level optimization.

Setting. We generate two disjoint sets V^* and $V^\#$ based on the vertex degrees: (a) V^* is the set of vertices within top 10% in the descending order of their degrees; and (b) $V^\# = V - V^*$. Then, we have three settings according to the locations of s and t: $\{V^*, V^\#\} \times \{V^*, V^\#\}$. For each setting, we generate 100 queries by choosing s and t uniformly at random. The processing time is set as INF and OM when an algorithm cannot finish in 10^5 seconds or runs out of memory, respectively. Without specifying, we set the default hop constraint k as 6.

5.2 Experimental results

Exp-1: Efficiency and memory consumption on different datasets. In this part, we evaluate the efficiency of all algorithms on all datasets.

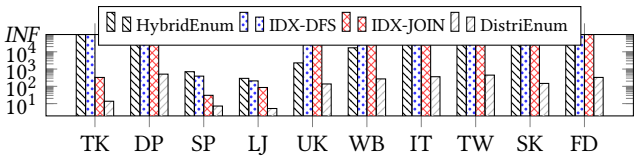


Figure 6: Processing time (s) on all datasets with $k = 6$

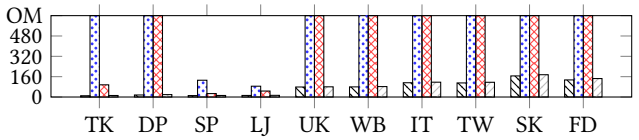


Figure 7: Memory consumption (GB) on all datasets

As shown in Fig. 6, DistriEnum consistently outperforms the other distributed methods in terms of efficiency. On many datasets, DistriEnum achieves up to three orders of magnitude faster execution times compared to HybridEnum. The superior performance of DistriEnum can be attributed to its effective pruning strategies, which eliminate redundant vertices and significantly reduce the traversal of shared subpaths based on TCBSearch. Additionally, the task division strategy employed by DistriEnum maximizes the utilization of computing resources, further enhancing query efficiency. In contrast, HybridEnum struggles to complete query tasks on large-scale graphs within a reasonable time limit. This is because HybridEnum involves traversing redundant vertices and exchanging messages during the enumeration process, leading to increased execution times. Furthermore, DistriEnum demonstrates significant speed improvements compared to IDX-DFS and IDX-JOIN on the LJ graph, achieving speedups of 41.8× and 18.5×, respectively.

Another important observation is that both IDX-DFS and IDX-JOIN encounter out-of-memory issues when dealing with query tasks on large-scale graphs with a large value of k. This is primarily due to the intermediate results generated during the enumeration process. Additionally, IDX-JOIN requires significant space overhead to store a large number of subpaths, further exacerbating memory consumption. As shown in Fig. 7, both IDX-DFS and IDX-JOIN run out of memory on all datasets except SP and LJ. In contrast, HybridEnum can mitigate the memory overhead by adjusting the number of activated subtasks. However, DistriEnum excels in reducing the storage requirements for subpaths and intermediate results through its `TCBSearch` paradigm and vertex migration strategy, respectively. Hence, it is easy for DistriEnum to handle large-scale graphs.

Exp-2: Efficiency with varying hop constraint k. In this experiment, we evaluate the efficiency of the algorithms as the hop constraint k varies from 3 to 7. The average processing time for each query is depicted in Fig. 8.

As the hop constraint k increases, the processing time of all methods also increases due to the larger search spaces of the query tasks. However, DistriEnum consistently outperforms the other distributed algorithms. In general, as k increases, HybridEnum struggles to complete the query tasks within a reasonable time. This is because fruitless exploration cannot be fully avoided by the pruning techniques in HybridEnum, leading to significant time costs. Furthermore, as k increases, both IDX-DFS and IDX-JOIN face severe memory consumption issues. This is because these methods require substantial space overhead to store intermediate results during the

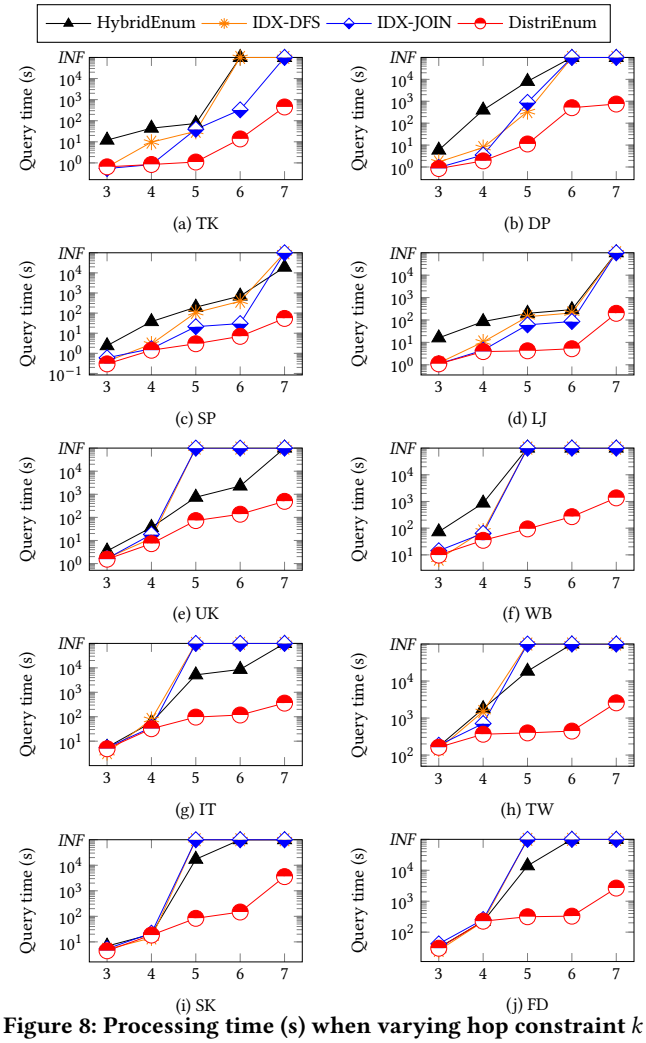


Figure 8: Processing time (s) when varying hop constraint k

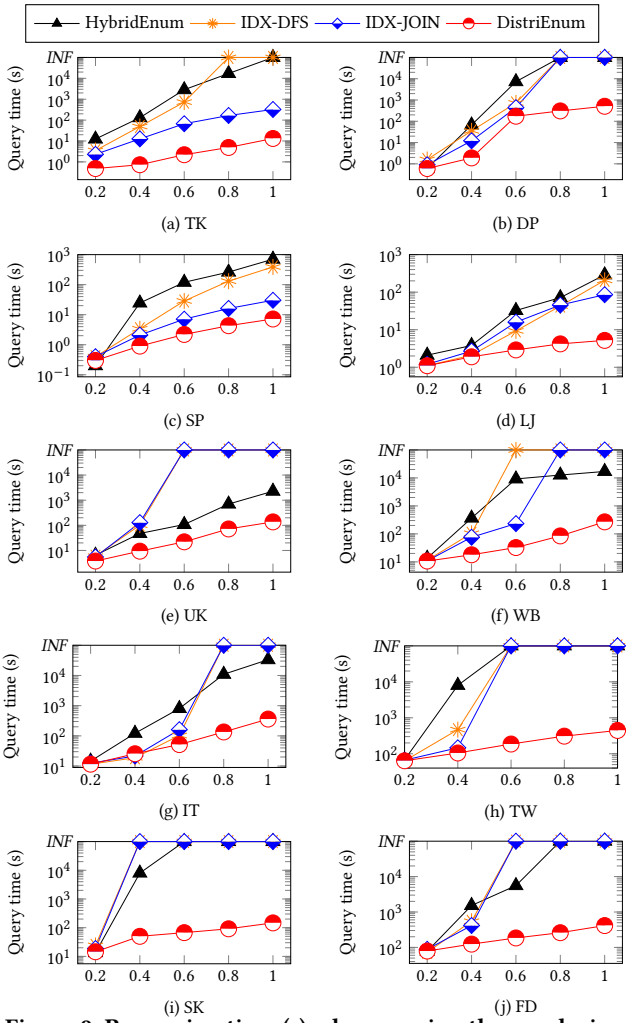


Figure 9: Processing time (s) when varying the graph size

message exchange process. In contrast, DistriEnum avoids memory crises and efficiently completes the enumeration on all graphs. This is attributed to several factors: (1) DistriEnum effectively prunes exploration and optimizes resource utilization through the task division strategy, (2) DistriEnum employs `TCBSearch` to reduce the storage of subpaths, mitigating excessive memory usage, and (3) DistriEnum minimizes the production of intermediate results by utilizing the vertex migration strategy.

Exp-3: Efficiency with varying graph size. In this experiment, we evaluate the efficiency of all algorithms as the graph size increases from 20% to 100%. We randomly generate several queries for each graph and measure the average processing time with a hop constraint of $k = 6$.

As shown in Fig 9, the processing time of all the methods increases with the increase in the graph size. The main reason is that the actual workload of each query task increases. As a result, it takes more time for the above four methods to obtain all simple paths. We also observe that DistriEnum outperforms the other distributed algorithms in almost all cases, and the performance gap increases as the graph size increases. For example, on graphs WB and SK, DistriEnum is up to 279.9× and 3 orders of magnitude faster than that of HybridEnum in terms of query time, respectively. Moreover, DistriEnum can also finish the enumeration efficiently on all graphs. This is because DistriEnum can largely reduce explorations and efficiently utilize all computing resources. In addition, it is difficult for IDX-DFS and IDX-JOIN to handle query tasks in large-scale graphs due to the substantial memory consumption.

Exp-5: Communication cost. In this experiment, we evaluate the communication cost of HybridEnum and DistriEnum on all datasets. Note that the experimental results in some datasets are not listed since these algorithms cannot finish within the given time limit.

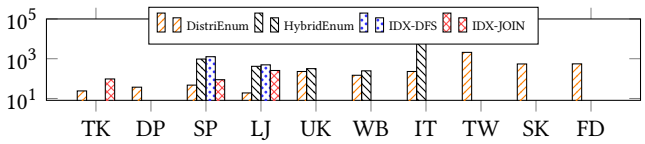


Figure 10: The communication cost (MB) on all datasets

As shown in Figure 10, DistriEnum always outperforms HybridEnum. Specifically, DistriEnum is 13.8× (resp. 26.6×, and 5.2×) less than HybridEnum (resp. IDX-DFS and IDX-JOIN) in terms of communication cost. The communication cost of DistriEnum mainly concentrates on constructing the sketch graph and executing task division. In contrast, HybridEnum needs to frequently exchange messages during path traversal, thereby generating serious communication overhead.

Exp-5: Scalability with varying #partitions. In this experiment, we assess the scalability of DistriEnum by varying the number of partitions from 10 to 60. We randomly generate several queries whose the predicted search spaces ($VS[s][k]$) are more than 10^{12}. In this setting, the other three algorithms cannot finish the tasks within a reasonable time. The average processing time of DistriEnum is presented in Fig 11.

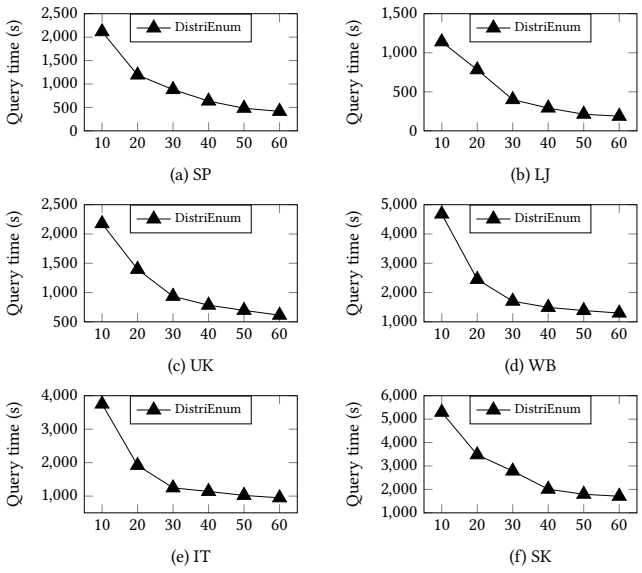


Figure 11: Processing time (s) when varying the partitions

As shown in Fig 11, the processing time of DistriEnum decreases as the number of partitions increases. The main reason is that DistriEnum adopts the task division strategy to divide each query task based on the workload distribution and the number of partitions, which helps to keep the workload balance and constrain the number of subtasks.

Exp-6: Effectiveness of task division In Exp-6, we evaluate the effectiveness of the task division module by comparing it with the hash method, where each query task is assigned to the computing node where the source vertex is located. We refer to these two methods as "Task" and "Hash", respectively. Note that, for a majority of the queries in our experiments, the ratio between the actual measured load and our projected upper bound exceeds 0.6, which helps to optimize the task division strategy. R2M2

In Fig 12, "Task" outperforms "Hash" with a 3.92× improvement in average query time. Furthermore, the query time of "Task" decreases with an increase in the number of cores. This suggests that the task division mechanism effectively balances the workload among the cores, resulting in improved query efficiency. Although the task division step incurs additional time, it is beneficial in reducing the overall processing time by maintaining workload balance.

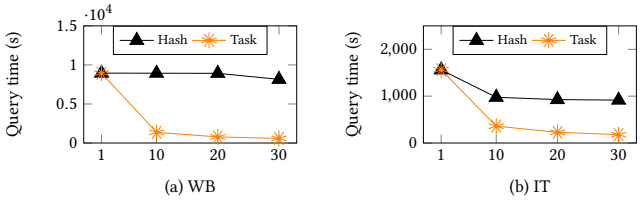


Figure 12: Processing time (s) when varying the cores

In Fig 13, we observe that both "Task" and "Hash" experience an increase in memory consumption with an increase in the number of cores. This is due to the data structures maintained by Blogel for each core to support computation. Additionally, the memory consumption of "Task" is slightly higher than that of "Hash" because "Task" generates a number of query subtasks that is not less than that of "Hash". Despite this, both strategies demonstrate consistent performance in terms of memory usage.

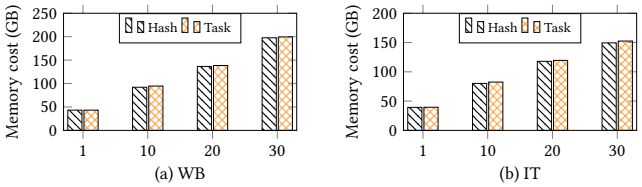


Figure 13: Memory consumption (GB) when varying the cores

Exp-7: Effectiveness of vertex migration In this experiment, we evaluate the effectiveness of the vertex migration module. Here, "DComm" denotes the DistriEnum method without the vertex migration module. As shown in Fig 14, DistriEnum is 15.4× faster than "DComm" in terms of processing time. This is because the vertex migration module can keep the balance distribution of candidate vertices to improve the query efficiency. Moreover, the memory overhead of "DComm" is 23.2% higher than that of DistriEnum. Compared to "DComm", DistriEnum can avoid the production of intermediate results while spending little extra space to store the migrated vertices.

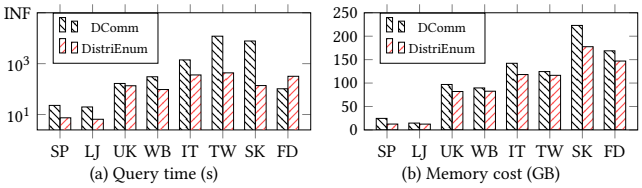


Figure 14: The performance evaluation on all datasets

Exp-8: Single machine comparison. In this experiment, we compare the performance of all methods on a single machine, where the data graph is stored in the same memory space. The results in Figure 15 show that DistriEnum outperforms the other methods by up to 2 orders of magnitude (resp. 6.48x and 2.95x faster than HybridEnum, IDX-DFS, and IDX-JOIN) in terms of processing

time. This significant improvement is attributed to DistriEnum's ability to reduce search spaces and optimize the path traversal and combination through TCBSearch. Moreover, the processing time of IDX-DFS and IDX-JOIN is lower than that of HybridEnum, as they avoid producing intermediate results during enumeration, mitigating memory constraints. Additionally, these methods employ effective index schemes to eliminate redundant vertices and prune exploration.

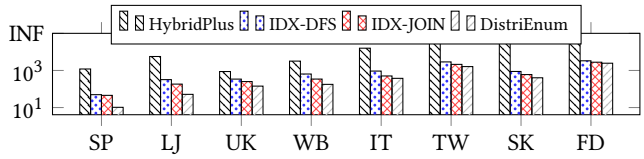


Figure 15: The processing time (s) in a single machine

Exp-9: Effectiveness of partitioning methods. In this experiment, we assess the effectiveness of different partitioning methods (Hash and KaHIP [26]) on query performance. KaHIP is known for generating vertex-balanced partitions with minimal cutting edges.

As shown in Table 7, we observe that the communication cost of "Graph Reduction" is reduced when adopting the KaHIp strategy whilst the time cost is increased. This is because "Graph Reduction" mainly consists of two steps, where the first step is computing the minimal hop numbers of each candidate node to s and t (Line 1 in Algorithm 1), and the second step is sorting the neighbors of candidate vertices (Lines 3-6 in Algorithm 1). Therefore, KaHIP can take less communication cost in the first step based on a good locality partitioning result. However, it also takes more time to execute the second step since the candidate vertices are located in fewer computing nodes, and DistriEnum cannot exploit the computing resources on other servers. Considering that it is time-consuming to get a good initial partition, our method is orthogonal to the partitioning method and actually more applicable to handle large-scale graphs. R3O1

Table 7: Performance evaluation of "Graph Reduction"

(a) LJ

	Time cost		Comm cost	
	Hash	KaHIP	Hash	KaHIP
3	1.8	2	1.8	0.8
4	2	3	6	4
5	2.5	3.5	7	5
6	3.2	5.1	9	6
7	5.4	6.8	12	8

(b) IT

	Time cost		Comm cost	
	Hash	KaHIP	Hash	KaHIP
3	4.6	5.4	2.8	2
4	18.1	19.2	75	50
5	34.2	37.6	155	98
6	40.1	46.4	188	116
7	42.8	51.2	191	118

Fig. 16 (a) and (b) reveal that the query efficiency of DistriEnum is slightly affected by the initial partitioning results. This is because the migration of candidate vertices and query tasks are not affected by the initial partitioning results. By contrast, the communication cost can be reduced by 25.2% on average when adopting KaHIP. This is because the partitioning result with high locality optimizes the communication cost incurred during the construction of sketch graphs.

Exp-10: Effectiveness of hop limits. Regarding the different hop limits of SP_s and SP_t, we clarify that the balance of storage costs and query efficiency is a key motivation. Higher hop limits increase storage requirements, reduce the number of backbone paths, and increase subpath concatenation time. Figs. 17 (a) and (b) show the experimental results on query time and memory cost of DistriEnum with four combinations respectively, which are $(0, 0)$, $(2, 1)$, $(2, 2)$, and $(\lceil k/2 \rceil, \lfloor k/2 \rfloor)$. We can find that the $(2, 1)$ setting gives the shortest query time in most cases, and takes relatively low memory costs. Hence, the current $(2, 1)$ setting is the result of a trade-off. R2O2,R3M4

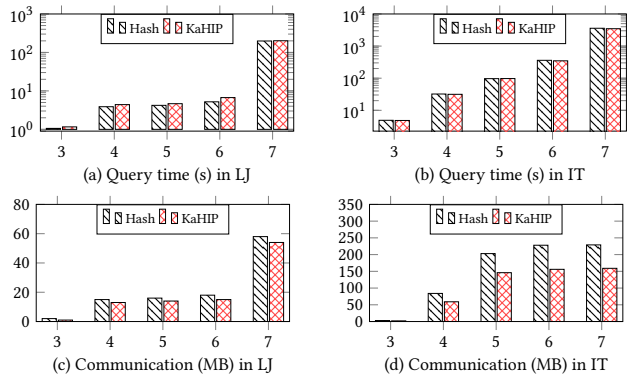


Figure 16: Performance evaluation with different partitioning methods

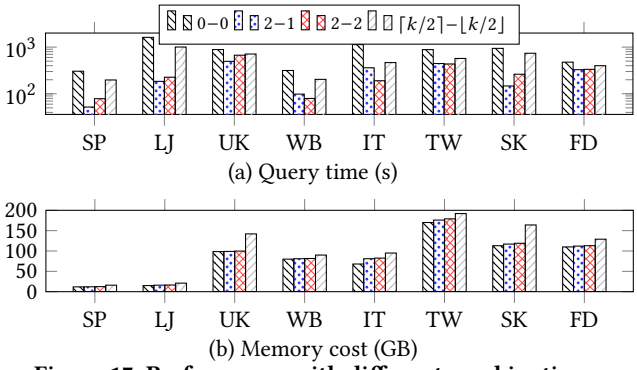


Figure 17: Performance with different combinations

6 CONCLUSION

In this paper, we address the hop-constrained path enumeration problem in the distributed environment. Specifically, we propose an efficient distributed approach **DistriEnum** which is composed of the modules of graph reduction, TCBSearch paradigm, task division, and vertex migration. Based on the above mechanism, **DistriEnum** can simultaneously satisfy excellent query performance, high parallelism, and good scalability with well-bound memory consumption. Our comprehensive experiments demonstrate that the proposed methods achieve great improvements in terms of query time, scalability, communication cost, and memory consumption. R3M7

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 349–360.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2016. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms* 12, 2 (2016), 14:1–14:22.

[3] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An Improved Algorithm for Incremental Cycle Detection and Topological Ordering in Sparse Graphs. In *SODA*. SIAM, 2509–2521.

[4] Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. 2013. Optimal Listing of Cycles and st-Paths in Undirected Graphs. In *SODA*. SIAM, 1884–1896.

[5] Katerina Böhmová, Luca Häfliger, Matús Mihalák, Tobias Pröger, Gustavo Sacomoto, and Marie-France Sagot. 2018. Computing and Listing st-Paths in Public Transportation Networks. *Theory Comput. Syst.* 62, 3 (2018), 600–621.

[6] Yuzheng Cai, Siyuan Liu, Weiguo Zheng, and Xuemin Lin. 2023. Towards Generating Hop-constrained s-t Simple Path Graphs. *CoRR* abs/2304.12656 (2023). https://doi.org/10.48550/arXiv.2304.12656

[7] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Jian Pei. 2015. Efficiently Computing Top-K Shortest Path Join. In *EDBT*. OpenProceedings.org, 133–144.

[8] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. Efficiently Answering Reachability and Path Queries on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 14, 10 (2021), 1845–1858.

[9] 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.

[10] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *Proc. VLDB Endow.* 9, 12 (2016), 1233–1244.

[11] Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Efficient Algorithms for Listing k Disjoint st-Paths in Graphs. In *LATIN (Lecture Notes in Computer Science, Vol. 10807)*. Springer, 544–557.

[12] Sairam Gurajada and Martin Theobald. 2016. Distributed Set Reachability. In *SIGMOD*. ACM, 1247–1261.

[13] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. 2012. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms* 8, 1 (2012), 3:1–3:33.

[14] Kongzhang Hao, Long Yuan, and Wenjie Zhang. 2021. Distributed Hop-Constrained s-t Simple Path Enumeration at Billion Scale. *Proc. VLDB Endow.* 15, 2 (2021), 169–182.

[15] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS, 2016*. ACM, 13–28.

[16] Larkshmi Krishnamurthy, Joseph H. Nadeau, Gultekin Özsoyoglu, Z. Meral Özsoyoglu, Greg Schaeffer, Murat Tasan, and Wanhong Xu. 2003. Pathways Database System: An Integrated System for Biological Pathways. *Bioinform.* 19, 8 (2003), 930–937.

[17] Rohit Kumar and Toon Calders. 2018. 2SCENT: An Efficient Algorithm to Enumerate All Simple Temporal Cycles. *Proc. VLDB Endow.* 11, 11 (2018), 1441–1453.

[18] Kisung Lee and Ling Liu. 2013. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.* 6, 14 (2013), 1894–1905.

[19] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *SIGMOD*. ACM, 1060–1077.

[20] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling Up Distance Labeling on Graphs with Core-Periphery Properties. In *SIGMOD*. ACM, 1367–1381.

[21] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. 2017. Compiling Graph Substructures into Sentential Decision Diagrams. In *AAAI*. 1213–1221.

[22] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. 2022. Answering reachability and K-reach queries on large graphs with label constraints. *VLDB J.* 31, 1 (2022), 101–127.

[23] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Hop-constrained s-t Simple Path Enumeration: Towards Bridging Theory and Practice. *Proc. VLDB Endow.* 13, 4 (2019), 463–476.

[24] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *CIKM*. ACM, 867–876.

[25] 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. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.

[26] Peter Sanders and Christian Schulz. 2013. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7933)*. Springer, 164–175.

[27] Baoxu Shi and Tim Weninger. 2016. Discriminative predicate path mining for fact checking in knowledge graphs. *Knowl. Based Syst.* 104 (2016), 123–133.

[28] Prashant Shiralkar, Alessandro Flammini, Filippo Menczer, and Giovanni Luca Ciampaglia. 2017. Finding Streams in Knowledge Graphs to Support Fact Checking. In *ICDM*. IEEE Computer Society, 859–864.

[29] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained s-t Path Enumeration. In *SIGMOD*. ACM, 1758–1770.

[30] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *SIGMOD*. ACM, 345–358.

[31] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *ICDE*. IEEE, 1153–1164.

[32] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (2014), 1981–1992.

[33] Norihito Yasuda, Teruji Sugaya, and Shin-ichi Minato. 2017. Fast Compilation of s-t Paths on a Graph for Counting and Enumeration. In *AMBN (Proceedings of Machine Learning Research, Vol. 73)*. PMLR, 129–140.

[34] Yuanyuan Zeng, Kenli Li, Xu Zhou, Wensheng Luo, and Yunjun Gao. 2022. An Efficient Index-Based Approach to Distributed Set Reachability on Small-World Graphs. *IEEE Trans. Parallel Distributed Syst.* 33, 10 (2022), 2358–2371.

[35] Yuanyuan Zeng, Wangdong Yang, Xu Zhou, Guoqin Xiao, Yunjun Gao, and Kenli Li. 2022. Distributed Set Label-Constrained Reachability Queries over Billion-Scale Graphs. In *ICDE*. IEEE.

[36] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *SIGMOD, 2023*. ACM, 61–68.

A ADDITIONAL EXPERIMENTS

Exp-11: Accuracy of workload estimator. In this part, we introduces experiments to illustrate the ratio between the actual measured load and our projected upper bound. As depicted in Fig. 18, for a majority of the queries, this ratio exceeds 0.6. In addition, we observe that the ratio slightly declines with the increase of k because the paths with repeated vertices are increased.

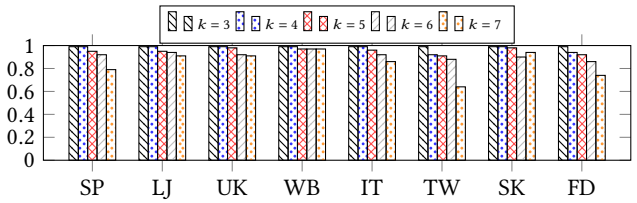


Figure 18: Ratio value on all datasets when varying k

Exp-12: Performance evaluation with shared indexes. We have evaluated IDX-DFS and IDX-JOIN with **shared indexes**, which perform better than those with distributed indexes. However, as shown in Fig. 19, DistriEnum still achieves speedups of 18.75× and 2.89× than IDX-DFS and IDX-JOIN on average in query performance.

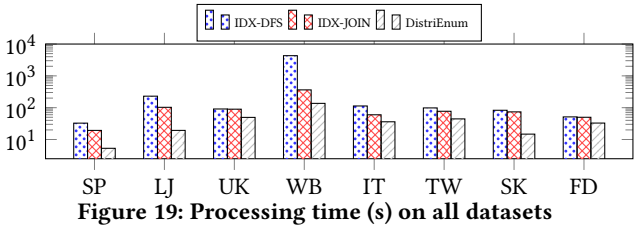


Figure 19: Processing time (s) on all datasets