Distributed Hop-Constrained s-t Simple Path Enumeration at Billion Scale

Kongzhang Hao
Nanjing University of Science and Technology
University of New South Wales
khao@cse.unsw.edu.au

Long Yuan*
Nanjing University of Science and Technology
longyuan@njust.edu.cn

Wenjie Zhang
University of New South Wales
zhangw@cse.unsw.edu.au

ABSTRACT

Hop-constrained *s-t* simple path (HC-s-t path) enumeration is a fundamental problem in graph analysis and has received considerable attention recently. Straightforward distributed solutions are inefficient and suffer from poor scalabiltiy when addressing this problem in billion-scale graphs due to the disability of pruning fruitless exploration or huge memory consumption. Motivated by this, in this paper, we aim to devise an efficient and scalable distributed algorithm to enumerate the HC-s-t paths in billion-scale graphs. We first propose a new hybrid search paradigm tailored for HC-s-t path enumeration. Based on the new search paradigm, we devise a distributed enumeration algorithm following the divide-and-conquer strategy. The algorithm can not only prune fruitless exploration, but also well bound the memory consumption with high parallelism. We also devise an effective workload balance mechanism that is automatically triggered by the idle machines to handle skewed workloads. Moreover, we explore the bidirectional search strategy to further improve enumeration efficiency. The experiment results demonstrate the efficiency of our proposed algorithm.

PVLDB Reference Format:
Kongzhang Hao, Long Yuan, Wenjie Zhang. Distributed Hop-Constrained s-t Simple Path Enumeration at Billion Scale. PVLDB, 15(2): 169 - 182, 2022.
doi:10.14778/3489496.3489499

1 INTRODUCTION

Graphs have been widely used to represent the relationships of entities in many areas [10, 34, 35, 39, 42, 44, 51–53, 55–57]. Recently, hop-constrained s-t simple path enumeration has received considerable attention [18, 43, 45, 47]. Given an unweighted directed graph G, a source vertex s, a target vertex t, and a hop constraint k, hop-constrained s-t simple path (HC-s-t path for short) enumeration computes all the simple paths (i.e., paths without repeated vertices) from s to t such that the number of hops in each path is not larger than k.

Applications. HC-s-t path enumeration can be used in many applications, for example:

• *Fraud detection in E-commerce transaction networks.* A cycle in a E-commerce transaction network is a strong indication of a fraudulent activity [58]. A recent paper by Alibaba group demonstrates that when a new transaction is submitted from account t to account s in its E-commerce network, HC-s-t path enumeration is used to report all newly formed cycles to detect fraudulent activities [45].
• *Pathway queries in biological networks.* Pathway queries are a fundamental tool in biological networks analytics [26, 32]. [32] shows that HC-s-t path enumeration is one of the most important pathway queries that can identify the chains of interactions.
• *Path ranking in knowledge graphs.* Path ranking algorithms enumerate the paths from one entity to another in a knowledge graph and use these paths as features to train a model for missing fact prediction [16, 31, 36]. HC-s-t path enumeration can be used in this case as shown in [16].

Motivation. In real-world applications, the scale of graphs is large and grows exponentially. For example, the sub-domain of a web graph from EU countries contains 1.07 billion vertices and 91.79 billion edges [12]. Obviously, processing such big graphs requires huge memory and computation resources that are expensive to obtain from a single machine, which makes centralized algorithms designed for a single machine [18, 43, 45, 47] not scalable for real-world applications. In contrast, distributed computing clusters provide sufficient resources in a relatively easy and cheap way, and efficient and scalable distributed solutions for many other graph problems have been proposed [19, 23, 25, 41, 59]. Therefore, we study the distributed HC-s-t path enumeration problem.

In the literature, there exists no specialized distributed solution for HC-s-t path enumeration. Nevertheless, as BFS-oriented exploration is relatively easy to parallelize and implement in a distributed setting, we can obtain a straightforward distributed solution for HC-s-t path enumeration based on the BFS-oriented exploration as follows: we start the search from the source vertex s following the BFS-oriented paradigm and conduct the enumeration in k synchronous rounds. In each round, the paths are extended by a vertex locally and the extended paths are sent to the machine where the new vertex resides. In detail, for a path $p = (s, \ldots, v)$, let v' be one of v's out-neighbors. If v' is not contained in p, then it extends p by v' to p' locally, if v' is the target vertex t, p' is output; otherwise, p' is sent to the machine where v' resides. It can be easily verified that all the HC-s-t paths are enumerated in k rounds.

Although this approach can fully utilize the computational resource due to its high parallelism, it is inefficient and suffers from poor scalability when enumerating HC-s-t paths in billion-scale graphs. This is because: (1) as proven in the design of centralized algorithms [18, 43, 47], pruning unnecessary exploration has crucial

* Long Yuan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489499

effects on the enumeration performance, but this approach does not consider the possible pruning opportunities to reduce unnecessary computation; (2) during the enumeration, massive intermediate paths are generated and maintained due to the exponentially growing search space in this approach, which leads to huge memory consumption and expensive communication costs.

On the other hand, since an HC-s-t path query can be considered as a special kind of subgraph matching query (path pattern with length not larger than k, refer to Section 3.2), the distributed algorithms designed for subgraph matching can also be adapted to address the HC-s-t path problem [2, 14, 27, 29, 46, 54]. However, the results from our experiments show that the performance of this approach is also poor in billion-scale graphs due to the lack of the ability to prune unnecessary computation.

Motivated by this, we aim to develop an efficient and scalable distributed algorithm tailored for HC-s-t path enumeration in billion-scale graphs. The algorithm should not only have the ability to prune unnecessary computation during the enumeration, but also fully utilize the computation resource with bounded memory consumption.

Challenges. It is challenging to develop an algorithm which can achieve the aforementioned goals at the same time. Regarding the pruning ability, the existing pruning techniques are proposed in centralized algorithms and these techniques heavily depend on the DFS-oriented paradigm. However, it is believed that DFS is inherently sequential and difficult to parallelize [22], which implies that if the pruning techniques are integrated into our distributed algorithm, it is quite possible that the computational resource of the distributed systems cannot be fully exploited. Additionally, in order to achieve high parallelism in our distributed algorithm, it seems inevitable to divide the enumeration into multiple rounds and maintain the intermediate paths, which means the memory consumption is hard to be well-bounded.

Our idea. Our general idea to overcome these challenges is simple: instead of directly using the BFS-oriented paradigm or DFS-oriented paradigm, we propose a new hybrid search paradigm which imitates the DFS procedure from an overall perspective while adopting an extending method similar to BFS at each step. Based on this new hybrid search paradigm, we are able to not only prune unnecessary computation without sacrificing parallelism, but also control the size of the generated immediate paths. However, to make our idea practically applicable, the following issues still need to be addressed: (1) how to correctly integrate the pruning techniques into a hybrid search paradigm? (2) how to develop the new search algorithm in a distributed setting? (3) how to handle the skewed workload which commonly occurs in distributed systems?

Contributions. In this paper, we address these issues and make the following contributions:

(A) The first work to study the distributed HC-s-t path *enumeration problem.* In this paper, we aim to address the problem of HC-s-t path enumeration in a distributed setting. To the best of our knowledge, this is the first work to study the distributed algorithm for HC-s-t path enumeration at billion scale.

(B) Efficient and scalable distributed algorithms for HC-s-t path *enumeration.* We propose a new hybrid search paradigm for the problem of HC-s-t path enumeration. Based on it, we devise a new distributed algorithm following the divide-and-conquer strategy. Besides fully exploiting the computational resource with bounded memory consumption, our new algorithm can also prune a huge amount of unnecessary computation during the enumeration, which significantly reduces the search space and improves enumeration efficiency. In addition, we design an effective work stealing mechanism to handle unbalanced workloads. Moreover, we develop a bidirectional search method based on our algorithm in distributed setting, which further accelerates the HC-s-t path enumeration.

(C) Extensive performance studies on real-world and synthetic datasets. We conduct extensive performance studies using large real-world graphs and synthetic graphs. The experiment results demonstrate that our proposed algorithms are efficient and scalable to enumerate HC-s-t paths in billion-scale graphs.

Outline. Section 2 provides the problem definition. Section 3 introduces related work. Section 4 shows the framework of our approach. Section 5 presents the details of the implementation. Section 6 evaluates the proposed algorithms and Section 7 concludes the paper. Full proofs and experimental results can be found in our technical report [20].

2 PRELIMINARIES

Let $G = (V, E)$ denote a unweighted directed graph, where $V(G)$ is the set of vertices and $E(G)$ is a set of directed edges. For a vertex $v \in V(G)$, we use $\mathcal{N}(v)$ to denote the neighbors of v and $\mathsf{nbr}^-(v)/\mathsf{nbr}^+(v)$ to denote the in-neighbors/out-neighbors of v. The in-degree/out-degree of v, denoted by $\deg^-(v)/\deg^+(v)$, is the number of in-neighbors/out-neighbors of v in G, i.e., $\deg^+(v) = |\mathsf{nbr}^-(v)|/\deg^+(v) = |\mathsf{nbr}^+(v)|$. Given a graph G, the reverse graph of G, denoted by $G^r = (V, E^r)$, is the graph generated by reversing the direction of all edges in G. A path from vertex u to vertex v, denoted by $p(u, v)$, is a sequence of vertices $\{u = v_0, v_1, ..., v_h = v\}$ such that $(v_{i-1}, v_i) \in E(G)$ for every $1 \leq i < h$. A simple path is a loop-free path where there are no repetitions of vertices and edges. By $|p|$, we denote the length (i.e., the number of hops in this paper) of path p. Given two vertices u and v, the shortest distance from u to v, denoted by $\mathsf{dist}(u, v)$, is the length of the shortest path from u to v. Given a pre-defined hop constraint k, we say a path p is a hop-constrained path if $|p| \leq k$. For presentation simplicity, we refer to the hop-constrained s-t simple path as the HC-s-t path.

Problem statement. Given an unweighted directed graph G, a source vertex s, a target vertex t and a hop constraint k, HC-s-t path enumeration computes the HC-s-t paths from s to t in G.

Graph storage. Given a data graph G, we use the widely used *hash partitioning* by default [1, 40] and assume the data graph is partitioned by vertices and stored in the cluster, that is, for each vertex $v \in V$, we store it with its adjacency list $(v; \mathcal{N}(v))$ in one of the partitions. We call a vertex that resides in the local partition a *local vertex*, or a *remote vertex* otherwise. Although we use the *hash partitioning* in this paper, our approach is orthogonal to partitioning, i.e., it flexibly supports all graph partitioning methods, including other edge-cut partitioning methods [3, 24], vertex-cut partitioning methods [8, 17], and hybrid partitioning methods [11, 33, 60]. Of

course, intelligent partitioning schemes could bring further performance benefits. We leave the design of a better graph partitioning strategy for HC-s-t path enumeration for future work.

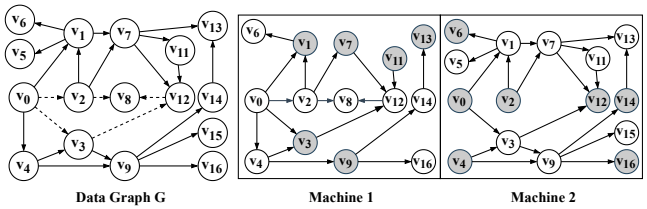


Figure 1: Example Data Graph

Example 2.1: Consider graph G shown in Figure 1, which is partitioned into two machines. In each partition, the *local vertices* and *remote vertices* are represented by white and grey respectively. Each *local vertex* has its adjacency list stored in the same machine. Given an HC-s-t path enumeration query with $s = v_0$, $t = v_8$ and $k = 4$, two HC-s-t paths can be found, namely $\{v_0, v_2, v_8\}$ and $\{v_0, v_3, v_{12}, v_8\}$, which are shown by dashed arrows in G.

3 RELATED WORK

In this section, we extend the description of the related work on HC-s-t path enumeration in Section 1. We roughly divide the related works into two categories: centralized related solutions and distributed related solutions.

3.1 Centralized Related Solutions

DFS-oriented exploration based approaches. HC-s-t path enumeration is a fundamental problem in graph analysis and several centralized DFS-oriented exploration based algorithms for this problem have been proposed [18, 43, 47]. Generally, starting from the source vertex s, these algorithms explore the vertices following a DFS-oriented paradigm with depth of at most k. During the exploration, an HC-s-t path is found if the target vertex t is encountered. To prune the fruitless exploration, different strategies are used in these algorithms.

Regarding T-DFS [47], for each out-going neighbor v of a visited vertex during the exploration, it computes the shortest path distance from v to t without containing any vertex on its current path, which can be achieved by one BFS starting from t on the reversed graph. By aggressively checking whether each search branch is promising, T-DFS guarantees that there is at least one HC-s-t path for each search branch explored. T-DFS2 [18] follows the same aggressive verification strategy as T-DFS, while it can reduce the shortest path distance computation by skipping some vertices associated with only one output in the following search. However, T-DFS and T-DFS2 show poor performance in practice due to the expensive verification cost [43]. To address the inefficiency of T-DFS and T-DFS2, [43] proposes the state-of-the-art algorithm JOIN for HC-s-t path enumeration in this category. JOIN dynamically maintains a lower bound of hops to the target vertex t for the vertices visited during the DFS-oriented exploration. When a new vertex v of a path is explored, if the sum of hops from s to v (already explored) and the lower bound of hops from v to t (to be explored) is larger than k, then out-neighbors of v will not be explored and the fruitless searches following v are pruned. As analyzed in [43], this pruning technique significantly reduces the search space and plays the key role in improving enumeration efficiency.

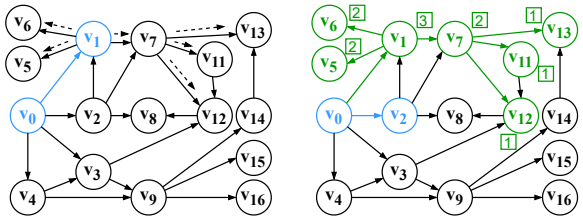


Figure 2: Key Idea of JOIN

Example 3.1: Consider graph G shown in Figure 1 and an HC-s-t path enumeration query with $s = v_0$, $t = v_8$ and $k = 4$. When T-DFS and T-DFS2 start their search with stack $S = \{v_0\}$, instead of directly extending v_1, they will first compute the shortest distance from v_1 to t, then check if the minimum number of hops required to reach t from v_1 is within the budget. As $\text{dist}(v_1, v_8) + |S| = 5 > k$, the exploration following v_1 is unpromising and is thus avoided by T-DFS and T-DFS2.

For JOIN, Figure 2 shows its key idea. Assume that the current search stack in JOIN is $S = \{v_0, v_1\}$. The vertices in S are marked in black in Figure 2 (a). The lower bound of the number of hops from a vertex to t is referred as its bar. After finishing the DFS exploration following v_1 (dashed arrows in Figure 2 (a)), the explored vertices are marked in green in Figure 2 (b) and no valid path is found. Then JOIN sets v_1.bar = k + 1 - $|S|$ = 3, which means v_1 requires at least 3 hops to reach t. Similarly, v_7.bar is set to 2. The value of bar for each vertex is shown next to it in Figure 2 (b). When v_1 is unstacked and v_2 is pushed into S, it will not explore v_1 and v_7 again, because it will check whether the minimum numbers of hops required to reach t from v_1 and v_7 are within the budget. In this example, $|S|$ + 1 + v_1.bar = 6 > 4 and $|S|$ + 1 + v_7.bar = 5 > 4, which means both v_1 and v_7 require more hops to reach t than the available budget. Hence v_1 and v_7 are pruned and the fruitless exploration is avoided.

Although these algorithms perform well in the centralized setting, it is hard to extend them to a distributed context. This is because the whole logic of the pruning techniques is based on the DFS-oriented paradigm. However, it is believed that DFS is inherently sequential and challenging to parallelize [22]. Hence, directly extending these centralized solutions to a distributed context seems unpromising.

BFS-oriented exploration based approaches. Apart from the DFS-oriented exploration based approaches, the HC-s-t paths can be enumerated following a BFS-oriented exploration as discussed in Section 1. In the literature, extensive works have been conducted on the multi-threading of BFS aiming to improve the performance of BFS by parallelization, such as [5, 21, 50]. By using a simple heuristic to dynamically pick the *top-down* or *bottom-up* BFS strategies at runtime, [5] can reduce the number of edges examined, which in turn accelerates the BFS exploration as a whole. [21] introduces a new implementation method of the parallel BFS algorithm on multi-core CPUs, which improves the performance by utilizing memory bandwidth more efficiently. [50] studies the problem of Multi-Source BFS, which increases the overall performance by sharing common computation across concurrent BFSs and reducing

the number of random memory accesses. Obviously, these techniques can also be extended to a distributed setting for HC-s-t paths enumeration as discussed in Section 1.

As discussed in Section 1, although the above BFS-oriented exploration based distributed approaches can address the problem, all of them have the following drawbacks: (1) the possible pruning opportunities to reduce unnecessary computation specific to HC-s-t path enumeration are not considered, which limits their enumeration performance. (2) the massive intermediate paths generated during the enumeration easily lead to out-of-memory problem when the hop constraint k is large (refer to Exp-2 in Section 6). Besides, in each round, these intermediate paths have to be shuffled through the network, which leads to expensive network communication cost. The above drawbacks together make these approaches inefficient to enumerate HC-s-t paths in billion-scale graphs.

Index-based approach. In the literature, a centralized index-based approach for HC-s-t path enumeration is also mentioned in [45], which proposes an index named HP-Index to continuously maintain the pairwise paths among hot points (i.e., vertices with high degree) in a graph. With the HP-Index, HC-s-t paths can be enumerated by utilizing a bidirectional search which does not explore the hot points encountered. The HC-s-t paths among the hot points are then computed based on the HP-Index. Following the idea of HPI, a distributed counterpart can be obtained easily as follows: for the HP-Index, we distribute the paths across the machines where the hot points reside. With the distributed HP-Index, we enumerate the HC-s-t paths in the following steps: (1) run a BFS-oriented exploration from s on G with search depth of at most k. During the search, the paths that encounter the hot points are not further extended but cached in their current machines. (2) run a BFS-oriented exploration from t on G^r similar to (1). (3) Find the HC-s-t paths among the hot points encountered in (1) and (2). (4) Concatenate the paths from (1), (2) and (3) to identify the remaining HC-s-t paths. In the experiment, we also evaluate the performance of this approach (called DisHPI). However, this approach performs poorly in our experiments and is not a good solution for distributed HC-s-t path enumeration.

3.2 Distributed Related Solutions

In the literature, there exists no specialized distributed solution for HC-s-t path enumeration. However, as the results of HC-s-t path enumeration are a series of directed edges from s to t with length not larger than k, HC-s-t path can be treated as a special kind of subgraph matching query (path pattern with length not larger than k) and the problem of HC-s-t path enumeration can be addressed by utilizing the existing subgraph matching systems.

Many distributed systems for subgraph matching have been proposed [2, 13, 14, 28, 49]. Of these, BiGJoin [2] and Fractal [14] are two representative ones. BiGJoin follows the worst-case optimal join algorithm [38], which extends the (intermediate) results one vertex at a time by intersecting the neighbors of all its connected vertices. As shown in [29], BiGJoin can process path pattern efficiently. Fractal is a distributed system for general graph pattern mining that includes subgraph matching. It enumerates subgraphs by combining a DFS strategy with a from-scratch processing paradigm to improve memory efficiency, and employs a dynamic load-balancing based on a work stealing mechanism that allows the system to handle different workload characteristics. Although these systems can be adapted to address the HC-s-t path enumeration problem as they support path pattern query, these systems focus on general subgraph matching and the optimizations specific to HC-s-t path enumeration are hard to be integrated into these systems. Therefore, these systems cannot enumerate HC-s-t paths in billion-scale graphs efficiently. As evaluated in our experiments, our proposed algorithm can achieve a speedup of up to two orders of magnitude compared to these approaches.

4 OUR FRAMEWORK OVERVIEW

4.1 A Hybrid Search-Oriented Framework

Based on the analysis in Section 3, we have to make a fresh start and design a new distributed algorithm tailored for HC-s-t path enumeration. The design goals of our new approach are: (1) *Pruning power.* Our approach should support effective pruning techniques similar to the centralized solution to reduce unnecessary computation, and improve enumeration efficiency consequently. (2) *Parallelism.* Our approach should fully utilize the computation resources of the distributed system. (3) *Memory consumption.* Our approach should be well controlled in terms of the memory consumption of each machine in the system. (4) *Load balance.* As a distributed solution, our approach should be able to handle situations with skewed workloads.

In this section, we present a hybrid search-oriented framework to achieve the goals on pruning power, parallelism, and memory consumption. We discuss the load balance in the next section.

A hybrid search paradigm. As discussed in the previous section, to obtain a similar pruning power to the centralized solution, DFS-oriented exploration has to be retained. This is because the pruning power of the centralized solution comes from the explored paths with k-hops or ending with t. Meanwhile, the good parallelism of BFS-oriented exploration is also consistent with one of our design goals. Revisiting these design goals, it can be observed that DFS-oriented and BFS-oriented explorations can be treated as two extreme cases regarding our design goals. This inspires us to devise a hybrid search paradigm aiming to combine the advantages of DFS-oriented exploration and BFS-oriented exploration. Intuitively, the hybrid search paradigm imitates the search procedure of DFS while in each step, instead of just one neighbor being extended, we extend at most Δ out-neighbors similar to BFS, where Δ is chosen by the users. To make the idea practical, we define:

Definition 4.1: (Intermediate Path Tree) The intermediate path tree Ψ is a tree where each node n represents a vertex in input graph G and stores a tuple (vid, state, pt, parent, children) that captures the data vertex v (vid), its state (state, a node has one of the five states: OPEN, CLOSED, POTENTIAL, POSTPONE, NULL), pruning thresholds for a set of data vertices (pt), a pointer to its parent in Ψ (parent), and the number of its children in Ψ (children). The five states are used to provide precise control over the flow of the search and ensure that the updates on the nodes' pruning thresholds are correctly propagated. Intuitively, nodes with state OPEN can be extended in the current round, nodes with state CLOSED are no longer extended, nodes with state POTENTIAL can be potentially extended in the next round and nodes with state POSTPONE are

delayed from being extended in the current round to being extended in the future.

For ease of presentation, we refer to each $v \in V(G)$ in G as a vertex and refer to each $n \in V(\Psi)$ in Ψ as a node. Furthermore, we call a path in Ψ as a tree path while a path in G as a graph path, and a graph path represented by a tree path means the graph path consisting of the vertices represented by the nodes in the tree path. Given a node n, we use level(n) to represent the level of n in Ψ, and the level of the root node is 0. Based on input graph G and intermediate path tree Ψ, we further define three operators:

Algorithm 1: Operator PrunExt

```
1  parallel foreach n ∈ OPEN nodes in Ψ do
2      if level(n) = k - 1 then
3          n.pt.(n.vid) ← 2; n.state ← CLOSED; continue;
4      foreach v' ∈ nbr+(n.vid) and v' ∉ root-to-n path do
5          if v' = t then
6              n.addChild(Node(t, CLOSED, {t : 1}, n, 0));
7              Output graph path;
8          else
9              n'' ← nearest node on root-to-n path where
                v' ∈ n''.pt;
10             if n''.pt.v' + level (n) + 1 ≤ k then
11                 n.addChild(Node(v', POTENTIAL, ∅, n, 0));
12         if no new nodes added then
13             n.state ← CLOSED;
                n.pt.(n.vid) ← k − level(n) + 1;
14         else n.state ← NULL;
15     if Δ nodes added in total then break;
```

Definition 4.2: (Operator PrunExt**)** Operator PrunExt explores the out-going neighbors of the vertices represented by the nodes with state OPEN and adds new nodes in Ψ. As shown in Algorithm 1, for each node n with state OPEN, if level(n) = $k-1$, n's state is set as CLOSED with n.pt.(n.vid) set as 2 (lines 1-3). Otherwise, PrunExt explores $v' \in$ nbr$^+$(n.vid) that is not contained in the graph path represented by the root-to-n tree path in Ψ (line 4). If v' is the target vertex t, a new node n' with tuple (t, CLOSED, $\{t:1\}$, n, 0) is added as a child of n and the root-to-n' path is output (lines 5-7). Otherwise, if n''.pt.v' + level(n) + 1 $\leq k$, where n'' is the nearest node to n on the root-to-n tree path that contains a pruning threshold regarding v' (if there is no such pruning threshold, n''.pt.$v' = 0$), a new node n' with tuple (v', POTENTIAL, $\emptyset$, n, 0) is added as a child of n (lines 9-11). If no new node has been added as a child of n, n's state is set as CLOSED and n.pt.(n.vid) is set as k − level(n) + 1; Otherwise, n's state is set as NULL (lines 12-14). PrunExt adds at most Δ new nodes each time (line 15).

Definition 4.3: (Operator BackProp**)** Operator BackProp mainly focuses on updating the pt field of nodes in Ψ. As shown in Algorithm 2, for each leaf node n with state CLOSED, let n' be its parent in Ψ (lines 1-2). BackProp (1) updates the pt field of n and n' as follows: for n, if n.pt.(n.vid) + level(n) $< k$, for each $v \in n$.pt, if n.pt.v > n.pt.(n.vid) + dist(v, n.vid), then BackProp sets n.pt.v as n.pt.(n.vid) + dist(v, n.vid) (lines 3-6); for n', it sets n'.pt.(n'.vid) to min$\{n'$.pt.(n'.vid), n.pt.(n.vid) + 1$\}$ (line 7). (2) merges n.pt into

Algorithm 2: Operator BackProp

```
1  parallel foreach n ∈ CLOSED nodes in Ψ do
2      n' ← n.parent;
3      if n.pt.(n.vid) + level(n) < k then
4          foreach v ∈ n.pt do
5              if n.pt.v > n.pt.(n.vid) + dist(v, n.vid) then
6                  n.pt.v ← n.pt.(n.vid) + dist(v, n.vid);
7      n'.pt.(n'.vid) ← min{n'.pt.(n'.vid), pt_v + 1};
8      for v ∈ n.pt do
9          if v ∈ n'.pt then n'.pt.v ← max{n'.pt.v, n.pt.v};
10         else n'.pt.v ← n.pt.v;
11     Remove(n); if n'.children = 0 then n'.state = CLOSED;
```

n'.pt as follows: for each vertex $v \in \{n.\text{pt} \bigcap n'.\text{pt}\}$, n'.pt.v is set as max$\{n'$.pt.v, n.pt.$v\}$ (line 9); for each vertex $v \in \{n.\text{pt} \backslash n'.\text{pt}\}$, a new entry ($v$, n.pt.v) is added in n'.pt (line 10). (3) removes n from n' (line 11). If n' becomes a leaf node due to the removal of n, BackProp continues to perform the same steps on n' by setting n'.state as CLOSED (line 11).

Algorithm 3: Operator FlowCtrl

```
1  l_max ← maximum level of Ψ;
2  parallel foreach n ∈ leaf nodes at level l_max of Ψ do
3      n.state ← OPEN;
4  parallel foreach n ∈ leaf nodes at level l_max-1 of Ψ do
5      n.state ← POSTPONE;
```

Definition 4.4: (Operator FlowCtrl**)** Operator FlowCtrl controls the direction and scope of the exploration by adjusting the state of the nodes in Ψ. As shown in Algorithm 3, let $l_{\max}$ be the maximum level of Ψ (line 1). FlowCtrl sets the state of all the leaf nodes at level $l_{\max}$ as OPEN (lines 2-3), and if leaf nodes exist at level $l_{\max} - 1$, sets the state of these leaf nodes as POSTPONE (lines 4-5).

By iteratively performing the three operators on the immediate path tree, we can achieve our desired hybrid search on G.

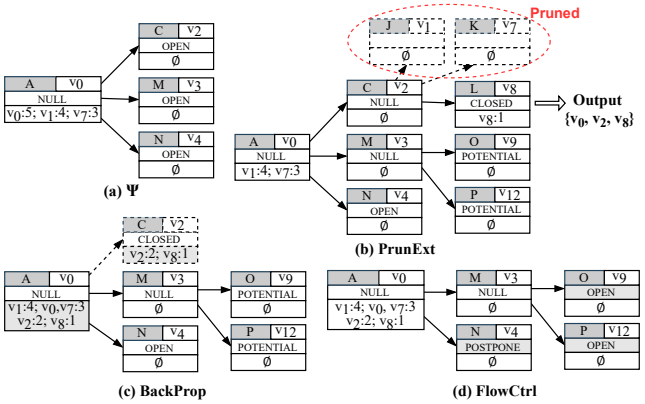


Figure 3: Running Example of Hybrid Search

Example 4.1: Consider G shown in Figure 1 and assume the HC-s-t path enumeration query is with $s = v_0$, $t = v_8$ and $k = 4$. Figure 3

demonstrates the execution of the three operators during the search, in which every node in Ψ is shown with its alias, its vid, its state and its pt. Δ is set as 3 for Ψ. For clearness of presentation, we assume that the graph is on a single machine.

Given the initial Ψ as shown in Figure 3 (a), PrunExt conducts the exploration on nodes with state OPEN, namely, C, M and N, which is shown in Figure 3 (b). Firstly, three out-neighbors of C.vid $= v_2$ are found, namely v_1, v_7 and v_8. For v_1, as A is the nearest node on the root-to-C path that contains the pruning threshold of v_1, and A.pt.v_1 + level(C) + 1 = 6 > k = 4, it is apparent that further exploration on v_1 will be fruitless. Hence v_1 is pruned, which is indicated by the dashed node J. v_7 can be pruned similarly, as demonstrated in the red dashed ellipse. For v_8, as v_8 is the target vertex, a node L with L.state = CLOSED and L.pt.v_8 = 1 is added and the found HC-s-t path $\{v_0, v_2, v_8\}$ is output. Due to the added node L, C.state is set as NULL. Similarly, the two out-neighbors of M.vid $= v_3$ are further added as child nodes (O, P) to let the number of extended children reach Δ.

After this, BackProp continues the search, which is shown in Figure 3 (c). Since the state of L is CLOSED, BackProp first removes L from Ψ. After L is removed, C.pt.(C.vid) is set as L.pt.(L.vid)+1 = 2 and L.pt is merged into C.pt, which is illustrated with the dashed node C. Because C subsequently becomes a leaf node, BackProp applies on C again. After C is removed, A.pt.(A.vid) is updated to min{A.pt.(A.vid), C.pt.(C.vid) + 1} = 3 and C.pt is merged into A.pt. Figure 3 (d) shows the following procedure of FlowCtrl. It changes the states of O and P from POTENTIAL to OPEN. The state of N is also changed to POSTPONE.

A hybrid search-oriented framework. With the hybrid search paradigm, the following problem is to design the enumeration algorithm in the distributed system based on the hybrid search paradigm. To address this problem, we adopt a divide-and-conquer strategy, which is based on the following lemma:

Lemma 4.1: *Given an* HC-s-t path *query with hop constraint* k *on* G, $P_k(s,t) = \bigcup_{v \in \mathsf{nbr}^+(s)} P_{k-1}(v,t)$.

According to Lemma 4.1, the HC-s-t paths from s to t with k hop constraint can be obtained by computing the HC-s-t paths from v to t with $k-1$ hop, where $v \in \mathsf{nbr}^+(s)$. Therefore, we can divide the enumeration from s to t with k hop constraint into a series of enumerations from v to t with $k-1$ hop constraint and compute the HC-s-t paths from v to t following the hybrid search paradigm on each machine respectively.

Algorithm 4: HybridEnum (s, t, k, Δ, G)

```
1 distribute the out-neighbors of s evenly to each machine;
2 foreach machine do
3 |   create a node n with tuple (s, NULL, ∅, n, λ), where λ is
  |     the number of v ∈ nbr+(s) assigned to the machine ;
4 |   foreach v ∈ nbr+(s) assigned to the machine do
5 |   |   create a node n' with tuple (v, POTENTIAL, ∅, n, 0)
  |   |     as a child of n;
6 foreach machine do
7 |   while ∃ n ∈ Ψ do
8 |   |   PrunExt(); BackProp(); FlowCtrl();
```

Algorithm. Following the above idea, our hybrid search-oriented algorithm, HybridEnum, is shown in Algorithm 4. Algorithm 4 first distributes the workload from the out-neighbors of s evenly on all the available machines (line 1). After splitting the workload, each machine starts its own exploration asynchronously. For each machine, it first creates the immediate path tree Ψ such that root node n represents vertex s with tuple $(s, \text{NULL}, \emptyset, n, \lambda)$, where λ is the number of $v \in \mathsf{nbr}^+(s)$ assigned to the machine and creates a node n' for each $v \in \mathsf{nbr}^+(s)$ assigned to the machine with tuple $(v, \text{POTENTIAL}, \emptyset, n, 0)$ as the child of n (lines 3-5). After this, each machine iteratively executes the three operators on its own Ψ and the procedure terminates when no node is left (lines 7-8).

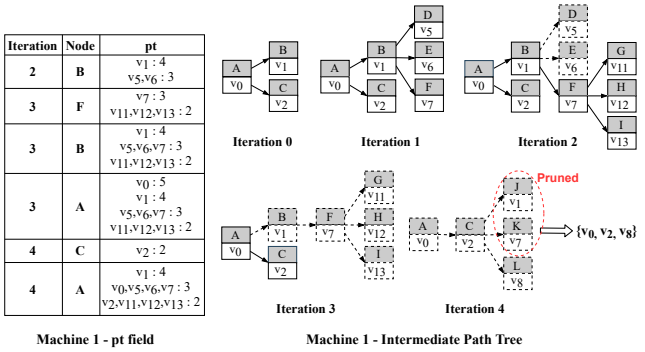


Figure 4: Steps of HybridEnum

Example 4.2: Reconsider G shown in Figure 1, and still assume the HC-s-t path enumeration query is with $s = v_0$, $t = v_8$ and $k = 4$. We set $\Delta = 3$. The workload is distributed equally based on the out-going neighbors of s. Assume that machine 1 and machine 2 are allocated with $\{(v_0, v_1), (v_0, v_2)\}$ and $\{(v_0, v_3), (v_0, v_4)\}$, respectively. Figure 4 only shows the detailed execution of HybridEnum on machine 1 as the execution on machine 2 is similar.

For clearness of presentation, in Figure 4, for each node, we only show its alias and vid. The state field is ignored as it can be derived directly. We also show the nodes with the value changed in their pt field between iterations together in the left. The dashed nodes are those that are removed in the corresponding iteration.

The whole search on machine 1 can be finished in 5 iterations. In *iteration 0*, the initial Ψ is constructed based on the allocated edges $\{(v_0, v_1), (v_0, v_2)\}$. In *iteration 1*, after extending nbr^+ (B.vid) from B, C is left unextended because the number of extended nodes has already reached Δ. In *iteration 2*, as $\mathsf{nbr}^+(D.\text{vid})$ and $\mathsf{nbr}^+(E.\text{vid})$ are both empty, D, E cannot be extended and F is extended with G, H and I. Additionally, as D and E can no longer be extended, they are removed and $\{v_5, v_6 : k - \text{level}(D) + 1 = 3; v_1 : 3 + 1 = 4\}$ is inserted into B.pt by BackProp. In *iteration 3*, as G, H and I have all reached the hop constraint of 4, they cannot be extended any more. Thus they are removed and F.pt is updated to $\{v_{11}, v_{12}, v_{13} : 2; v_7 : 3\}$. It is noticed that F becomes a new leaf node and hence BackProp is applied continuously on F. Therefore, B.pt.(B.vid) is set as min{B.pt.(B.vid), F.pt.(F.vid) + 1} = 4 and F.pt is merged into B.pt. As B subsequently becomes a leaf node, BackProp is applied on B again, which removes B. As a result, A.pt.(A.vid) is set as B.pt.(B.vid) + 1 = 5 and B.pt is merged into A.pt. In *iteration*

4, PrunExt is applied on C first. Three out-going neighbors of C.vid are found in G, namely v_1, v_7 and v_8 (i.e. t). For v_1 and v_7, it is found in the pruning check that $A.\text{pt}.v_1 + \text{level(C)} + 1 = 6 > k$ and $A.\text{pt}.v_7 + \text{level(C)} + 1 = 5 > k$, which indicates that the searches following v_1 and v_7 are fruitless. Hence v_1 and v_7 are pruned, shown in the red dashed ellipse. For v_8, as v_8 is the target vertex, the corresponding HC-s-t path $\{v_0, v_2, v_8\}$ is output. The search finishes when Ψ has no nodes.

Theorem 4.1: *The memory usage of* HybridEnum *on G in each machine is bounded by $O(k|V(G)| + k^2\Delta + k\Delta)$.*

Theorem 4.2: *The communication cost and computation cost of* HybridEnum *on G in all machines are bounded by $O(k|E(G)|\phi)$ and $O(k|V(G)|\phi\Delta)$ respectively, where ϕ is the output path number.*

Tuning Δ. Based on the above analysis, the value of Δ can affect the algorithm's overall performance. Intuitively, a big Δ is preferred from the perspective of parallelism, but it would lead to the out of memory issue if the value of Δ is too big. Therefore, we can estimate a proper Δ based on the available memory. Specifically, let M be the size of the available memory for each machine. The memory consumption of each machine consists of three parts: (1) the Intermediate Path Tree Ψ, whose size can be represented as $\alpha k\Delta$; (2) the memory used for the partitioned input graph, whose size can be represented as $\beta|G|$; (3) the constant memory overhead such as the cache, whose size can be represented as γ. where α, β and γ are coefficients related to the system configuration. To avoid the problem of out of memory, we have $M \geq \alpha k\Delta + \beta|G| + \gamma$. Therefore, we have $\Delta \leq \frac{M-\beta|G|-\gamma}{\alpha k}$. For an HC-s-t path enumeration query on a given graph running on a specific distributed cluster, the values of M, $|G|$, k, α, β and γ can be known in advance or easily estimated. Thus, we can obtain a proper value of Δ following the above formula.

4.2 Load balance

In the real world, there exist some skewed graphs that make the split workload for the machines imbalanced. We address the straggler problem by devising a dynamic work stealing based mechanism. Its main idea is that the idle workers automatically "steal" the unfinished workload from the busy workers to accelerate the whole process. To apply dynamic work stealing, the main challenge is to migrate the workload without affecting the correctness of pt for each node $n \in \Psi$ on both machines. According to the procedure of BackProp, the key point to maintain the correctness of each n.pt is to guarantee that the out-neighbors of n.vid have been fully explored when operator BackProp is applied on n. Based on this, a straightforward solution is to let the machine that is in charge of performing BackProp wait until the stealing machines finish their workloads and the results are received. However, this approach has the potential to cause more waiting and communication overhead, thus the straggler problem is still not addressed.

For ease of presentation, we denote the leaf nodes whose state $\neq$ POSTPONE as non-POSTPONE nodes. To devise a blocking-free work stealing mechanism, we first prove the following lemma:

Lemma 4.2: *Given Ψ on a machine M, for a node n in Ψ such that (1) n is not an ancestor of non-*POSTPONE *nodes, (2)* level$(n) \leq$ level(n')*, where n' represents any other node in Ψ that is not an ancestor of non-*POSTPONE *nodes, if the enumeration workload following root-to-n on machine M is migrated to another idle machine M', all* HC-s-t paths *can be correctly enumerated on M and M'.*

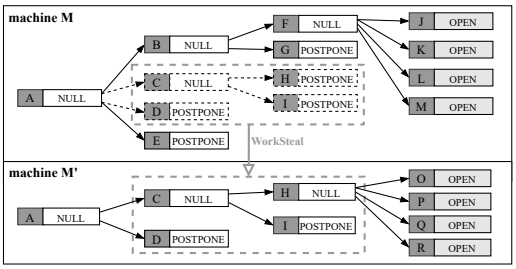


Figure 5: Dynamic Work Stealing

Following Lemma 4.2, when a machine M' finishes its work and becomes idle, it can automatically "steal" the workload from a busy machine M to accelerate the whole enumeration. Moreover, after the nodes are removed from Ψ on M due to the workload migration, there will be new nodes in the new Ψ satisfying the condition shown in Lemma 4.2, which means more workload can be further migrated. Inspired by this, we propose a dynamic work stealing mechanism as follows: (1) when a machine M' becomes idle, it sends a work steal request to a busy machine M. (2) When M receives the request, it (a) performs a BFS exploration on Ψ from the root until touching a node n which is not an ancestor of non-POSTPONE nodes. (b) retrieves the set of all the nodes at level(n) which are not ancestors of non-POSTPONE nodes, denoted by $\mathcal{T}$. (c) removes $\lceil \frac{|\mathcal{T}|}{2} \rceil$ number of nodes in $\mathcal{T}$ along with their descendants from Ψ. (d) sends the tree consisting of the tree paths of removed nodes to machine M'. (3) M' receives the tree from M and continues the enumeration.

Example 4.3: Figure 5 demonstrates an example of dynamic work stealing. When machine M' finishes its work and becomes idle, it automatically selects a random busy machine M to send a work steal request. Every node is shown with its alias and its vid, while the dashed nodes are those being "stolen" from M to M'. Specifically, when M receives the work steal request from M', it performs a BFS on Ψ from root until touching C which is not an ancestor of non-POSTPONE nodes. It further finds that on level(C) of Ψ, both D and E are not ancestors of non-POSTPONE nodes. Therefore, $\mathcal{T}$ is collected as $\{C, D, E\}$. To migrate $\lceil \frac{|\mathcal{T}|}{2} \rceil$ number of nodes in $\mathcal{T}$ to M', C and D with its descendants H and I are removed and sent to M' with corresponding tree paths. Then, M' receives the tree from M and continues the enumeration.

Lemma 4.3: *For a straggler machine M, the cost of computing the stolen nodes in Ψ after receiving a work steal request is $O(k\Delta)$.*

4.3 Further Optimization

As demonstrated in [43], by adopting a bidirectional search strategy, the centralized algorithm further improves enumeration performance. Hence, we further optimize the HybridEnum by adopting the same strategy in a distributed setting. The idea of the bidirectional search strategy in [43] is based on the path middle vertex.

Definition 4.5: (Path Middle Vertex) Given a path $p = (v_1, \ldots, v_n)$, the middle vertex of p is the $\lceil \frac{n}{2} \rceil$-th vertex of p.

With the path middle vertex, the enumeration-concatenation strategy contains four steps: (1) compute the set $\mathcal{M}$ of path middle vertices of all the HC-s-t paths. (2) add a virtual vertex t' and an edge (v, t') for each $v \in \mathcal{M}$ and compute HC-s-t$'$ paths P_l from s to t' with $\lceil \frac{k}{2} \rceil + 1$ hop constraint. (3) add a virtual vertex s' and an edge (s', v) for each $v \in \mathcal{M}$ and compute HC-s$'$-t paths P_r from s' to t with $\lfloor \frac{k}{2} \rfloor + 1$ hop constraint. (4) concatenate the paths from P_l and P_r based on $v \in \mathcal{M}$ and the concatenated path is HC-s-t path if and only if it is a simple path and v is its middle vertex.

Algorithm 5: HybridEnum$^+$ (s, t, k, Δ, G)

```
1  M ← ∅;
2  start two distributed BFS-oriented search from s/t on G/G^r;
3  foreach round i = 1 to ⌈k/2⌉) do
4  |  S_i ← i-hop reachable vertices from s on G;
5  |  T_i ← i-hop reachable vertices from t on G^r;
6  |  M ← M ⋃ (S_i ⋂ T_i);
7  s' ← |V| + 1; t' ← |V| + 2;
8  foreach v ∈ M do
9  |  add a virtual out-neighbor with id t' to v;
10 |  add a virtual in-neighbor with id s' to v;
11 P_l ← HybridEnum(s, t', ⌈k/2⌉ + 1, Δ, G) ; //omit t'
12 P_r ← HybridEnum(s', t, ⌊k/2⌋ + 1, Δ, G); //omit s'
13 distribute the paths in P_l/P_r with the same tail/head vertex
     to the same machine;
14 foreach Machine do
15 |  parallel foreach (p_l, p_r) ∈ P_l × P_r do
16 |  |  if p_l.tail() == p_r.head() then
17 |  |  |  if len(p_l) == len(p_r) or len(p_l) == len(p_r + 1)
   |  |  |  then
18 |  |  |  |  concatenate p_l and p_r into p;
19 |  |  |  |  if p has no repeated vertex then
20 |  |  |  |  |  output p;
```

Algorithm. Following the above idea, our optimized algorithm, HybridEnum$^+$, is shown in Algorithm 5. HybridEnum$^+$ first computes the set of path middle vertices $\mathcal{M}$ with two BFS-oriented searches (lines 1-6). After this, HybridEnum$^+$ adds the virtual vertices s' and t' in G (lines 8-10) and computes P_l and P_r by HybridEnum (lines 11-12). Lastly, HybridEnum$^+$ concatenates the paths in P_l and P_r to compute the HC-s-t paths (lines 13-20). The algorithm's correctness is straightforward following [43].

Theorem 4.3: HybridEnum$^+$ *further reduces the overall computation cost of* HybridEnum *on* G *to* $O(k|V(G)|\Delta(\phi_l+\phi_r)+\phi)$*, where* ϕ_l *and* ϕ_l *are the output path numbers in* P_l *and* P_r *respectively. The additional communication cost in* HybridEnum$^+$ *is* $O(\phi_l+\phi_r+k|V(G)|)$.

5 IMPLEMENTATION

5.1 Architecture

Our framework adopts a shared-nothing architecture in a cluster shown in Figure 6. Each machine has the following components:

Intermediate path tree. Each machine maintains an instance of intermediate path tree, which is defined in Definition 4.1.

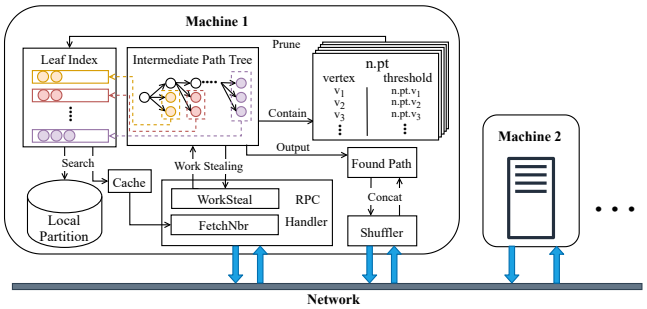


Figure 6: Architecture

k-layer leaf index. In each iteration, both PrunExt and BackProp apply on the leaf nodes which are on the same level of the intermediate path tree. Therefore, a k-layer index is maintained to store the leaf nodes on k levels using k vectors, and both PrunExt and BackProp can directly apply on the nodes in a leaf index level l, where l is changed by FlowCtrl at the end of each iteration.

RPC handler. RPC handler is used to communicate between machines, which is supported with a RPC server and a RPC client. The server is responsible for answering incoming requests from other machines, while the requests are sent through the client. There are two RPC functions, namely FetchNbr() and WorkSteal().

Cache. Each machine individually maintains a cache that stores the already fetched neighbors of remote vertices, which aims to reduce the communication cost during enumeration. The cache is shared among all workers in the same machine and is designed to be lock-free. When a machine starts to fetch remote vertices, only the vertices that are not in the cache are fetched remotely and the returned results are inserted into cache as $(v, \mathcal{N}(v))$ pairs. If the cache is full during the insertion of a returned pair, a random pair that is not used in the current round will be replaced.

Shuffler. The shuffler shuffles data across the machines, which is used in HybridEnum$^+$ to concatenate paths between different machines in line 13 of Algorithm 5.

5.2 Operator Implementation

This section presents the implementation details of the operators. The inputs of the operators include: Ψ that represents the intermediate path tree, I that represents the k-layer leaf index, C that represents the cache, l that represents the level of the intermediate path tree .

PrunExt. As shown in Algorithm 6, while the number of extended nodes is smaller than Δ and there is node left in the current level of I, we keep extending the nodes by their vids' out-going neighbors (line 1). In lines 2-4, the neighbors of remote vertices are fetched and stored into the cache. For each node n in I_l, if the length of root-to-n path has reached k, n's state is set as CLOSED with n.pt.(n.vid) set as 2 (lines 5-6). For a node n, if the length of I_{l+1} reaches Δ and the current extended neighbor is not the last element of $\text{nbr}^+(n)$ (line 23), n.children is added with an offset to record the number of out-neighbors that are not yet iterated (lines 24-25). After skipping the extended neighbors of n.vid based on the offset (line 8-11), if t is found to be a direct out-neighbor of n.vid, n.pt.(n.vid) is set as 2 and the graph path represented by the nodes is output (lines 12-13).

Algorithm 6: PrunExt (Ψ, I, C, l, B)

```
1  while I_{l+1}.length() ≠ Δ and I_l is not empty do
2      S_f ← remote vids in I_l[0..B];
3      foreach (u, N(u)) ∈ FetchNbr (S_f) do
4          C.insert(u, N(u));
5      parallel foreach n ∈ I_l[0..B] do
6          if l = k - 1 then n.pt.(n.vid) ← 2; continue;
7          foreach v' ∈ nbr+(n.vid) do
8              if n.children > deg+(n.vid) then
9                  n.children -= 1; continue;
10             else if n.children = deg+(n.vid) then
11                 n.children ← 0;
12             if v' = t then
13                 n.pt.(n.vid) ← 2; Output found path;
14             else if v' ∉ root-to-n graph path then
15                 pt_{v'} ← 0; n'' ← n;
16                 while n'' ≠ NULL do
17                     if v' ∈ n''.pt then
18                         pt_{v'} ← n''.pt.v'; break;
19                     else n'' ← n''.parent;
20                 if pt_{v'} + l + 1 ≤ k then
21                     n' ← Node(v', POTENTIAL, ∅, n, 0);
22                     n.children += 1; I_{l+1}.push(n');
23             if |I_{l+1}| = Δ and v' ≠ nbr+(n.vid).last() then
24                 offset ← 2 * deg+(n.vid) - count_rest;
25                 n.children += offset; break;
26         if 0 < n.children < deg+(n.vid) then
27             n.state := NULL; I.remove(n);
28             if I_{l+1}.length() = Δ then break;
29         else if n.children = 0 then
30             n.state = CLOSED; n.pt.(n.vid) ← k − l + 1;
```

Otherwise, after finding the pruning threshold of v' (lines 14-19), a pruning check is performed (line 20). If passed, a new node n' with tuple $(v', \text{POTENTIAL}, \emptyset, n, 0)$ is added as a child (lines 21-22). If no new nodes have been added as n's children, n.state is set as CLOSED and n.pt.(n.vid) is set as $k - l + 1$ (lines 29-30). Otherwise, n's state is set as NULL and n is removed from I_l (lines 26-27). PrunExt stops if Δ children are extended (line 28).

BackProp. As shown in Algorithm 7, for each node $n \in$ CLOSED nodes that need to be updated in S_B, let n' be its parent and pt_v be n.pt.(n.vid), n'.pt.(n'.vid) is firstly set as $\min\{n'.\text{pt}.(n'.\text{vid}), pt_v + 1\}$ (lines 1-6). If a valid path is found from n, S_u stores any vertex v where n.pt.v is possibly larger than $\text{dist}(v, t|S(n'))$ (line 7). Lines 8-18 iteratively update the pruning thresholds of the vertices in n.pt, in order to guarantee that $n.\text{pt}.v \le \text{dist}(v, t|S(n'))$ for all $v \in n$.pt. After updating the pruning thresholds of vertices in n.pt, n.pt is merged into n'.pt (lines 19-21). Then, n is removed and n' is added to S_N if n' subsequently becomes a leaf node (lines 22-23). Lastly, we assign S_N to S_B to repeat the update on new leaves (line 24).

FlowCtrl. FlowCtrl changes the value of l to represent the new index level that future operators will apply. The implementation of FlowCtrl is straightforward, which has been shown in Algorithm 3.

Algorithm 7: BackProp (Ψ, I, C, l)

```
1  S_B ← nodes referred in I_l with state CLOSED;
2  while S_B is not empty do
3      S_N ← ∅;
4      parallel foreach n ∈ S_B do
5          n' ← n.parent; pt_v ← n.pt.(n.vid); d ← 0; S_u ← ∅;
6          n'.pt.(n'.vid) ← min{n'.pt.(n'.vid), pt_v + 1};
7          if pt_v + l < k then S_u ← nbr−(n.vid);
8          while S_u is not empty do
9              d += 1; S_p, S_f ← ∅;
10             foreach v' ∈ S_u do
11                 if v' ∈ n.pt and n.pt.v' > pt_v + d then
12                     n.pt.v' ← pt_v + d; S_p ← S_p ∪ {v'};
13                     if v' is remote then S_f.push(v');
14             S_u ← ∅;
15             foreach (v', N(v')) ∈ FetchNbr (S_f) do
16                 C.insert(v', N(v'));
17             foreach v' ∈ S_p do
18                 foreach v'' ∈ nbr−(v') do S_u ← S_u ∪ {v''};
19         for v ∈ n.pt do
20             if v ∈ n'.pt then n'.pt.v ← max{n'.pt.v, n.pt.v};
21             else n'.pt.v ← n.pt.v;
22         Remove (n); n'.children -= 1;
23         if n'.children = 0 then S_N ← S_N ∪ {n'};
24     S_B ← S_N;
```

6 EVALUATION

In this section, we evaluate the efficiency of the proposed algorithms. All the experiments except Exp-5 are performed on a local cluster of 10 machines, each with one 4-core Intel Xeon CPU E3-1220, 64GB memory, 1T disk, connected via a 10Gbps network, running Red Hat Linux 7.3, 64 bit. Each machine runs 4 workers. For Exp-5, we use a machine with one 20-core Intel Xeon CPU E5-2698 and 768 GB main memory running Red Hat Linux 7.3, 64 bit.

Table 1: Statistic of the datasets

Dataset	Name	|V|	|E|	d_{max}	d_{avg}
web-Google	GO	875K	5M	6,332	5.0
LiveJournal	LJ	4M	68M	20,333	17.9
Twitter-WWW	TW	42M	1.46B	2,997,487	70.5
Friendster	FS	65M	1.81B	5,214	27.5
Twitter-MPI	TM	52M	1.96B	3,691,240	74.7
UK-2007	UK	134M	5.51B	6,366,528	41.2
Synthetic	SY	372M	10B	613,461	53

Datasets. We evaluate our algorithms on six real-world graphs and one synthetic graph. The size of the graphs is shown in Table 1. GO, LJ, TW and FS are downloaded from SNAP (http://snap.stanford.edu/data/index.html), TM is downloaded from KONECT (http://konect.cc/networks/) and UK is downloaded from LAW (http://law.di.unimi.it/datasets.php). SY is a synthetic power law graph generated by Graph500 generator [9]. Note that SY occupies roughly 80GB of space, and is larger than our machine's configured memory.

Due to the limited space, we show only part of the results, and the complete results can be found in our technical report [20].

Algorithms. We compare the following algorithms:

- DisBFS: The distributed BFS-oriented exploration algorithm for HC-s-t path enumeration.
- DOBFS/RQBFS/MSBFS: The distributed extensions of the Direction-Optimized/Read&Queue-Optimized/Multi-Source BFS algorithms [5], [21], [50] for HC-s-t path enumeration.
- DisHPI: The distributed extension of the HP-Index algorithm [45] for HC-s-t path enumeration.
- BiGJoin: The distributed HC-s-t path enumeration algorithm based on the subgraph matching system BiGJoin [2].
- Fractal: The distributed HC-s-t path enumeration algorithm based on the graph pattern mining system Fractal [14].
- HybridEnum: Our hybrid search-oriented enumeration algorithm (Algorithm 4 in Section 4.1).
- HybridEnum$^+$: Our bidirectional search optimized enumeration algorithm (Algorithm 5 in Section 4.3).
- T-DFS/T-DFS2/JOIN: The centralized HC-s-t path enumeration algorithms proposed in [47], [18], [43].

All the algorithms except Fractal are implemented in Rust 1.43. We implement DisBFS as discussed in Section 1. For DOBFS, RQBFS and MSBFS, we extend the techniques in [5], [21], and [50], respectively, to address the problem of HC-s-t path enumeration in a distributed setting through the Timely Dataflow engine [37]. For DisHPI, we communicated with the authors and implemented the distributed version with our best efforts. For BiGJoin and Fractal, we directly adopt their original implementations (from [29] and [15]) which are built on Timely Dataflow and Spark, respectively. For the HC-s-t path enumeration problem, their processing procedures consist of a number of intermediate path extensions. At each extension point, we add an extra conditional statement to ensure that the paths ending with t are output without extension. For T-DFS, T-DFS2 and JOIN, we implement them in Rust 1.43 following their original implementations. As big-data engines [37, 48] typically do not support pulling communication in HybridEnum and distributed key-value store [30] lacks the functionality of pushing communication in HybridEnum$^+$, we implement HybridEnum and HybridEnum$^+$ based on RPC [6].

In the experiments, we enable the workload balance mechanism for HybridEnum and HybridEnum$^+$ and set the cache size to 30% of the graph size by default. The time cost is measured as the amount of wall-clock time elapsed during the program's execution. If an algorithm cannot finish in 10,000 seconds or runs out of memory, we denote the processing time as INF. Moreover, if an algorithm runs out of memory, we also mark the case with a × on the top of the figure. In the experiments, we set the default hop constraint k as 6, and the queries are generated by randomly selecting pairs (s, t) such that s can reach t in k hops in the dataset.

Exp-1: Efficiency on different datasets. In this experiment, we evaluate the efficiency of the algorithms on different datasets. We randomly generate 100 query pairs for each dataset and report the average processing time for each query in Figure 7.

As shown in Figure 7, HybridEnum and HybridEnum$^+$ always outperform the other distributed algorithms. For example, on graph

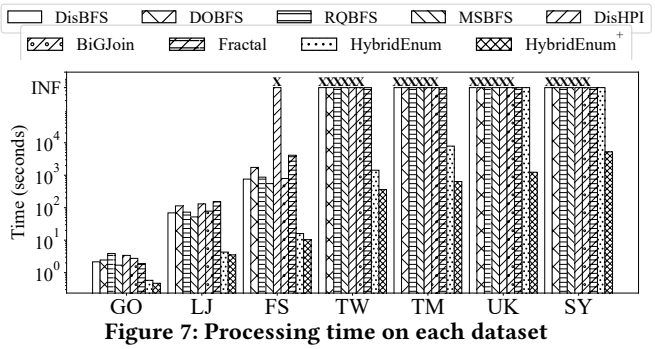


Figure 7: Processing time on each dataset

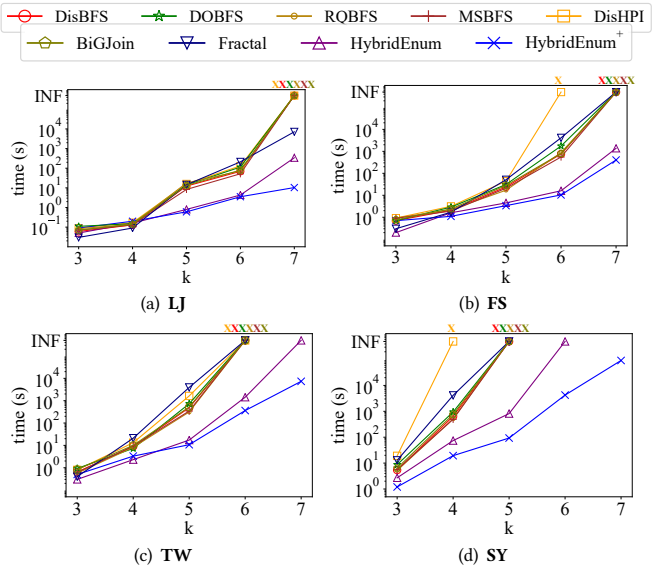
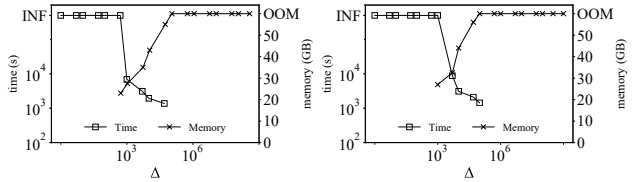


Figure 8: Processing time when varying hop constraint k

LJ, HybridEnum is 16.1× (resp. 26.5×, 16.9×, 12.1×, 30.5×, 18.1× and 35.1×) faster than DisBFS (resp. DOBFS, RQBFS, MSBFS, DisHPI, BiGJoin and Fractal). Comparatively, HybridEnum$^+$ demonstrates a better performance, which is 19.7× (resp. 32.5×, 20.9×, 14.9×, 37.1×, 22.4× and 60.3×) faster than DisBFS (resp. DOBFS, RQBFS, MSBFS, DisHPI, BiGJoin and Fractal). This is because the pruning technique used in our proposed algorithms can significantly reduce the search space and avoid the fruitless exploration. Moreover, due to the designed hybrid search paradigm and bounded memory consumption, when k becomes large, BFS-oriented algorithms (i.e. DisBFS, DOBFS, RQBFS, MSBFS and DisHPI) suffer from the out-of-memory issue while HybridEnum and HybridEnum$^+$ do not have this problem. For example, all BFS-oriented algorithms run out of memory on TW, TM, UK and SY, while HybridEnum and HybridEnum$^+$ never encounter memory crisis and can finish the enumeration efficiently on all graphs. This verifies the effectiveness of the hybrid search paradigm in controlling memory consumption. For BiGJoin and Fractal, they both perform worse than our algorithms due to the lack of ability to prune fruitless exploration. Although Fractal never encounters memory crisis because of its DFS-oriented design, it typically shows worse performance than BiGJoin due to its recomputing-from-scratch strategy. For HybridEnum and HybridEnum$^+$, HybridEnum$^+$ always outperforms HybridEnum on



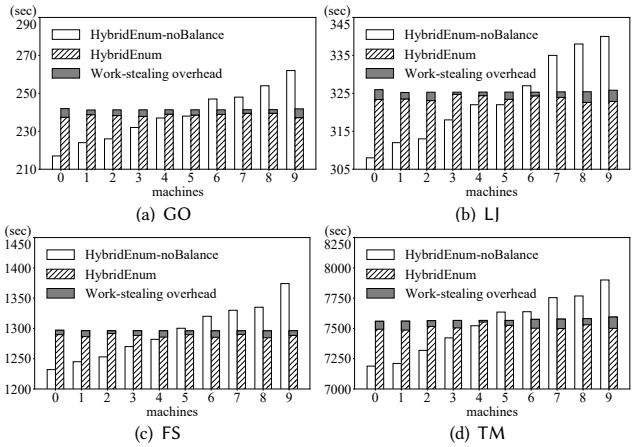
(a) **FS** (b) **TW**

Figure 9: Processing time and memory when varying Δ

all datasets, because by using the bidirectional search, some computed paths can be shared during enumeration, which is consistent with the analysis in [43].

Exp-2: Efficiency regarding hop constraint k. In this experiment, we evaluate the efficiency when varying hop constraint k from 3 to 7. For each k, we randomly generate 20 queries. The average processing time for each query is shown in Figure 8.

In Figure 8, as the hop constraint k increases, the processing time of all algorithms also increases. This is because as k increases, the number of HC-s-t paths also increases. Moreover, HybridEnum and HybridEnum$^+$ always outperform the other distributed algorithms, and the performance gap increases as the hop constraint k increases. This is because when the search space grows larger due to the increase of k, the fruitless exploration in the baseline algorithms increases accordingly, which is effectively avoided by the pruning techniques in HybridEnum and HybridEnum$^+$. Furthermore, as the hop constraint k increases, the memory crisis issue in BFS-oriented algorithms becomes serious while HybridEnum and HybridEnum$^+$ do not have this issue. The two distributed subgraph matching solutions BiGJoin and Fractal perform worse than our algorithms in all cases because of their lack of pruning. Based on the results, it is clear that HybridEnum and HybridEnum$^+$ are more scalable than the compared algorithms. HybridEnum$^+$ is more efficient than HybridEnum in all cases, the reason is similar to that in Exp-1.

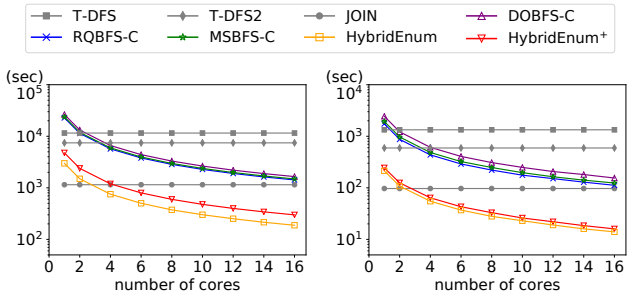


(a) GO (b) LJ

(c) FS (d) TM

Figure 10: Load Balancing

Exp-3: Efficiency and memory consumption regarding Δ. In this experiment, we evaluate the impact of Δ on our algorithm. We record the processing time and maximum memory consumption of the machine during the execution of HybridEnum by varying the value of Δ on FS and TW. The results are shown in Figure 9.



(a) **TM** (b) **UK**

Figure 11: Single machine comparison

Figure 9 shows that: (1) for the processing time, as Δ increases, the processing time decreases. This is because as Δ increases, the parallelism of HybridEnum becomes higher, and consequently the computing resources are more fully utilized. (2) for the memory consumption, as Δ increases, the memory consumption increases as well. This is because the size of intermediate path tree stored in each machine becomes large as Δ increases, which is consistent with our theoretical analysis on the memory consumption. Moreover, based on our proposed method for tuning Δ, $\Delta = 10^4$ and $\Delta = 10^{4.5}$ are chosen for FS and TW, respectively, which achieves relatively high parallelism while the memory consumption is still bounded.

Exp-4: Effectiveness of load balance. In this experiment, we evaluate the effectiveness of the load balance mechanism. We report the processing time for a query on each machine when the load balance mechanism is enabled or not on each dataset. The results are shown in Figure 10. We also evaluate the overhead of our work-stealing technique by measuring the time spent on work-stealing related code in each machine, shown by the grey filling.

As shown in Figure 10, without load balance mechanism, the straggler problem exists to some degree. For example, on TM, without the load balance mechanism, the running time of the fastest machine is 7189s while that of the slowest machine is 7904s. The results also find that the average time taken for work-stealing overhead occupies only 1.4% of the total processing time with load balance enabled. The experiment results demonstrate the effectiveness of our load balance mechanism.

Exp-5: Single machine comparison. In this experiment, we evaluate the performance of the parallel and distributed algorithms on a single machine, compared to the existing state-of-the-art single-threaded solutions to HC-s-t path enumeration. We evaluate the performance of the parallel algorithms by increasing the number of cores used, while the single-threaded algorithms are always executed with one core. DOBFS-C, RQBFS-C and MSBFS-C represent the centralized parallel extensions for HC-s-t path enumeration of the algorithms proposed in [5], [21], and [50], respectively.

As shown in Figure 11, HybridEnum and HybridEnum$^+$ always outperform the compared algorithms except JOIN. For the single-threaded algorithm JOIN, HybridEnum and HybridEnum$^+$ require 3 and 4 cores to outperform it in the worst-case, respectively. In contrast, the three multi-threaded BFS algorithms are typically slower than JOIN, even when running with 16 cores. This is because they were originally designed for BFS traversal and lack the power of pruning. In particular, DOBFS-C shows the worst performance. This is because its optimized search direction cannot

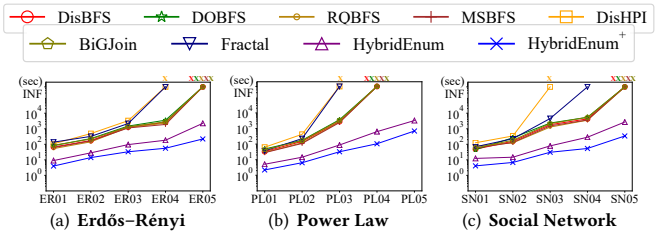


Figure 12: Processing time when varying graph size

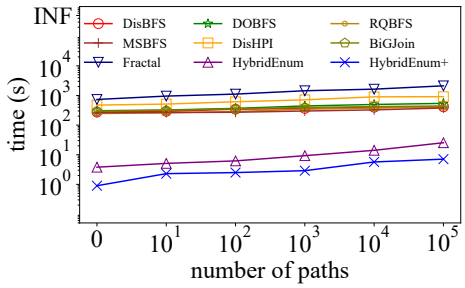


Figure 13: Efficiency regarding number of paths

practically reduce the number of edges explored due to the nature of path enumeration. RQBFS-C and MSBFS-C demonstrate improved performance over DOBFS-C, as they adopt effective strategies to reduce random memory access and utilize memory bandwidth more efficiently. Additionally, T-DFS and T-DFS2 always perform worse than JOIN due to their expensive verification cost.

Table 2: Statistic of the synthetic datasets

Erdős–Rényi (ER)	ER01	ER02	ER03	ER04	ER05
|V|	30M	60M	120M	240M	480M
|E|	99M	201M	400M	801M	1.6B
Power Law (PL)	PL01	PL02	PL03	PL04	PL05
|V|	3M	7M	12M	24M	49M
|E|	100M	200M	401M	800M	1.6B
Social Network (SN)	SN01	SN02	SN03	SN04	SN05
|V|	15M	31M	60M	118M	241M
|E|	101M	200M	399M	800M	1.6B

Exp-6: Efficiency regarding graph size. In this experiment, we evaluate the efficiency of the algorithms as the graph grows in size. We generate synthetic graphs following three commonly-used synthetic graph models: Erdős–Rényi graphs by the Erdős–Rényi graph generator [4], power law graphs by the Graph500 generator [9] and social network graphs by the LDBC Datagen [7]. The size of the graphs is shown in Table 2. We randomly generate 20 queries for each graph, with hop constraint k ranging from 4 to 7. Due to the limited space, we only present the average processing time with $k = 6$ for each query in Figure 12. The complete results are presented in our technical report.

As shown in Figure 12, when the graph size increases, the processing time of all the algorithms increases as well. This is because as the graph size increases, the number of HC-s-t paths and the size of the search space also increase. Moreover, it can be seen that HybridEnum and HybridEnum$^+$ outperform the other distributed algorithms in almost all cases, and the performance gap increases as the graph size increases. This is because when the search space grows larger as the size of graph increases, the fruitless computation in the baseline algorithms also increases, which is effectively avoided in HybridEnum and HybridEnum$^+$.

Exp-7: Efficiency regarding number of paths. In this experiment, we evaluate how the algorithms perform when the number of results for each query varies. We generate 120 queries with $k = 5$ on TW, where the number of results varies from 0 to 10^5. The average processing time for each query is shown in Figure 13.

As can be seen in Figure 13, when the number of query results increases, the processing time of HybridEnum and HybridEnum$^+$ increases accordingly. This is because when the number of valid paths is smaller, there is more fruitless computation to be avoided during the enumeration and hence, HybridEnum and HybridEnum$^+$ perform better. In contrast, the remaining algorithms are less sensitive to the number of path results.

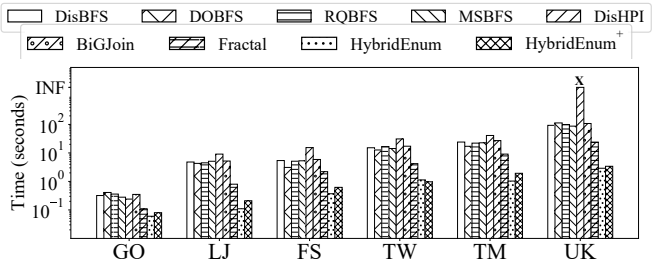


Figure 14: Processing time to the first solution

Exp-8: Time to the first solution. In this experiment, we report the time that the algorithms take to output their first solution. We randomly generate 20 query pairs for each dataset and report the average processing time for each query in Figure 14.

Figure 14 shows that the performance of our algorithms are always better than the others regarding outputting the first solution. Moreover, it can be seen that HybridEnum typically performs better than HybridEnum$^+$ in terms of the time to the first solution. This is due to the overhead of computing path middle vertices in HybridEnum$^+$, which involves two distributed BFS-oriented exploration from s/t on G/G^r.

7 CONCLUSION

In this paper, we study the problem of distributed HC-s-t path enumeration. We first propose a novel hybrid search paradigm. Based on it, we devise a new distributed HC-s-t path enumeration algorithm, HybridEnum, following the divide-and-conquer strategy. In addition, we design an effective work stealing mechanism to handle unbalanced workload. The experiment results demonstrate the efficiency and scalability of our proposed algorithms.

ACKNOWLEDGMENTS

Long Yuan is supported by NSFC61902184 and NSF of Jiangsu Province BK20190453, Science and Technology on Information Systems Engineering Laboratory WDZC20205250411. Wenjie Zhang is supported by DP210101393.

REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming Graph Partitioning: An Experimental Study. *Proc. VLDB Endow.* 11, 11 (July 2018), 1590–1603. https://doi.org/10.14778/3236187.3236208

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704. https://doi.org/10.14778/3184470.3184473

[3] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.

[4] David A Bader and Kamesh Madduri. 2006. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February* 38 (2006).

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. 2012. Direction-optimizing breadth-first search. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE/ACM, 12. https://doi.org/10.1109/SC.2012.50

[6] Andrew Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (1984), 39–59. https://doi.org/10.1145/2080.357392

[7] Peter Boncz. 2013. LDBC: benchmarks for graph and RDF data management. In *Proceedings of the 17th International Database Engineering & Applications Symposium*. 1–2.

[8] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1456–1465.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[10] Zi Chen, Long Yuan, Xuemin Lin, Lu Qin, and Jianye Yang. 2020. Efficient Maximal Balanced Clique Enumeration in Signed Networks. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 339–349. https://doi.org/10.1145/3366423.3380119

[11] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 219–230.

[12] LAW Dataset. EU-2015. http://law.di.unimi.it/webdata/eu-2015/.

[13] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: an integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 2. https://doi.org/10.1145/2960414.2960416

[14] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*. 1357–1374.

[15] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. *Fractal Open Source Implementation.* https://github.com/dccspeed/fractal

[16] André Freitas, João Carlos Pereira da Silva, Edward Curry, and Paul Buitelaar. 2014. A Distributional Semantics Approach for Selective Reasoning on Commonsense Graph Knowledge Bases. In *Natural Language Processing and Information Systems - 19th International Conference on Applications of Natural Language to Information Systems, NLDB 2014, Montpellier, France, June 18-20, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8455)*, Elisabeth Métais, Mathieu Roche, and Maguelonne Teisseire (Eds.). Springer, 21–32. https://doi.org/10.1007/978-3-319-07983-7_3

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[18] Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Efficient Algorithms for Listing k Disjoint st-Paths in Graphs. In *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10807)*, Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro (Eds.). Springer, 544–557. https://doi.org/10.1007/978-3-319-77404-6_40

[19] Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. 2019. PatMat: a distributed pattern matching engine with cypher. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2921–2924.

[20] Kongzhang Hao, Long Yuan, and Wenjie Zhang. 2021. *Technical Report.* https://www.dropbox.com/s/3ea2h9azy5llpjp/technical%20report.pdf?dl=0

[21] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, Lawrence Rauchwerger and Vivek Sarkar (Eds.). IEEE Computer Society, 78–88. https://doi.org/10.1109/PACT.2011.14

[22] Will McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel Distributed Comput.* 65, 8 (2005), 901–910. https://doi.org/10.1016/j.jpdc.2005.03.007

[23] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[24] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[25] Arijit Khan, Gustavo Segovia, and Donald Kossmann. 2018. On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 401–412. https://www.usenix.org/conference/atc18/presentation/khan

[26] 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. https://doi.org/10.1093/bioinformatics/btg113

[27] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228. https://doi.org/10.14778/3021924.3021937

[28] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228. https://doi.org/10.14778/3021924.3021937

[29] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112. https://doi.org/10.14778/3339490.3339494

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

[31] Ni Lao and William W. Cohen. 2010. Relational retrieval using a combination of path-constrained random walks. *Mach. Learn.* 81, 1 (2010), 53–67. https://doi.org/10.1007/s10994-010-5205-8

[32] Ulf Leser. 2005. A query language for biological networks. In *ECCB/JBI'05 Proceedings, Fourth European Conference on Computational Biology/Sixth Meeting of the Spanish Bioinformatics Network (Jornadas de BioInformática), Palacio de Congresos, Madrid, Spain, September 28 - October 1, 2005*. 39. https://doi.org/10.1093/bioinformatics/bti1105

[33] Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. 2019. TopoX: Topology refactorization for efficient graph partitioning and processing. *Proceedings of the VLDB Endowment* 12, 8 (2019), 891–905.

[34] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (a,β)-core Computation: an Index-based Approach. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 1130–1141. https://doi.org/10.1145/3308558.3313522

[35] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *VLDB J.* 29, 5 (2020), 1075–1099. https://doi.org/10.1007/s00778-020-00606-9

[36] Sahisnu Mazumder and Bing Liu. 2017. Context-aware Path Ranking for Knowledge Base Completion. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 1195–1201. https://doi.org/10.24963/ijcai.2017/166

[37] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[38] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[39] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *Proc. VLDB Endow.* 13, 5 (2020), 602–615. https://doi.org/10.14778/3377369.3377371

[40] Anil Pacaci and M. Tamer Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1375–1392. https://doi.org/10.1145/3299869.3300076

[41] Peng Peng, Lei Zou, and Runyu Guan. 2019. Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 112–123. https://doi.org/10.1109/ICDE.2019.00019

[42] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. 2021. Answering reachability and K-reach queries on large graphs with label constraints. *The VLDB Journal* (2021), 1–27.

[43] 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. http://www.vldb.org/pvldb/vol13/p463-peng.pdf

[44] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient probabilistic k-core computation on uncertain graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1192–1203.

[45] 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. https://doi.org/10.14778/3229863.3229874

[46] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and Robust Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 12, 11 (July 2019), 1344–1356. https://doi.org/10.14778/3342263.3342272

[47] Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. 2014. Efficiently Listing Bounded Length st-Paths. In *Combinatorial Algorithms - 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8986)*, Jan Kratochvíl, Mirka Miller, and Dalibor Froncek (Eds.). Springer, 318–329. https://doi.org/10.1007/978-3-319-19315-1_28

[48] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[49] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 425–440. https://doi.org/10.1145/2815400.2815410

[50] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (2014), 449–460. https://doi.org/10.14778/2735496.2735507

[51] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2021. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* (2021), 1–24.

[52] Kai Wang, Shuting Wang, Xin Cao, and Lu Qin. 2020. Efficient radius-bounded community search in geo-social networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[53] Kai Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Discovering Significant Communities on Bipartite Graphs: An Index-based Approach. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[54] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 136–147. https://doi.org/10.1109/ICDE.2019.00021

[55] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2016. Diversified top-k clique search. *VLDB J.* 25, 2 (2016), 171–196. https://doi.org/10.1007/s00778-015-0408-z

[56] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Effective and Efficient Dynamic Graph Coloring. *Proc. VLDB Endow.* 11, 3 (2017), 338–351. https://doi.org/10.14778/3157794.3157802

[57] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2018. Index-Based Densest Clique Percolation Community Search in Networks. *IEEE Trans. Knowl. Data Eng.* 30, 5 (2018), 922–935. https://doi.org/10.1109/TKDE.2017.2783933

[58] Dianmin Yue, Xiaodan Wu, Yunfeng Wang, Yue Li, and Chao-Hsien Chu. 2007. A review of data mining-based financial fraud detection research. In *2007 International Conference on Wireless Communications, Networking and Mobile Computing*. Ieee, 5519–5522.

[59] Xiaofei Zhang, Hong Cheng, and Lei Chen. 2015. Bonding Vertex Sets Over Distributed Graph: A Betweenness Aware Approach. *Proc. VLDB Endow.* 8, 12 (2015), 1418–1429. https://doi.org/10.14778/2824032.2824041

[60] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 301–316.