Hop-constrained s-t Simple Path Enumeration: Towards Bridging Theory and Practice

You Peng
The University of New South Wales
unswpy@gmail.com

Ying Zhang
The University of Technology Sydney
ying.zhang@uts.edu.au

Xuemin Lin
The University of New South Wales
lxue@cse.unsw.edu.au

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

Lu Qin
The University of Technology Sydney
lu.qin@uts.edu.au

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com

ABSTRACT

Graph is a ubiquitous structure representing entities and their relationships applied in many areas such as social networks, web graphs, and biological networks. One of the fundamental tasks in graph analytics is to investigate the relations between two vertices (e.g., users, items and entities) such as how a vertex A influences another vertex B, or to what extent A and B are similar to each other, based on the graph topology structure. For this purpose, we study the problem of hop-constrained s-t simple path enumeration in this paper, which aims to list all simple paths from a source vertex s to a target vertex t with hop-constraint k. We first propose a polynomial delay algorithm, namely BC-DFS, based on barrier-based pruning technique. Then a join-oriented algorithm, namely JOIN, is designed to further enhance the query response time. On the theoretical side, BC-DFS is a polynomial delay algorithm with $O(km)$ time per output where m is the number of edges in the graph. This time complexity is the same as the best known theoretical result for the polynomial delay algorithms of this problem. On the practical side, our comprehensive experiments on 15 real-life networks demonstrate the superior performance of the BC-DFS algorithm compared to the state-of-the-art techniques. It is also reported that the JOIN algorithm can further significantly enhance the query response time.

PVLDB Reference Format:
You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. Hop-constrained s-t Simple Path Enumeration:Towards Bridging Theory and Practice. *PVLDB*, 13(4): 463 -476, 2019.
DOI: https://doi.org/10.14778/3372716.3372720

1. INTRODUCTION

Graph is a ubiquitous structure representing entities and their relationships applied in many areas such as social networks [45, 46, 65, 64, 41, 51], web graphs, and biological networks. One of the fundamental problems in graph analytics [28, 7, 18, 16, 11, 14, 15, 17, 12, 13] is to investigate the relations between two given vertices (e.g., users or entities) in a graph such as how a vertex s influences another vertex t, or to what extent s and t are similar to each other according to the network topology. In addition to a few metrics such as shortest path distance (e.g., [20, 30, 72]), these relations can be naturally captured by a set of paths from s to t In some applications, we may derive a score based on these paths to capture their relations (e.g., similarity) This is the problem of s-t path enumeration, which has a long study history in the literature (e.g., [3, 8, 36, 50, 68, 69]).

In many real-life applications, it is rather natural to impose a hop constraint, say k, to s-t path enumeration because: (1) it is usually not interesting/meaningful for users to identify the s-t path with a large number of hops because the strength of the relation drops dramatically with the number of hops; and (2) it is well-known that the number of paths may grow exponentially w.r.t the number of hops in real-life graphs. A user may be overwhelmed by a huge number of paths without a proper hop constraint. Moreover, same as the most existing studies on path enumeration, in this paper we consider the *simple* path since a path with loop (i.e., with repeated vertices along the path) is less interesting and may significantly increase the total number of s-t paths.

Motivated by these, in this paper we study the problem of hop-constrained s-t *simple* path (hc-s-t path for short) enumeration. More specifically, given an unweighted directed graph G, a source vertex s, and a target vertex t, we aim to enumerate all simple paths from s to t with number of hops not larger than k.

Applications. Below are three representative real-life applications for the problem of hc-s-t path enumeration.

(1) Biological Networks. Pathway queries are essential in the biological networks analytics, where vertices represent the entities such as enzymes and genes while edges represent some forms of interactions or relations [40, 43]. As shown in [43], s-t path enumeration is one of the four important pathway queries in biological networks. Given two substances s and t, it finds all paths, i.e., chains of interactions, from s to t. In the pathway query language proposed in [43], a hop constraint can be used in the path expression such that users can focus on a limited number of important paths (i.e., interaction chains) between two substances.

Proceedings of the VLDB Endowment, Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372720

(2) E-commerce Networks. In a recent industry paper from Alibaba group [52], the activities of the electronic payment system and relationships among users are modeled as a directed graph, in which each vertex represents an entity (e.g., person or account) and each edge $A \rightarrow B$ indicates the relations from A to B such as A issues a payment to B or A is a friend/family member of B. A circle in such a graph suggests an interesting relationship among the involved personnels, which is a strong indication of a fraudulent activity or even a finance crime like money laundering [71]. For instance, when a new edge $t \rightarrow s$ is inserted (e.g., a new money transfer transaction), the currently finance risk control system in Alibaba group needs to report the resulting new cycles for further risk analysis. These new cycles can be detected by enumerating all *simple* paths from s to t. In the system, it is critical to impose a hop constraint (e.g., k is set to 6 in [52]). Otherwise the system will be overwhelmed by a huge number of false alarms.

(3) Knowledge Graphs. Path ranking (PR) algorithms (e.g., [42, 19, 49]) have received increasing attentions in recent years, which enumerate paths from one entity to another in a knowledge graph and use those paths as features to train a model for missing fact prediction [49]. Intuitively, a long path is not useful to capture the tie between two entities because the relation strength usually drops dramatically with the number of hops (i.e., interactions). For instance, a hop-constraint is imposed in [19] when the paths between two terms are enumerated.

In addition to the above applications, the problem of *hc-s-t* path enumeration can find many other applications. For instance, one may need to enumerate *hc-s-t* paths to investigate the possible connections between two entities such as two companies for background check in business networks, two suspects for risk analysis in social networks, two failure points for failure analysis in the infrastructure dependency networks, etc.

It is worth to mention that, in addition to hop constraints, other constraints regarding other features of the *s-t* path may be considered for *s-t* path enumeration in some real-life applications. For instance, one may apply the label constraint in the biological network applications such that only some specific types of enzymes or reactions are considered. As shown in [52], one may ignore the transactions with less amount of money in fraud detection in the e-commerce network. Meanwhile, one may enforce the time order of the transactions along the s-t path, e.g., disregard a transaction if it comes earlier than the currently visited edge (transaction) during the search. In Section 5.2, we show that our proposed solutions for hop-constrained s-t simple path can be easily extended to support these constraints.

Challenges. The main obstacle in solving the hc-*s-t* path enumeration problem is the huge search space involved even for a small k value because the number of hc-*s-t* paths may grow exponentially w.r.t k. Another subtle difficulty lies in the duplication and loop checks which ensure the output are simple and unique hc-*s-t* paths. It is less efficient in terms of response time and memory usage if we simply enumerate all *s-t* paths with duplicates and loops, and then verify them. Moreover, same as other enumeration problems with possibly large size of output, it is desirable to develop *polynomial delay* algorithm [32] such that the delay between the output of two consecutive hc-*s-t* paths is bounded by a polynomial time.

As discussed in Section 2, there are some existing algorithms closely related to the problem of hc-*s-t* enumeration. However, their solutions are not suitable to the problem studied in this paper due to inherently different research focuses or problem settings. For instance, the *s-t* simple path enumeration algorithm *simpath* is presented by Knuth in [36]. However, the focus of the algorithm is the succinct presentation of the huge volume of *s-t* simple paths.

As an alternative, one may repeatedly apply a top k' shortest path algorithm (e.g., [21, 6]) by increasing k' until the number of paths retrieved exceeds the hop constraint k. As shown in [21], it takes $O(km)$ time to retrieve one shortest path in each iteration. Thus, the extended algorithm is a polynomial delay algorithm for the problem of hop-constrained s-t simple path enumeration, with time complexity $O(km\delta)$, where δ is the number of valid paths. In the experiments, two latest algorithms [21, 6] on the top k' shortest path search are employed, and their performance is not competitive compared to the proposed solutions.

State-of-the-art. To the best of our knowledge, two polynomial delay algorithms for the problem of hc-*s-t* path enumeration are proposed in two theoretical papers [24, 54], namely **T-DFS** and **T-DFS2** in this paper. Their theoretical results are quite attractive, with $O(km)$ time per output where m is the number of edges. Thus, the time complexity of their algorithms is $O(km\delta)$ where δ is the number of hc-*s-t* paths. However, our empirical study shows that their practical performance is disappointing due to the expensive checking costs involved.

Recently, Qiu *et al.* from Alibaba Group study the problem of hop-constrained simple cycle detection [52], in which their technique, namely HP-Index, is used to enumerate hc-*s-t* paths. However, the time complexity of `HP-Index` remains $O(n^k)$ in the worst case.

Our Approaches. In this paper, we propose two hc-*s-t* path enumeration algorithms, namely BC-DFS and JOIN, with both nice theoretical time complexity and excellent practical performance.

(1) *BC-DFS.* A common practice to develop polynomial delay algorithm for enumeration problem is to ensure that each search branch has at least one output, which is used by T-DFS and T-DFS2 algorithms in [24, 54]. However, we observe that this strategy may lead to high overhead in our problem, which results in the poor practical performance. In Section 4, we develop a simple and elegant barrier-pruning based DFS algorithm, namely **BC-DFS**. The philosophy of the algorithm is "*do not fall in the same trap twice by learning from mistakes*"; that is, we allow the failure (i.e., explore some search branches without output), but ensure to learn from it (i.e., avoid exploring non-promising search branch in the future). We also propose barrier level optimization techniques to enhance the performance of BC-DFS. On the theoretical side, we show that BC-DFS is a polynomial delay algorithm with $O(km)$ time per output.

(2) *JOIN.* In some scenarios, users may be more interested in the overall response time. Thus, we also develop a join-oriented algorithm, namely JOIN, which is efficient in practice, especial for large k values. The philosophy of JOIN is "*share the computation by divide and conquer*". We observe that DFS-oriented technique enumerates the hc-*s-t* paths one by one and hence cannot effectively share the computation. This motivates us to develop join-oriented technique which joins (i.e., concatenates) partial paths based on some cutting vertices. By doing this, the computation of the partial paths can be shared and hence significantly improve the query response time.

Contributions. Our principal contribution in this paper is summarized as follows.

- We develop a DFS-oriented approach, namely BC-DFS, based on the barrier-based pruning technique. We show that BC-DFS is a polynomial delay algorithm with $O(km)$ time per output (i.e., hc-s-t path) where m is the number of edges and k is the hop constraint. The time complexity is $O(km\delta)$ where δ is the number of hop-constrained s-t *simple* paths.
- We design a join-oriented approach, namely JOIN, based on the BC-DFS and middle vertexes based cut techniques. The time complexity of JOIN is $O(km\alpha)$ where α is the number of hop-constrained s-t paths.
- Our comprehensive experiments on 10 real-life graphs demonstrate the superior performance of our proposed methods, compared to the baseline solutions. Particularly, we show that our proposed BC-DFS algorithm has the same time complexity with two recent polynomial delay algorithms but much better practical performance, with up to 2 orders of magnitude speedup. It is also reported that JOIN can further significantly improve the query response time by computation sharing.

Roadmap. The rest of the paper is organized as follows. Section 2 surveys important related work. Section 3 formally defines the problem, and describes the baseline solutions. Section 4 designs a DFS-oriented approach by applying barrier-based pruning technique. A join-oriented approach is proposed in Section 5, followed by the empirical study in Section 6. Section 7 analyzes the time complexity of two existing polynomial delay algorithms. Section 8 concludes the paper.

2. RELATED WORK

In this section, we review closely related works.

2.1 Simple Path Enumeration and Counting

There are some existing works on the problem of enumerating s-t simple paths (e.g., [4, 36, 50, 68]) including the *simpath* algorithm introduced by Knuth in [36]. However, their research focus is the succinct presentation of these paths; that is, given a huge number of paths between two vertices s and t, how to construct a succinct presentation of these simple paths such that we can efficiently enumerate the simple paths without explicitly store each individual path. Note that their algorithms are not competitive for the problem of s-t simple path enumeration, and can only handle small scale graphs with thousands of vertices. In [3], Birmele *et. al* studied the problem of s-t simple path enumeration, but their solution only supports the undirected graphs. Two polynomial delay algorithms are proposed in [24, 54], which take $O(km)$ time per output. The counting of s-t simple paths is a well-known $\#P$ hard problem, which has been studied with different approaches such as recursive expressions of the adjacency matrix (e.g., [8, 22, 35]) and immanantal equations [5]. Nevertheless, they cannot be trivially extended to efficiently enumerate hop-constrained simple paths without materializing the paths during the computation, which will easily blow up the main memory even for a small k.

In addition to HP-index technique introduced in Section 3.2, there are also many existing work to enumerate the simple cycles of the graph (e.g., [2, 31, 37, 38, 59]). Nevertheless, these technique cannot be trivially extended to efficiently solve our problem because none of them considers the length constraint.

2.2 Shortest Path Enumeration

In addition to the classical s-t shortest path computation (see [70] and [60] for a comprehensive survey), there are several variants in which a set of paths are considered.

The problem of top-k' shortest path has been intensively studied in the literature (e.g., [10, 23, 26, 34, 48, 55, 69]). Yen's algorithm [69] is the most representative work with good practical performance and theoretical analysis. An efficient implementation of Yen's algorithm is proposed in [9]. It is immediate that we can keep on invoking the top k' shortest simple path algorithm by increasing k' until the shortest path detected exceeds the distance threshold k. However, our initial experiments show that this method is not competitive even compared with our slowest method in this paper because we have to enforce the output order of the paths according to their distances.

A considerable amount of literature has been published on constrained shortest path [1, 25, 29, 33, 53, 57, 58, 62, 66, 67]. This problem can be defined as finding the shortest path between two vertices on a network whenever the traversal of any arc/vertex consumes certain resources and the resources consumed along the path chosen must lie within given limits(both lower and upper limits). The problem of diversified shortest path has been intensively studied in the literature as well (e.g., [47, 61, 63]) which consider both distance and diversity of the s-t shortest paths. Nevertheless, these techniques cannot be used to efficiently enumerate hc-s-t paths due to the inherent difference between those problems.

3. PRELIMINARY

In this section, we first formally introduce the problem of hc-s-t path enumeration, then present four baseline solutions. In Table 1, we summarize the important mathematical notations appeared throughout this paper.

Table 1: The summary of notations

Notation	Definition
G, G^r	a given graph, its reverse graph
p, $p(u,v)$, $p(u \rightsquigarrow v)$	a path in G, a path from u to v
s, t	source and target vertices
hc-s-t path	hop-constrained s-t simple path
$len(p)$	length (i.e., number of hops) of path p
$sd(u,v)$	shortest path distance from u to v i.e., minimal number of hops from u to v
$sd(u,v\|T)$	shortest path distance from u to v not containing any vertex in T
k	hop constraint
P_k	hc-s-t paths returned
$p[x]$	number of hops the vertex x can reach the end vertex of p along the path p
$\mathcal{S}$, $\|\mathcal{S}\|$	the stack in DFS and its size
$p(\mathcal{S})$	the path associated with stack $\mathcal{S}$
$len(\mathcal{S})$	the length of the path associated with $\mathcal{S}$ where $len(\mathcal{S}) = len(p(\mathcal{S})) = \|\mathcal{S}\| - 1$
P_m	the middle vertex cut

3.1 Problem Definition

Let $G = (V, E)$ denote a directed graph, where V is the set of n vertices and $E \subseteq V \times V$ is a set of m directed edges. We assume $m \geq n$ in this paper. We use $e(u, v) \in E$ to denote a directed edge from the vertex u to the vertex v. When the context is clear, we use "neighbor" to refer the "out-going neighbor". By $G^r = (V, E^r)$, we denote the reverse graph of G, which reverses the direction of all edges in G. A *path* p from the vertex v to the vertex v' is a sequence of vertices

$v = v_0, v_1, \ldots, v_h = v'$ such that $(v_{i-1}, v_i) \in E$ for every $i \in [1, h]$. In this paper, we use $p(u, v)$ or $p(u \rightsquigarrow v)$ to denote a path from u to v. A *simple* path is a loop-free path where there are no repetitions of vertices and edges. By $len(p)$, we denote the length (i.e., the number of hops in this paper) of the path p. We say a path p is a hop-constrained path if $len(p) \leq k$ where k is the pre-defined hop constraint. For presentation simplicity, we use hc-s-t path to denote hop-constrained s-t simple path. Given two vertices u and v, we use $sd(u, v)$ to denote the shortest path distance (i.e., minimal number of hops) from u to v in the graph G. We use $sd(u, v|T)$ to denote the shortest path distance from u to v, *not containing any vertex in* T.

Problem Statement. In this paper, we study the problem of hop-constrained s-t simple path enumeration on a directed graph. Specifically, given a directed graph G, an integer k, the source vertex s and the target vertex t, we use P_k to denote the set of hc-s-t paths where $P_k = \{p \mid len(p) \leq k, p \text{ is a simple path, and } s \text{ and } t \text{ are start and end vertices of } p\}$. we aim to develop efficient algorithms to enumerate all paths in P_k.

3.2 Baseline Solutions

In this subsection, we introduce four baseline solutions.

Algorithm 1: **C-DFS** (u, $\mathcal{S}$)

```
   Input  : u : the vertex to be processed ;
            S: the stack
1  S.push(u);
2  if u == t then
3      output p(S) where P_k := P_k ∪ p(S);
4  else if len(S) < k then
5      for each out-going neighbor v of u where v ∉ S do
6          C-DFS(v, S);
7  u is unstacked from S;
```

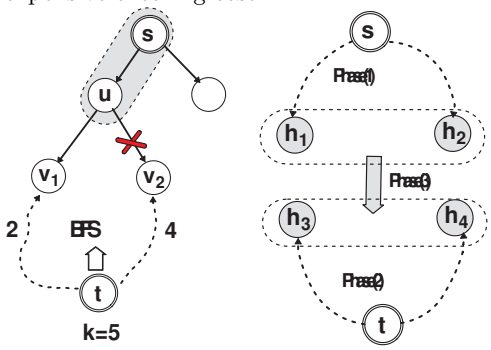
(1) Hop-Constrained Depth-First-Search (C-DFS). As illustrated in Algorithm 1, a straightforward solution is to conduct a Depth-First-Search (DFS) starting from source vertex s with search depth at most k. A stack $\mathcal{S}$ is deployed for the search where $|\mathcal{S}|$ denotes the number of vertices in $\mathcal{S}$ and $p(\mathcal{S})$ denotes the corresponding path in $\mathcal{S}$ with length $len(\mathcal{S}) = |\mathcal{S}| - 1$. Note that we enforce that the search does not touch the vertices already in the current stack $\mathcal{S}$ and hence the output is loop-free (Line 5). The time complexity of the Algorithm 1 is $O(n^k)$.

(2) DFS with polynomial delay (T-DFS and T-DFS2). In [54] an extension of C-DFS, namely T-DFS in this paper, can be used to enumerate hc-s-t paths for the directed graph. Particularly, T-DFS carefully explores the out-going neighbors of a vertex and ensures that every search branch in the DFS comes up with at least one hc-s-t path (i.e., *never fall in the trap*). For each out-going neighbor v of u to be visited in C-DFS (Line 5 in Algorithm 1), T-DFS will compute its shortest path distance to t, without containing any vertex in the stack $\mathcal{S}$, i.e., $sd(v, t|\mathcal{S})$. The vertex v will not be explored if $|\mathcal{S}| + sd(v, t|\mathcal{S}) > k$.

Example. We give an example in Figure 1(a). In this example, the hop constraint $k = 5$, and search stack $\mathcal{S} = \{s, u\}$ while v_1 and v_2 are two out-going neighbors of u. If we use C-DFS (Algorithm 1), both v_1 and v_2 will be explored in the following computation. However, v_2 will not be visited since we have $|\mathcal{S}| = 2$, $sd(v_2, t|\mathcal{S}) = 4$ and $2 + 4 > 5$ in T-DFS.

Note that the shortest path distance computation for all out-going neighbors of u can be completed by one BFS on the reversed graph, with time $O(m)$. By aggressively checking if each search branch is promising, T-DFS guarantees that there is at least one hc-s-t for each search branch explored. As the search depth is at most k for each search branch, T-DFS is a polynomial delay algorithm [32] with $O(km)$ time per output. Thus, the time complexity of T-DFS is $O(km\delta)$ where δ is the number of hc-s-t paths. Recently, Grossi et al. study the problem of listing k disjoint s-t paths in [24], and a new algorithm, namely T-DFS2 in this paper, is proposed for hc-s-t path enumeration following the same aggressive checking strategy. It can reduce shortest path distance computation by skipping some vertices associated with only one output in the following search. In Section 7, we analyze the time complexity of two algorithms and show the performance of T-DFS2 is the same as T-DFS, i.e., $O(km)$ time per output, on the directed graph.

As demonstrated in our empirical study, the performance of two theoretical studies is not competitive in practice due to the expensive checking cost.



(a) T-DFS (b) HP-Index

Figure 1: Key idea of T-DFS and HP-Index

(3) Hot-Point index based algorithm (HP-Index). In [52], a novel indexing technique called HP-index has been proposed to continuously maintain the pairwise hc-s-t paths among a set of hot points (i.e., vertices with high degree). By applying a bi-directed search and hot point indexing technique, their search algorithm can avoid expanding the high-degree vertices and hence may reduce the response time. For an incoming edge (t, s), they can identify the new hop-constrained simple cycles by enumerating hc-s-t paths as follows:

1. Conduct a C-DFS starting from s but does not explore the hot points encountered, e.g., h_1 and h_2 as illustrated in phase (1) of Figure 1(b);
2. Conduct a reversed C-DFS starting from t in the same way, e.g., h_3 and h_4 as illustrated in phase (2) of Figure 1(b);
3. Find hc-s-t paths among the hot points involved in the above computation based on the HP-Index, as illustrated in phase (3) of Figure 1(b);
4. Concatenate the paths from steps (1), (2) and (3) to identify hc-s-t paths.

As reported in [52], HP-index demonstrates better performance compared with C-DFS on the Alibaba transaction network data. However, we observe that HP-Index can only achieve good performance on the extremely skewed graph

data which has relatively small number of hop-constrained paths (e.g., Amazon network in our empirical study) due to the nature of HP-Index. Moreover, the time complexity remains $O(n^k)$.

4. BARRIER-BASED CONSTRAINED DFS (BC-DFS)

In this section, we present a polynomial delay algorithm, namely BC-DFS.

4.1 Motivation

As discussed in Section 3.2, T-DFS and T-DFS 2 are over-pessimistic because they enforce that every search branch has at least one output with high checking overhead. On the contrary, C-DFS is over-optimistic and never learns from the mistakes. The general idea of our approach is "*do not fall in the same trap twice by learning from mistakes*", which can be regarded as a trade-off between C-DFS and T-DFS in the sense that we allow our algorithm to explore fruitless branches, but we also learn from the mistakes.

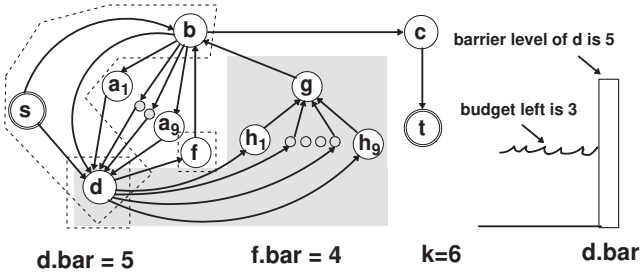


Figure 2: Key idea of the barrier-based pruning

Example. As shown in Figure 2 with $k = 6$, and s, b, and d into the stack $\mathcal{S}$, i.e., $\mathcal{S} = \{s, b, d\}$. Then it turns out that we cannot find *any* hc-s-t path in the following search space (shaded area) since we cannot reach t in 6-2=4 hops when s, b, d are blocked from current search space. If C-DFS is used, it will make the same mistake multiple times because the following vertices $\{a_1, a_2, \ldots, a_9\}$ will be visited and hence fall in the same trap (e.g., the shaded area) with fruitless search efforts. Inspired by the blocking technique in [31], we propose a barrier-based pruning technique to learn from mistakes.

In the above example, if the vertex d is unstacked without any new output after exhaustively exploring its corresponding search space with depth at most k, we know that d needs to take at least 5 hops to reach t without touching any node in $\mathcal{S} = \{s, b\}$, i.e., $sd(d, t|\mathcal{S}) \geq 5$. This is because, the vertex d already consumes 2 hops when it is pushed to $\mathcal{S}$ at the first time, and we cannot find any hop-constrained simple path from d to t, not containing $\{s, b\}$. When a_1 is pushed into stack with $\mathcal{S} = \{s, b, a_1\}$, we can safely stop exploring the vertex d because: (1) we already consume 3 hops to reach d and the budget left for the following search is only 3; and (2) there is no path p, without containing vertices s or d, from d to t with $len(p) \leq 4$. Similarly, we can also avoid the trap for the vertices $\{a_2, \ldots, a_9\}$. By doing this, the number of fruitless searches can be significantly reduced.

In this paper, we continuously maintain a *barrier level* for each vertex u regarding the *current* stack $\mathcal{S}$, denoted by $u.bar$; that is, $sd(u, t|\mathcal{S}) \geq u.bar$. In the above example, we have $d.bar = 5$ after the search branch "$s \to b \to d$" fails and d is unstacked. Similarly, we have $f.bar = 4$. Thus, given the search stack $\mathcal{S} = \{u_1, \ldots, u_t\}$ where u_t is the top element and v is a out-going neighbor of u_t with $v \notin \mathcal{S}$, we do not need to explore v if $len(\mathcal{S}) + 1 + v.bar > k$. Regarding the above example as shown in Figure 2, we have $d.bar = 5$ and $len(\mathcal{S}) = 2$, i.e., the budget left is $k-2-1 = 3$ when we arrive d, and hence we do not need to visit d since $2 + 1 + 5 > 6$.

When a new path is output in the search, we may need to decrease the barrier levels of the vertices. For instance, regarding the stack $\mathcal{S} = \{s, b\}$ after visiting d, $a_1, \ldots, a_9$, the vertex c will be explored and a new path $p(sbct)$ is reported. Thus, when the vertex b is unstacked, the barrier level of the vertex f should be updated. Now, there is a path $p(bct)$, not containing the vertex in $\mathcal{S} = \{s\}$, from b to t with $len(p) = 2$. Since there is an edge (f, b), we should set $f.bar$ to 3 after b is unstacked. With similar argument, the update will be propagated to d. In this paper, we will carefully maintain the influence of the vertices such that the barrier levels of the vertices are correctly updated.

4.2 Algorithm Description

Algorithm 2: **BC-DFS** (u, $\mathcal{S}$)

```
   Input  : u: the vertex to be pushed;
            S: the stack used for DFS search
   Output : F : the number of hops to t if a new path is
            output, otherwise F := k + 1
 1 F := k + 1;
 2 S.push(u);
 3 if u == t then
 4 |   output p(S) where P_k := P_k ∪ p(S);
 5 |   u is unstacked from S;
 6 |   return F := 0
 7 else if len(S) < k then
 8 |   for each out-going neighbor v of u where v ∉ S do
 9 |   |   if len(S) + 1 + v.bar ≤ k then
10 |   |   |   f := BC-DFS(v, S);
11 |   |   |   if f ≠ k + 1 then
12 |   |   |   |   F := min{F, f + 1};
13 if F == k + 1 then
14 |   u.bar := k − len(S) + 1 ;
15 else
16 |   UpdateBarrier(u, F);
17 u is unstacked from S ;
18 return F
```

Algorithm 3: **UpdateBarrier**(u, l)

```
   Input  : u: the vertex to be updated;
            l: the number of hops from u to the target vertex
            t, not touching any vertex in S
 1 if u.bar > l then
 2 |   u.bar := l ;
 3 |   for each in-coming neighbor v of u with v ∉ S do
 4 |   |   UpdateBarrier(v, l + 1);
```

Algorithm 2 presents the pseudo-code of the hop-constrained DFS equipped with the proposed barrier based pruning technique, namely **BC-DFS**. Initially, for every vertex u we set $u.bar = 0$. The the procedure BC-DFS is called recursively to output all hc-s-t paths, with initial call BC-DFS(s, $\mathcal{S} = \emptyset$). We use a flag F to record the minimal number of hops from u to the target vertex t, and F is set to $k + 1$ if there is no new output in the following search. For each newly pushed vertex u, we first set its flag $F := k + 1$ at Line 1. If $u == t$, a new hc-s-t path is output at Line 4. If $len(\mathcal{S}) < k$ at Line 7, we will continue the search through the neighbors of u (Lines 8-12). For each neighbor $v \notin \mathcal{S}$, we do not need to consider v if $len(\mathcal{S}) + v.bar + 1 > k$ (Line 9).

If there is no new output after exploring all of the neighbors of u (i.e., $F == k+1$), Line 14 sets $u.bar$ to $k - len(\mathcal{S}) + 1$, which indicates that $sd(u, t|\mathcal{S}) \geq u.bar$. Otherwise, it implies that u can reach the vertex t within F hops, without touching the vertices in $\mathcal{S}$ (i.e., $sd(u, t|\mathcal{S}) < u.bar$). Thus, the bar levels of the vertices influenced by u will be updated at Line 16 in a recursive way as described in Algorithm 3. Note that we use f to record the number of hops from child vertex v to t at Line 10 of Algorithm 2. Clearly, u can reach t with at most $f+1$ hops, and F is set to the minimal $f+1$ value among all neighbors at Line 12. Note that we only need to update the $v.bar$ at Line 2 in Algorithm 3 when $u.bar > l$ where l is the number of hops from u to t without touching any vertex in $\mathcal{S}$.

4.3 Analysis

The key of the analysis is the correctness of the barrier level of each vertex during the search. For a given vertex u, we say $u.bar$ is *correct* if and only if (1) $u \in \mathcal{S}$ [1]; *or* (2) given the current stack $\mathcal{S}$, if there is a path $p(u \rightsquigarrow t)$, not containing any vertex in $\mathcal{S}$, we have $len(p) \geq u.bar$, i.e., $sd(u, t|\mathcal{S}) \geq u.bar$.

In this paper, for a vertex u in the top of the stack $\mathcal{S}$, the **budget** of u is $k - len(\mathcal{S})$, which is the number of hops left for u to continue the search without violating the hop constraint. Given a path $p(u \rightsquigarrow t)$, we use $p[x]$ to denote the position of x in the path p with $p[u] = len(p)$ and $p[t] = 0$, i.e., $p[x]$ is the number of hops x can reach t along the path p. The following lemma shows the condition that a vertex u on the top of stack can reach the target vertex t in Algorithm 2.

Lemma 1. *Suppose $u.bar$ value is correct for every vertex u. Given a stack $\mathcal{S}$ with top vertex u and a path $p(u \rightsquigarrow t)$, u will reach the target vertex t in Algorithm 2 before it is unstacked if $k - len(\mathcal{S}) \geq len(p)$ and every vertex (except u) in the path is not contained by $\mathcal{S}$.*

Proof. The fact $k - len(\mathcal{S}) \geq len(p)$ implies that the vertex u has enough budget to reach t in the following search. As all vertices $\{x\}$ along the path are not contained by $\mathcal{S}$, the search can only be blocked by their barriers. Since x is not in the $\mathcal{S}$ and $x.bar$ is correct w.r.t $\mathcal{S}$, we have $x.bar \leq p[x]$ since x can reach t within $p[x]$ hops. This implies that x will not block the search by barrier-based pruning. Thus, u can reach t in Algorithm 2. □

Now we prove that the barrier levels are correctly maintained in Algorithm 2.

Theorem 1. *For any vertex u, $u.bar$ is correctly maintained in Algorithm 2.*

Proof. Recall that $u.bar$ is always correct if $u \in \mathcal{S}$ since it will not be used in the computation. When we set $u.bar$ at Lines 14 and 16, the value is correct w.r.t $\mathcal{S}$ because the search space constraint by $\mathcal{S}$ has been exhaustively explored. Now we need to show $u.bar$ is correctly maintained regarding $\mathcal{S}$ in the following search. Suppose a new vertex v is pushed to $\mathcal{S}$, the correctness of $u.bar$ is immediate because $\mathcal{S} = \mathcal{S} \cup \{v\}$ leads to a strictly smaller search space.

Now we consider the situation a vertex v is unstacked from $\mathcal{S}$. We use v to denote the first vertex which causes the incorrectness of $u.bar$. If the unstack of v does not affect $u.bar$, this implies that $u.bar$ is still correct for the new stack $\mathcal{S} \setminus \{v\}$. Otherwise, $u.bar$ may be updated in two ways:

(1) If $v == u$, the correctness of $u.bar$ still holds according to the above argument.

(2) Given $v \neq u$ and the unstack of v affects the $u.bar$, this implies that there is a path $p(u \rightsquigarrow v \rightsquigarrow t)$, not containing any vertex in $\mathcal{S}$, such that $len(p) < u.bar$; that is, u can reach t with less number of hops due to the release of v from $\mathcal{S}$. We assume that the barrier levels of the vertices are correctly maintained before v is unstacked. We need to consider two cases:

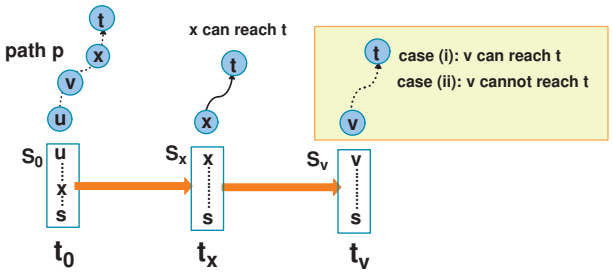


Figure 3: Proof outline for cases (i) and (ii)

Case (i): Suppose v can reach the target t through this path p in Algorithm 2 before v is unstacked, we have $v.bar \leq p[v]$. Now we show the update of v can be correctly propagated to u. Here, we only need to prove that $y.bar > p[y]$ for every vertex y along the path p according to our barrier level update strategy in Algorithm 3.

Construction of $\mathcal{S}_0$, $\mathcal{S}_x$ and $\mathcal{S}_v$. Let t_0 denote that *last* timestamp that $u.bar$ is increased with $u.bar > len(p)$. The corresponding stack is denoted by $\mathcal{S}_0$ where u is the top vertex. Note that the barrier level of a vertex can only be increased when it is unstacked. We use t_v to denote the timestamp when v will be unstacked with stack $\mathcal{S}_v$ and $u.bar > len(p)$. According to our assumption, the barrier level of every vertex is correct at every timestamp t_i with $t_i < t_v$. Let M denote the set of vertices appearing in both $\mathcal{S}_0$ and the path p, which *block* the path p at timestamp t_0, i.e., path p is not available for u because they are in the stack. By x, we denote the vertex in M which is the closest one to the bottom of $\mathcal{S}_0$ as shown at time t_0 in Figure 3. Since all vertices on p are already unstacked at time t_v, there must exist a timestamp t_x with $t_0 < t_x \leq t_v$ such that x is the *first* time unstacked after t_0, as illustrated in Figure 3. Note that, we have $t_x = t_v$ if x corresponds to v in M.

Barrier value propagation. Since x is the vertex in M which is closest to the bottom of $\mathcal{S}_0$, all vertices in the path p except x are unstacked before time t_x. Now we show the update of x can be prorogated to u along the reverse of the path $p(u \rightsquigarrow x)$. Let N denote the set of vertices from u to x along the path $p(u \rightsquigarrow x \rightsquigarrow t)$ (exclusive). If the algorithm cannot find a new output for all these vertices in N, every vertex $y \in N$ must meet the condition that $y.bar > p[y]$, since the budge of the vertex y is not less than that of u and they will be unstacked no later than t_v (x could be v, but the proof is similar). Here, we discuss the case that there is a path which updates the bar value of a vertex n in N. If the updated bar value is still larger than $p[u]$, it will not affect the result. Otherwise, according to our algorithm, all the barrier values of the vertices from u to n along the path p will be updated to $p[u]$ with $u.bar \leq p[u]$, which contradicts our initial assumption that $u.bar > p[u]$. This implies that the update of v will eventually be propagated to u along

[1]Note that we do not need to maintain $u.bar$ after u is pushed to the stack $\mathcal{S}$, and $u.bar$ is correctly calculated when u is unstacked.

the reverse direction of the path ($u \rightsquigarrow v$) according to our update propagation strategy in Algorithm 3.

Case (ii): Now we assume that v cannot reach t in Algorithm 2 but $u.bar > len(p) = p(u)$ regarding the current stack $\mathcal{S}$. That is, although $v.bar$ is correctly calculated, but there is no propagation to update $u.bar$ and $u.bar > p(u)$. Similar to case(i), we have stacks $\mathcal{S}_0$, $\mathcal{S}_x$ and $\mathcal{S}_v$. Note that we have $S_0(u) > S_0(x)$ and $|\mathcal{S}_x| = S_0(x)$ where $S_0(y)$ denotes the stack size when the vertex y is pushed to the stack S_0. Given that $u.bar = k - len(S_0) + 1$ and $u.bar > p(u)$, we have $len(S_0) + len(p) - 1 < k$. Given the fact that $len(S_0) + 1 = S_0(u) \geq |\mathcal{S}_x| + 1$ and $p[u] \geq p[x] + 1$, we have $k - len(\mathcal{S}_x) > p[x]$. According to Lemma 1, the vertex x can reach the target t. Following the similar rationale in case (i), this update should be correctly propagated to the vertex u. Since t_0 is the last timestamp that the barrier value of u is increased before time t_v and t_x $u.bar \leq len(p)$, this contradicts the assumption that $u.bar > len(p)$ at time t_v. □

Correctness. According to Theorem 1, the barrier levels of the vertices are correct. The Algorithm 2 (BC-DFS) is a hop-constrained DFS equipped with barrier-based pruning techniques. Given the correctness of the hop-constrained DFS and the barrier-based pruning, the correctness of the algorithm immediately follows. Note that BC-DFS naturally ensures that there are no duplicate results and all outputs are simple paths.

Time complexity. Following theorem shows that BC-DFS is a polynomial delay algorithm.

Theorem 2. *BC-DFS is a polynomial delay algorithm with $O(km)$ time per output. The time complexity of BC-DFS is $O(km\delta)$, where δ is the number of hc-s-t paths.*

Proof. Suppose a vertex u is unstacked twice and there is no new output in Algorithm 2. Let S_1 and S_2 denote the stack size after u is pushed into the stack at the first and the second time, respectively. We have $u.bar = k - S_1 + 2$ after u is unstacked at the first time. As there is no new output, the propagation of barrier values will not be invoked. Thus, $u.bar$ remains the same when u is pushed to stack at the second time. As u passes the barrier-based pruning in the second visit, we have $S_2 + u.bar \leq k$, and hence $S_2 < S_1$. So $u.bar$ will be increased by at least one after u is unstacked without any new output. This implies that a vertex cannot be pushed to stack more than k times unless there is a new output. An edge (u, v) will be visited when u is pushed into the stack. Together with the propagation cost, an edge will be visited at most $k+1$ times before a new output or the end of the call. Thus, BC-DFS is a polynomial delay algorithm with $O(km)$ time per output. And its time complexity is $O(km\delta)$. □

Space Complexity of Algorithm 2 is $O(m + k)$ since the stack size is always bounded by k.

4.4 Barrier Level Optimization Techniques

For a vertex u, we use $\Delta_s(u)$ and $\Delta_t(u)$ to denote $sd(s, u)$ and $sd(u, t)$ respectively. A hop-constrained BFS from source vertex s is conducted to compute $\Delta_s(u)$ and $\Delta_s(u)$ is set to $k + 1$ if u is not visited. Similarly, we compute $\Delta_t(u)$ by conducting a hop-constrained BFS on the reverse graph starting from t.

As $\Delta_t(u)$ and $\Delta_s(u)$ are pre-computed, we can ensure that $u.bar \geq \Delta_t(u)$ because we have $sd(u, v|S) \geq sd(u, v)$ for any stack $\mathcal{S}$. Meanwhile, we have $u.bar \leq k - \Delta_s(u)$ because the maximal possible budget left is $k - \Delta_s(u)$ when we arrive the vertex u, and there is no any output path through u if $u.bar > k - \Delta_s(u)$.

Time and space complexity. The following theorem indicates that barrier level optimization techniques can improve the time complexity of BC-DFS.

Theorem 3. *The time complexity of BC-DFS with barrier level optimization techniques is $O(k'm\delta)$ where $k' = k - sd(s, t)$.*

Proof. The cost of the graph reduction is $O(m)$ since two BFS are applied. It is immediate that the graph reduction will strictly narrow down the search space of BC-DFS, and the time complexity is still $O(km\delta)$ if graph reduction is directly applied to BC-DFS. With barrier level optimization techniques, we ensure that $\Delta_t(u) \leq u.bar \leq k - \Delta_s(u)$ holds for every vertex $u \notin \mathcal{S}$. Therefore, the maximal gap of $u.bar$ is bounded by $k - (\Delta_s(u) + \Delta_t(u)) \leq k - sd(s, t)$. Note that the maximal gap is k in BC-DFS. Thus, the time complexity of BC-DFS with barrier level optimization techniques is $O(k'm\delta)$ where $k' = k - sd(s, t)$. □

Same as BC-DFS, the space used by barrier level optimization techniques is $O(k + m)$.

5. JOIN ORIENTED APPROACH

In this section, we present a join-oriented approach for hc-*s*-*t* path enumeration.

5.1 Motivation

DFS based approaches enjoy advantages such as memory efficiency, duplication-free and loop-free. However, it is difficult for DFS-oriented approaches to share computation by divide and conquer computing paradigm because the paths are output one by one. As shown in Figure 4, there are 9 paths from s to c_1 through vertices a_1 to a_9, denoted by P_l. Similarly, we use P_r to denote the 9 paths from c_1 to t through vertices d_1 to d_9. In this example, any pair of paths $p_1 \in P_l$ and $p_2 \in P_r$ can come up with a simple 4-path by the concatenation (i.e., join) of p_1 and p_2. Thus, there are totally $9 * 9 = 81$ hc-*s*-*t* paths from s to t for $k = 4$, containing the vertex c_1. If the DFS is applied, we notice that the computation of these paths in P_r will be repeated 9 times (i.e., c_1 will be pushed to the stack 9 times and each following computation is exactly the same), and hence each edge in P_r will be visited 9 times. Thus, to enumerate hc-*s*-*t* paths containing c_1, the total number of edges visited in DFS is $18 + 18 * 9 = 180$. As an alternative, we can compute the paths in P_l and P_r separately, say through BC-DFS, then apply a simple join on P_l and P_r. In this way, the total number of edges visited is only $18 + 18 = 36$.

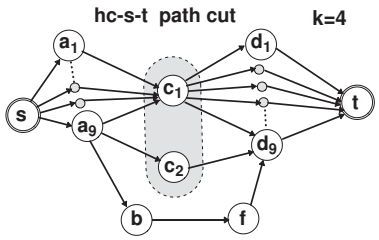


Figure 4: Key idea of join-oriented approach

Motivated by the above example, we develop join-oriented approach for efficient hc-*s*-*t* path enumeration. Intuitively,

we can find a cut for vertices s and t such that any hc-s-t path must contain at least one of the cut vertices, which is called **hc-s-t path cut**, denoted by C_k. For a cut vertex $c \in C_k$, we use $P_l(c)$ and $P_r(c)$ to denote the paths on its left side $\{p(s \rightsquigarrow c)\}$ and right side $\{p(c \rightsquigarrow t)\}$. By P_l, we denote the *left-side paths* where $P_l = \bigcup_{u \in C_k} P_l(u)$, while P_r denotes the *right-side paths* with $P_r = \bigcup_{u \in C_k} P_r(u)$. Once $P_l(c)$ and $P_r(c)$ are available for every cut vertex c in C_k, we may simply join (i.e., concatenate) paths from $P_l(c)$ and $P_r(c)$, and remove the paths with loops or $len(p) > k$, denoted by $P(c)$. Let $P = \bigcup_{c \in C_k} P(c)$, the result can be output after removing the duplication.

Given the above computing paradigm, an immediate solution is to apply a state-of-the-art s-t cut technique to find the cut vertices, and then apply the DFS technique to compute $P_l(c)$ and $P_r(c)$ for each cut vertex c. In addition to the expensive cost of traditional s-t cut algorithms (e.g., $O(mn \log n)$ in the classical s-t cut work [27]), we may come up with numerous duplicate k-paths during the computation, and lead to expensive duplicate detection cost. Moreover, we need to use $k-1$ as search depth in the DFS for the computation of $P_l(c)$ (resp. $P_r(c)$) because there may exist path with length 1 in $P_r(c)$ (resp. $P_l(c)$). This makes this solution less attractive because the BC-DFS algorithm proposed in Section 4 uses search depth k.

Our Approach. Following the above computing paradigm, we propose a new approach which can quickly find a hc-s-t path cut. The key idea is to use the "middle vertex" of a hc-s-t path to be its unique representative, which enables us to: (1) apply the BC-DFS method to compute $P_l(c)$ and $P_r(c)$ with search depth at most $\lceil \frac{k}{2} \rceil$; (2) develop efficient hc-s-t path cut computing algorithm; and (3) avoid duplicate detection because a path only has one unique middle vertex.

5.2 Join Method

In this subsection, we will introduce the algorithm to find middle vertices, followed by the join-oriented hc-s-t path enumeration algorithm.

Middle vertices cut (P_m). Given a path $p = (v_1, \ldots, v_h)$, its middle vertex is the $\lceil \frac{h}{2} \rceil$-th vertex along the path, denoted by $m(p)$. In this paper, we will find all middle vertices for hc-s-t paths, denoted by P_m. As every hc-s-t path has a unique middle vertex, P_m is naturally a hc-s-t path cut, namely *middle vertices cut*.

Find middle vertices cut. A naive way to find P_m is to report middle vertex for each available hc-s-t path, which is infeasible because our problem itself is to enumerate them. Thus, we develop an efficient algorithm for this purpose without materializing these paths, and the pseudo-code is presented in Algorithm 4. For each vertex u, we use $L_m(u)$ to record the positions that u appears among the paths starting from source vertex s; that is, $i \in L_m(u)$ implies that u is located at the i-th position of a path starting from source vertex s. Similarly, By $R_m(u)$, we indicate its possible *reverse positions* for the paths from the target vertex t (i.e., the number of hops from t in the reverse graph G^r). Particularly, we only consider the positions not larger than $\lceil \frac{k}{2} \rceil$ (resp. $\lfloor \frac{k}{2} \rfloor$) in $L_m(u)$ (resp. $R_m(u)$). For instance, in Figure 4 we have $L_m(c_1) = \{2\}, R_m(c_1) = \{2\}$, $L_m(c_2) = \{2\}$, $R_m(c_2) = \{2\}$, $L_m(a_1) = \{1\}$, $R_m(a_1) = \emptyset$, $L_m(b) = \{2\}$, $R_m(b) = \emptyset$, $L_m(d_1) = \emptyset$, $R_m(d_1) = \{1\}$.

In this paper, we say (i, j) is a *matched pair* for a vertex u if $i \in L_m(u)$, $j \in R_m(u)$, and $i == j$ (or $i == j+1$). Such a matched pair suggests that u is the middle vertex of a path with length $i + j$. A vertex u can be chosen as a middle vertex if there is such a matched pair. In the example of Figure 4, we have $P_m = \{c_1, c_2\}$ because c_1 and c_2 can find matched pairs.

As a straightforward solution to find P_m, we can compute $L_m(u)$ by applying DFS with depth $\lceil \frac{k}{2} \rceil$ from source vertex s and compute $R_m(u)$ by applying reversed DFS with depth $\lfloor \frac{k}{2} \rfloor$ from target vertex t. Then we can remove a vertex u from $L_m(u)$ and $R_m(u)$ if we cannot find a matched pair. This cost is expensive since DFS is involved.

In Algorithm 4, we present an efficient implementation to find P_m based on a synchronized bidirectional search. The key idea is to avoid explicitly enumerating the paths, and only keep the positions each middle vertex contributes. Particularly, by $P_m(i)$ we denote the middle vertices of s-t paths with length i. At Line 3, We use $\mathcal{L}_i$ to keep the vertices in the i-th expansion (i.e., go through *out-going* edges) starting from source vertex s with $\mathcal{L}_0 = \{s\}$. Similarly, Line 7 uses $\mathcal{R}_i$ to keep the vertices in the i-th expansion (i.e., go through *in-going* edges) starting from target vertex t with $\mathcal{R}_0 = \{t\}$. The hc-s-t path cut P_m can be calculated for paths with lengths $2i-1$ and $2i$ at Lines 4 and 8, respectively. The time complexity of the algorithm is $O(km)$) because (1) each edge will be visited at most twice at each iteration and the number of iterations is $\lceil \frac{k}{2} \rceil$; and (2) it takes $O(n)$ (we assume $n \leq m$) time at each iteration to identify the intersection of $\mathcal{L}_i$ and $\mathcal{R}_i$ (or $\mathcal{R}_{i-1}$) if the hashing approach is employed.

Algorithm 4: **Find Middle Vertices**(s,t, G)

```
   Input  : s, t: the source and target vertices;
            G: the graph
   Output : P_m: the middle vertices
1  L_0 = {s}; R_0 := {t};
2  for i = 1 to ⌈k/2⌉ do
3      L_i ← the out-going neighbors of the vertices in L_{i-1};
4      P_m(2i − 1) := L_i ∩ R_{i-1} ;
5      if i == ⌊k/2⌋ + 1 then
6          break;
7      R_i ← the in-going neighbors of the vertices in R_{i-1};
8      P_m(2i) := L_i ∩ R_i ;
9  return P_m
```

Join Algorithm. Now we introduce the join-oriented algorithm, namely JOIN, based on the middle vertices cut P_m with the following steps:

Step (1). Apply Algorithm 4 to compute the middle vertices cut P_m.

Step (2). Add a virtual target vertex t', and put an edge (u, t') from every middle vertex u in P_m. Apply the BC-DFS search proposed in Section 4, with search depth $\lceil \frac{k}{2} \rceil + 1$ and target vertex $t = t'$. During the BC-DFS search, when a vertex u is pushed to the stack $\mathcal{S}$ with $\mathcal{S} = \{s, \ldots, u\}$, we will put the path $p = \mathcal{S}$ to $P_l(u)$ if $u \in P_m(2 \times len(p)) \cup P_m(2 \times len(p) - 1)$; that is, u might be the middle vertex of a path $p(s \rightsquigarrow u \rightsquigarrow t)$.

Step (3). Compute $P_r(u)$ for every middle vertex $u \in P_m$ in a similar way with Step (2) with a virtual vertex s', where the search depth is $\lfloor \frac{k}{2} \rfloor + 1$.

Step (4). Enumerate each pair (p_1, p_2) where $p_1 \in P_l(u)$ and $p_2 \in P_r(u)$ for each middle vertex u. Let p denote the concatenation of p_1 and p_2, we output p if u is the middle vertex of p (i.e., $len(p_1) = len(p_2)$ or $len(p_1) = len(p_2) + 1$) and there is no repeated vertex in p.

Correctness. For any s-t simple k-path p, Algorithm 4 will include its middle point vertex u in $P_m(u)$ at Step (1); that is, we have $p = (s \rightsquigarrow u \rightsquigarrow t)$ which can be decomposed to $p_1 = (s \rightsquigarrow u)$ and $p_2 = (u \rightsquigarrow t)$ where $len(p_1) \leq \lceil \frac{k}{2} \rceil$, $len(p_2) \leq \lfloor \frac{k}{2} \rfloor$, and $len(p_1) = len(p_2)$ (or $len(p_1) = len(p_2) + 1$). Recall that we have $len(p) \leq k$ if p is a hc-s-t path. For each u, our modified BC-DFS can find all paths $\{p_l = (s, \ldots, u)\}$ with $len(p_l) \leq \lceil \frac{k}{2} \rceil$, and hence p_1 will be founded. Similarly, path p_2 will be retrieved as well. After the loop check at Step (4), p will be output if it is a simple path. Note that we will not generate any duplicate path in Steps (2) and (3) due to the use of BC-DFS. There is no duplicate path in Step (4) as well because each path will be output by its middle point only. Thus, the correctness of JOIN algorithm follows.

Time Complexity. Step (1) takes $O(km)$ time according to the above analysis. The time complexity of Steps (2) and (3) are bounded by $O(mkc_1)$ and $O(mkc_2)$, respectively, according to the analysis in Section 4, where c_1 and c_2 denote the number of left-side simple paths (P_l) and right-side simple paths (P_r), respectively. Note that each left-side path $p_l \in P_l(u)$ can match at least one right-side path from $P_r(u)$ to form a hop-constrained s-t path. The time complexity of the Step (4) is bounded by $O(\alpha)$ where α is the number of hop-constrained s-t paths. Since there is at least one matched pair for each path in P_l and P_r, we have $c_1 + c_2 \leq 2\alpha$ and hence the time complexity of join-oriented algorithm is $O(km\alpha)$.

Space Complexity. The space complexity of JOIN is dominated by c_1 and c_2, where c_1 and c_2 denote the number of left-side paths in P_l and right-side paths in P_r. As shown in the time complexity analysis, we have $c_1 + c_2 \leq 2\alpha$. Thus, the space is bounded by $O(\alpha)$ where α is the number of hop-constrained s-t paths. In practice, $c_1 + c_2$ is much smaller than 2α because a middle vertex u can cover multiple left-side and right-side paths.

5.3 More Constraints

As shown in Section 1, in addition to hop constraint k, one may need to consider other constraints based on the attributes of the vertices and edges along the s-t path. Since our proposed two approaches are online searching algorithms, it is immediate that these attribute constraints (e.g., label and value constraints) can be naturally integrated into current algorithms by pruning the vertices and edges violating the constraints. We can expect better performance of two proposed algorithms after imposing other attribute constraints due to the reduced search space.

6. EVALUATION

In this section, we evaluate the efficiency of proposed techniques on comprehensive experiments.

6.1 Experimental Setting

Algorithms. In this section, we compare proposed algorithms with state-of-the-art solutions for hc-s-t path enumeration. We also evaluate proposed algorithms with the-state-of-the-art algorithm for dynamic hop-constrained cycle detection problem. Below are algorithms evaluated in the experiments.

- **C-DFS**. The hop-constrained DFS algorithm presented in Section 3.2.
- **T-DFS**. The polynomial delay DFS algorithm [54] introduced in Section 3.2.
- **T-DFS2**. The polynomial delay DFS algorithm [24] introduced in Section 3.2.
- **HPI**. The HP-Index based algorithm [52] introduced in Section 3.2.
- **BC-DFS**. The C-DFS algorithm equipped with barrier-based pruning technique and barrier level optimization techniques (Section 4.2).
- **JOIN**. The join-oriented algorithm presented in Section 5.2.

Table 2: Statistics of Datasets

Dataset	$\|V\|$	$\|E\|$	d_{avg}	D	D_{90}
Reactome	6.3K	147K	46.64	24	5.39
econ-psmigr3	3K	540K	343	-	-
bio-mouse-gene	43K	14M	670	-	-
soc-Epinions1	75K	508K	13.42	14	5
Slashdot0902	82K	948K	23.08	12	4.7
Amazon	334K	925K	6.76	44	15
twitter-social	465K	834K	3.86	8	4.96
Baidu	425K	3M	15.8	32	8.54
BerkStan	685K	7M	22.18	208	9.79
web-google	875K	5M	11.6	24	7.95
Skitter	1.6M	11M	13.08	31	5.85
WikiTalk	2M	5M	4.2	9	4
LiveJournal	4M	68M	28.4	16	6.5
DBpedia	18M	172M	18.85	12	4.98
Twitter(WWW)	42M	1.46B	70.51	23	3.97

Datasets. We deploy 10 real-life graphs to evaluate the efficiency of the algorithms. Table 2 shows important statistics of these graphs. D_{90} means the 90-percentile effective diameter and D denotes the diameter.

Settings. We randomly generate $1,000$ random query pairs $\{s,t\}$ on the graph where source vertex s could reach target vertex t in k hops. In the experiments, all programs are implemented in standard C++ and compiled with g++ 4.8.5. The source code of HPI in C++ is obtained from the authors in [52]. We tune the number of hot-points chosen for HPI on each individual graph to achieve a good performance, and no parameter tuning is required for other algorithms. All experiments are performed on a machine with 20 Intel Xeon 2.3GHz and 768 GB main memory running Linux (Red Hat Linux 7.3, 64 bit).

6.2 Datasets Details

All the datasets are downloaded from two public websites: Konect[2] [39] , NetworkRepository[3] [56] and SNAP[4] [44]. The following are detailed data descriptions obtained from the above two websites.

6.3 Efficiency of hc-s-t Path Enumeration

We randomly generate $1,000$ random query pairs $\{s,t\}$ where source vertex s could reach target vertex t in k hops. In this subsection, we report the average query response time of the algorithms to evaluate their time efficiency.

Efficiency on different datasets. First we compare the average runtime of 6 algorithms (C-DFS, T-DFS, T-DFS2, HPI, BC-DFS and JOIN) on all 15 graphs. We set $k = 8$ for *Amazon* and *twitter-social* to achieve similar performance

[2] http://konect.uni-koblenz.de/networks/
[3] http://networkrepository.com/networks.php
[4] http://snap.stanford.edu/data/

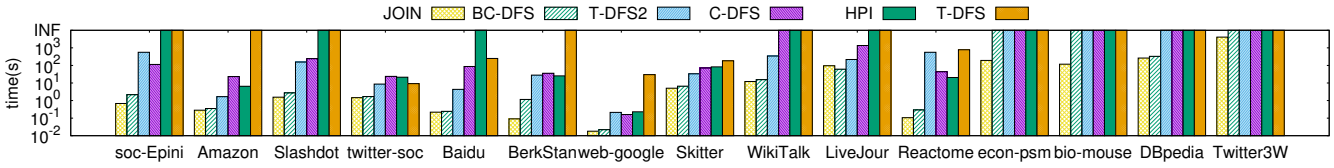
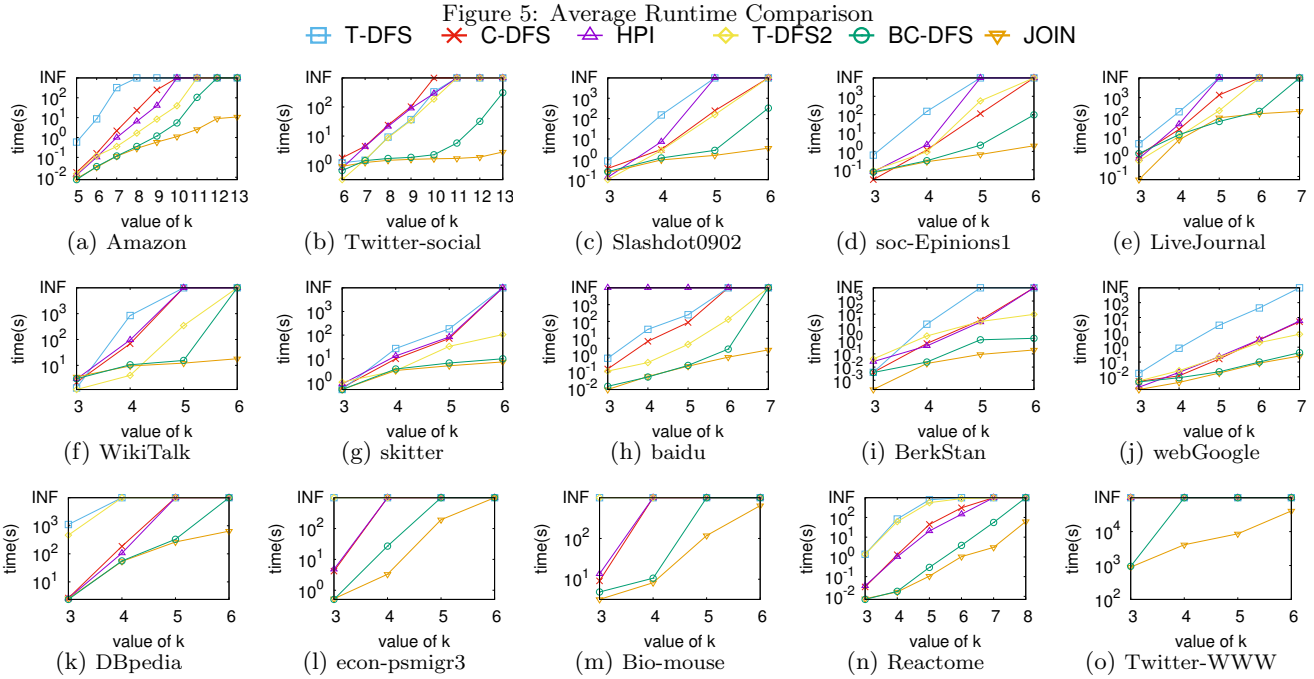


Figure 5: Average Runtime Comparison



(a) Amazon (b) Twitter-social (c) Slashdot0902 (d) soc-Epinions1 (e) LiveJournal

(f) WikiTalk (g) skitter (h) baidu (i) BerkStan (j) webGoogle

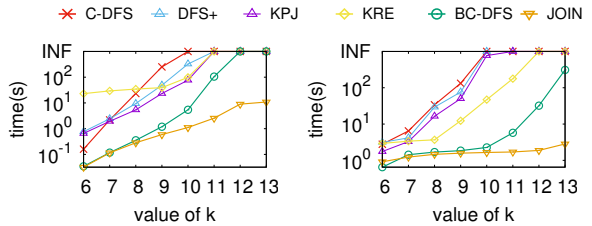
(k) DBpedia (l) econ-psmigr3 (m) Bio-mouse (n) Reactome (o) Twitter-WWW

Figure 6: Tuning k

with other graphs while $k = 5$ for remaining graphs. We set the runtime of an algorithm to Inf if it cannot finish in 10,000 seconds. It is reported in Figure 5 that JOIN has the best overall performance in the experiments and BC-DFS ranks second. Note that JOIN can be regarded as an advanced version of BC-DFS for better path computation sharing by using the join strategy, with some overhead costs such as finding cut vertices and duplicate detection in the join process. We notice that these overhead costs are well paid-off on most of the graphs. One exception is the Liverjournal graph, in which the overhead costs dominate the benefits brought by the join strategy and hence BC-DFS outperforms JOIN. Not surprisingly, T-DFS demonstrates the worst performance on most of the graphs although it is a polynomial delay algorithm with nice theoretical guarantee. This is because of the expensive shortest path distance computation cost cannot be paid-off. It is shown that T-DFS2 outperforms T-DFS because T-DFS2 reduces the shortest path distance computation. Nevertheless, T-DFS2 has similar overall performance with C-DFS, which is not attractive in practice. For instance, T-DFS2 is outperformed by C-DFS on *soc-Epinions* and *web-google* and they have similar performance on *Slashdot0902*, *Berkstan* and *Skitter*. We notice that HPI only outperforms C-DFS on Amazon and twitter-social datasets, which have much less number of result paths compared to other datasets (see Figures 10 and 11 in Section 6.7). This is because the performance of HPI is quite sensitive to the number of hop-constrained *s-t* paths. Note that we even cannot build a HP-Index with around 100 hot points within one day on most of the datasets, except Amazon and twitter-social. Although BC-DFS, T-DFS and T-DFS2 are all $O(km\delta)$, T-DFS and T-DFS2 show much worse performance than BC-DFS. The reason is that both T-DFS and T-DFS2 use a positive check strategy which leads to heavy overheads, while BC-DFS adaptively learns from failure and in most cases it will terminate earlier than T-DFS and T-DFS2.

Effect of hop-constraint k. In Figure 6, we evaluate the running time of the algorithms on 15 graphs by varying the hop-constraint k. Similar results are observed, compared with that of Figure 5. Particularly, it is shown that JOIN algorithm demonstrates the best scalability regarding the growth of the size constraint k among all evaluated algorithms. As reported in Section 6.7 (Figure 11), the number of s-t simple paths grows exponentially w.r.t k. The overhead costs of JOIN algorithm such as finding the middle vertex cut and duplicate detection in join process are better paid-off with larger k values. Thus, although JOIN may rank after BC-DFS when k is small because of the extra overhead costs involved, it eventually overtakes BC-DFS when k grows. As reported in Figure 6 (o), JOIN can still answer hc-s-t queries on billion-scale graph tweet-WWW within 20786 seconds on average for k= 6 and BC-DFS can support k=3 with around 947 seconds on average while the average response time all other algorithms exceed 100,000 seconds for $k \geq 3$. We also notice that the margin increases rapidly with k on most of the graphs except for skitter (Figure 6(g)) and webGoogle (Figure 6(j)). As shown in Figure 11(b), the number of paths in skitter and webGoogle grows much slower compared to other graphs.

6.4 Comparison With Top-k' Shortest Path Algorithm



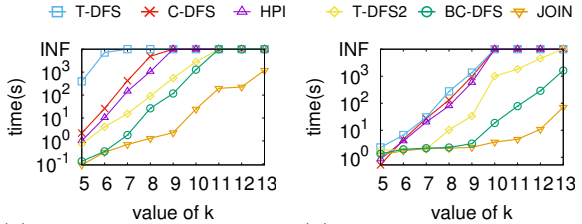
(a) Amazon (b) Twitter-social

Figure 7: Average Runtime

To investigate if we can simply apply the existing top k' shortest path algorithm to the problem of hc-s-t path enumeration, we consider two latest algorithms, namely KRE [21] and KPJ [6] respectively. The source codes are kindly provided by the authors. We also evaluate a variant of DFS, namely DFS+, by combining the C-DFS with the reverse shortest path tree technique used in [6]; that is, we can find a set of candidate vertices with lower bound distance to the target vertex t, and some vertices can be pruned based on current search depth and the lower bound distance. In Figure 7, we report the performance of KRE, KPJ, DFS+, C-DFS and our proposed two approaches BC-DFS and JOIN on two graphs Azamon and Twitter-social, with the growth of the hop constraint k. It is shown that top k' shortest path algorithms have been significantly outperformed by BC-DFS and JOIN, and the margin quickly grows with k. The key reason is that, although KRE, KPJ and DFS+ can also take advantage of the distance lower bound to t for pruning purpose with some overhead, the reversed shortest path tree calculated is static while our BC-DFS uses a dynamic one which is tighter because we carefully consider the currently visited vertices during the search. The other key reason is that our BC-DFS can learn from failure while KRC, KPJ, DFS+ cannot. The JOIN algorithm can further improve the performance for large k by computing sharing.

6.5 Efficiency of Dynamic Cycle Detection

HPI [52] is the state-of-the-art algorithm for hop-constrained simple cycle detection on dynamic graph. Since there is no index structure involved, C-DFS, T-DFS, BC-DFS, T-DFS2 and JOIN algorithms can be immediately deployed for the hop-constrained cycle detection on dynamic graph. That is, for each incoming edge (t, s), we invoke a hc-s-t path enumeration query with $k = k' - 1$ where k' is the hop constraint of the cycle. Same as [52], we use 99.9% latency to measure performance of these algorithms. Six algorithms are evaluated on *Amazon* and *twitter-social* with the growth of k value. Similar to the claim in [52], Figure 8 shows that HPI outperforms C-DFS because it can avoid some pitfalls caused by hot points. Nevertheless, its performance is not competitive compared with BC-DFS and JOIN.

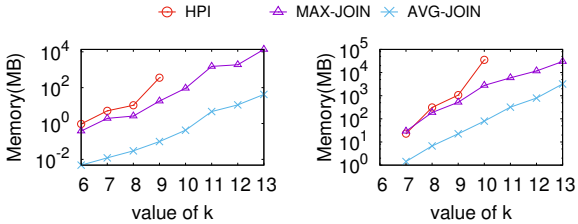


(a) Amazon 99.9% Runtime (b) Twitter 99.9% Runtime

Figure 8: Evaluate Dynamic Cycle Detection

6.6 Main Memory Usage

In this section, we evaluate the main memory usage of two algorithms HPI and JOIN. Note that C-DFS, T-DFS, T-DFS2 and BC-DFS are DFS-oriented algorithms, which are naturally space efficient. To better show the tendency of the two algorithms, we exclude memory usage of the graph storage. Figure 9 shows that memory usage of JOIN and HPI on *Amazon* and *Twitter*. MAX-JOIN and AVG-JOIN shows the maximum and the average main memory usage of JOIN over 1,000 queries. Note that the dominance main memory consumption of HPI is the HP-Index, and its size is reported as the memory usage. It is shown that JOIN always use less main memory compared with that of HPI.

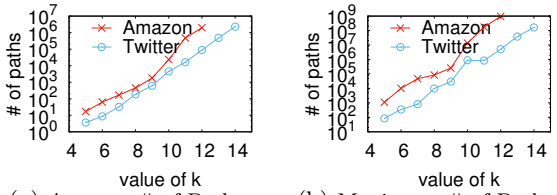


(a) Amazon (b) Twitter-Social

Figure 9: Average Memory Usage

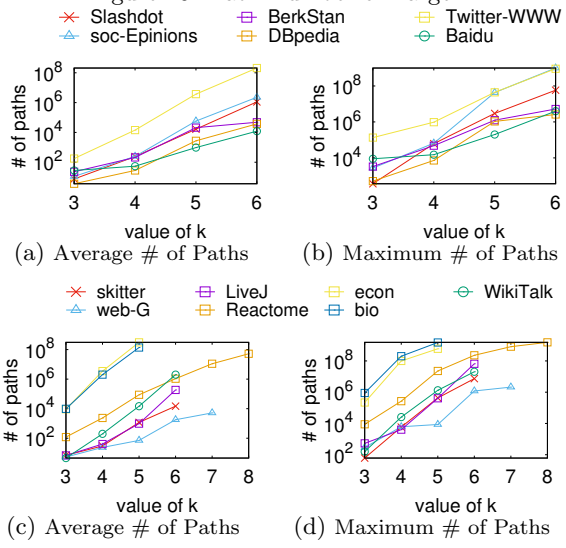
6.7 Number of hc-s-t Paths

In this subsection, we report the average and maximum number of hc-s-t paths on all datasets with different hop-constraint (k) values in Figures 10 and 11. As expected, the average and maximal number of hc-s-t paths grows exponentially with k.



(a) Average # of Paths (b) Maximum # of Paths

Figure 10: Path Number of Large k



(a) Average # of Paths (b) Maximum # of Paths

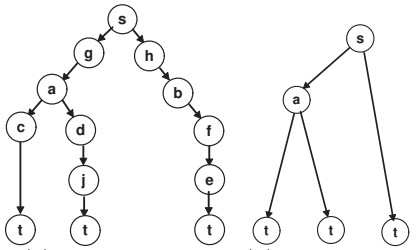
(c) Average # of Paths (d) Maximum # of Paths

Figure 11: Path Number of Small K

7. ANALYSIS OF TWO EXISTING POLYNOMIAL DELAY ALGORITHMS

In this section, we show that both T-DFS and T-DFS2 take $O(km)$ time per output on *directed graph*.

In [54], Rizzi et al. show that they can achieve $O(m)$ time per output on *undirected graph* by carefully building the recursion tree where each leaf-node represent an unique output and there are at least two branches for every internal node. As shown in Figure 12(a), if there are some single-child internal nodes which corresponds to only one leaf node (output), the delay time is $O(km)$ since each inter-node incurs $O(m)$ cost for a reverse BFS. In the *undirected graph*, they can identify and "merge" all the consecutive single-child nodes in the recursion tree (e.g., Figure 12(b)). This guarantees that the remaining inter-nodes have at least two child nodes (i.e., corresponds to two valid paths), resulting in $O(m)$ time delay for *undirected* graph. Nevertheless, authors stress that it is difficult to apply the above technique to *directed graph*, thus T-DFS takes $O(km)$ time per output on *directed graph* as shown in Section 3.2.

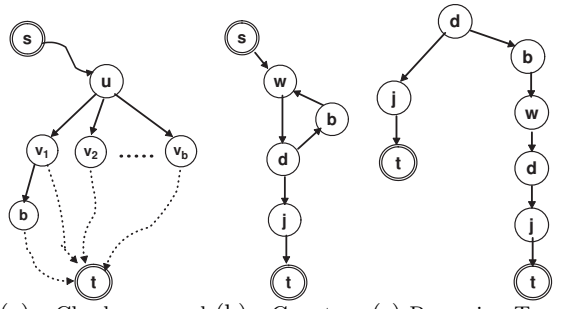


(a) Recursion Tree (b) Merged Recursion Tree

Figure 12: Example of merging recursion tree

Recently, Grossi et al. study the problem of listing k disjoint s-t paths in [24]. As the enumeration of hc-s-t paths is a special case of k disjoint s-t path listing problem, they propose a polynomial delay technique following the aggressive checking strategy. The key idea is to merge the recursion tree on the *directed graph* as follows. Let u be the currently processed vertex (i.e., the top vertex of the stack) of the DFS as shown in Figure 13(a), and the path $p(s \rightsquigarrow u)$ records the vertices visited (i.e., vertices in the stack). We say a neighbor v of u is a *good neighbor*, if v can reach t in $k - len(p) - 1$ hops without touching any vertex on p, i.e., $sd(v, t|p) \leq k - len(p) - 1$. Same as BC-DFS, we can conduct a reverse BFS with $O(m)$ time to find all vertices $T = \{x\}$ with $sd(x, t|p) \leq k - len(p) - 1$. Then we can say the neighbor vertex v is a good neighbor of u if $v \in T$. If there is no good neighbor or there are at least two good neighbors, T-DFS2 is the same as T-DFS; that is, we will terminate the search if there is no good neighbor, or all good neighbors will be explored (i.e., regard u as an internal node in the recursion tree) if there are at least two good neighbors. The key difference is that T-DFS2 will skip the vertex (i.e., merge the node in the recursion tree) if there is only one good neighbor. For instance, we will skip the vertex v_1 and continue to process the vertex b if v_1 is the only good neighbor of u. This procedure will continue if b has only good neighbor as well, i.e., merging consecutive single-child nodes in the recursion tree. Note that there is no shortest path distance computation involved if a vertex is skipped.

If we can ensure that every internal node has at least two good neighbors, T-DFS2 algorithm will have $O(m)$ time delay per hc-s-t path, as claimed in [24]. Unfortunately, we show that there may exist *false* good neighbor in T-DFS2; that is, a good neighbor identified in [24] may not be able to generate a valid output. A counter-example is constructed in Figure 13(b) with 6 vertices and $k = 8$, and there is only one hop-constraint s-t *simple* path $p = (s, w, d, j, t)$. Starting from source vertex s, we know that every other vertex can reach target vertex t within 7 hops without touching s by one reverse BFS from t. Then the vertex w will be skipped as it has only one good neighbor. When d is processed, as shown in Figure 13(c), we need to regarded it as an internal node since it has two good neighbors b and j with shortest distance 4 and 1 to t, respectively. Therefore, two more shortest path computations will be invoked when b and j are processed. However, there is only one hc-s-t path in this example. This is because the branch from b leads to a path $p(s, w, d, b, w, j, t)$ with a loop. We can easily ensure the correctness of T-DFS2 by conducting a loop check, but the claim of $O(m)$ time per output does not hold. The key reason is that the shortest path distance from b to t not containing visited vertices has been updated from 4 to ∞, but we cannot update this information with time $O(1)$ when we skip the single-child nodes on the *directed graph*. Consequently, the time complexity of T-DFS2 is the same as T-DFS, i.e., $O(km)$ time per output on *directed graphs*.



(a) Check good neighbors (b) Counter-example (c) Recursion Tree

Figure 13: T-DFS2 and counter-example

8. CONCLUSION

In this paper, we investigated the problem of hop-constrained s-t simple path (hc-s-t path) enumeration, which is fundamental in the graph analytics. Although there were some existing solutions, they either do not have attractive theoretical time complexity result or demonstrate poor time efficiency on real-life graphs. By designing novel barrier-based techniques and join framework, we developed two efficient algorithms, namely BC-DFS and JOIN. On theoretical side, we showed that BC-DFS algorithm is a polynomial delay algorithm with $O(km)$ time per output, which is the same as the current state-of-the-art theoretical studies. On the practical side, extensive empirical study on 10 real-life graphs showed that BC-DFS significantly outperformed the state-of-the-art techniques. By applying the join computing paradigm, our proposed JOIN algorithm can further significantly enhance the query response time.

9. ACKNOWLEDGMENTS

Ying Zhang is supported by ARC DP180103096 and FT170100128. Lu Qin is supported by ARC DP160101513. Wenjie Zhang is supported by ARC DP180103096. Xuemin Lin is supported by 2018YFB1003504, NSFC61232006, ARC DP180103096 and DP170101628.

10. REFERENCES

[1] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.

[2] A. Bernstein and S. Chechik. Incremental topological sort and cycle detection in expected total time. In *SODA*, pages 21–34, 2018.

[3] E. Birmelé, R. A. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *SODA*, pages 1884–1896, 2013.

[4] K. Böhmová, L. Häfliger, M. Mihalák, T. Pröger, G. Sacomoto, and M.-F. Sagot. Computing and listing st-paths in public transportation networks. *Theory of Computing Systems*, 62(3):600–621, 2018.

[5] G. G. Cash. The number of n-cycles in a graph. *Applied Mathematics and Computation*, 184(2):1080–1083, 2007.

[6] L. Chang, X. Lin, L. Qin, J. X. Yu, and J. Pei. Efficiently computing top-k shortest path join. In *EDBT 2015-18th International Conference on Extending Database Technology, Proceedings*, 2015.

[7] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang. Exploring communities in large profiled graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[8] G. L. D. and K. N. P. Identifying certain types of parts of a graph and computing their number. *Ukrainian Mathematical Journal*, 24(3):313–321, 1972.

[9] E. de Queirós Vieira Martins and M. M. B. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR*, 1(2):121–133, 2003.

[10] D. Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

[11] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu. Effective and efficient attributed community search. *The VLDB JournalThe International Journal on Very Large Data Bases*, 26(6):803–828, 2017.

[12] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment*, 10(6):709–720, 2017.

[13] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment*, 9(12):1233–1244, 2016.

[14] Y. Fang, R. Cheng, S. Luo, J. Hu, and K. Huang. C-explorer: browsing communities in large graphs. *Proceedings of the VLDB Endowment*, 10(12):1885–1888, 2017.

[15] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, Jul 2019.

[16] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen. On spatial-aware community search. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):783–798, 2018.

[17] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[18] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin. Efficient algorithms for densest subgraph discovery. *Proc. VLDB Endow.*, 12(11):1719–1732, July 2019.

[19] A. Freitas, J. C. P. da Silva, E. Curry, and P. Buitelaar. A distributional semantics approach for selective reasoning on commonsense graph knowledge bases. In *International Conference on Applications of Natural Language to Data Bases/Information Systems*, pages 21–32. Springer, 2014.

[20] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *Proceedings of the VLDB Endowment*, 5(4):358–369, 2011.

[21] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 509–518. ACM, 2010.

[22] P. Giscard, N. Kriege, and R. C. Wilson. A general purpose algorithm for counting simple cycles and simple paths of any length. *CoRR*, abs/1612.05531, 2016.

[23] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, 109(7):352–355, 2009.

[24] R. Grossi, A. Marino, and L. Versari. Efficient algorithms for listing k disjoint st-paths in graphs. In *Latin American Symposium on Theoretical Informatics*, pages 544–557. Springer, 2018.

[25] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

[26] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, 3(4):45, 2007.

[27] D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.

[28] J. Hu, R. Cheng, K. C.-C. Chang, A. Sankar, Y. Fang, and B. Y. Lam. Discovering maximal motif cliques in large heterogeneous information networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 746–757. IEEE, 2019.

[29] S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pages 33–65. Springer, 2005.

[30] R. Jin and N. Ruan. Shortest path computation in large networks, Dec. 19 2013. US Patent App. 13/899,124.

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

[32] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[33] C. V. Karsten, D. Pisinger, S. Ropke, and B. D. Brouer. The time constrained multi-commodity network flow problem and its application to liner shipping network design. *Transportation Research Part E: Logistics and Transportation Review*, 76:122–138, 2015.

[34] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.

[35] A. A. Khan and H. Singh. Petri net approach to enumerate all simple paths in a graph. *Electronics Letters*, 16(8):291–292, 1980.

[36] D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*. Addison-Wesley Professional, 2011.

[37] S. R. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In *STOC*, pages 398–406, 1988.

[38] R. Kumar and T. Calders. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *PVLDB*, 11(11):1441–1453, 2018.

[39] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[40] K. L, J. Nadeau, G. Ozsoyoglu, Z. Ozsoyoglu, G. Schaeffer, M. Tasan, and W. Xu. Pathways database system: An integrated system for biological pathways. *Bioinformatics*, 19:930–7, 06 2003.

[41] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, et al. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112, 2019.

[42] N. Lao and W. W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81(1):53–67, 2010.

[43] U. Leser. A query language for biological networks. *Bioinformatics*, 21:ii33–9, 10 2005.

[44] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[45] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou.

Efficient (a, β)-core computation: an index-based approach. In *The World Wide Web Conference*, pages 1130–1141. ACM, 2019.

[46] B. Liu, F. Zhang, C. Zhang, W. Zhang, and X. Lin. Corecube: Core decomposition in multilayer graphs. In *International Conference on Web Information Systems Engineering*, pages 694–710. Springer, 2019.

[47] H. Liu, C. Jin, B. Yang, and A. Zhou. Finding top-k shortest paths with diversity. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):488–502, 2017.

[48] E. Q. Martins and M. M. Pascoal. A new implementation of yens ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):121–133, 2003.

[49] S. Mazumder and B. Liu. Context-aware path ranking for knowledge base completion. In *IJCAI*, pages 1195–1201, 2017.

[50] M. Nishino, N. Yasuda, S. Minato, and M. Nagata. Compiling graph substructures into sentential decision diagrams. In *AAAI*, pages 1213–1221, 2017.

[51] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin. Efficient probabilistic k-core computation on uncertain graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1192–1203. IEEE, 2018.

[52] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *PVLDB*, 11(12):1876–1888, 2018.

[53] J. C. Rivera, H. M. Afsar, and C. Prins. Mathematical formulations and exact algorithm for the multitrip cumulative capacitated single-vehicle routing problem. *European Journal of Operational Research*, 249(1):93–104, 2016.

[54] R. Rizzi, G. Sacomoto, and M. Sagot. Efficiently listing bounded length st-paths. In *IWOCA*, pages 318–329, 2014.

[55] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 249–260. Springer, 2005.

[56] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[57] J.-J. Salazar-González. Approaches to solve the fleet-assignment, aircraft-routing, crew-pairing and crew-rostering problems of a regional carrier. *Omega*, 43:71–82, 2014.

[58] N. Shi, S. Zhou, F. Wang, Y. Tao, and L. Liu. The multi-criteria constrained shortest path problem. *Transportation Research Part E: Logistics and Transportation Review*, 101:13–29, 2017.

[59] O. Shmueli. Dynamic cycle detection. *Inf. Process. Lett.*, 17(4):185–188, 1983.

[60] C. Sommer. Shortest-path queries in static networks. *ACM Comput. Surv.*, 46(4):45:1–45:31, 2014.

[61] L. Talarico, K. Sörensen, and J. Springael. The k-dissimilar vehicle routing problem. *European Journal of Operational Research*, 244(1):129–140, 2015.

[62] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 421–432. ACM, 2011.

[63] C. Voss, M. Moll, and L. E. Kavraki. A heuristic approach to finding diverse short paths. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4173–4179. IEEE, 2015.

[64] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin. Efficient computing of radius-bounded k-cores. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 233–244. IEEE, 2018.

[65] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment*, 12(10):1139–1152, 2019.

[66] L. Wang, L. Yang, and Z. Gao. The constrained shortest path problem with stochastic correlated link travel times. *European Journal of Operational Research*, 255(1):43–57, 2016.

[67] S. Wang, X. Xiao, Y. Yang, and W. Lin. Effective indexing for approximate constrained shortest path queries on large road networks. *Proceedings of the VLDB Endowment*, 10(2):61–72, 2016.

[68] N. Yasuda, T. Sugaya, and S. Minato. Fast compilation of s-t paths on a graph for counting and enumeration. In *Proceedings of the 3rd Workshop on Advanced Methodologies for Bayesian Networks, AMBN*, pages 129–140, 2017.

[69] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[70] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. 2010.

[71] D. Yue, X. Wu, Y. Wang, Y. Li, and C. Chu. A review of data mining-based financial fraud detection research. In *International Conference on Wireless Communications, Networking and Mobile Computing*, pages 5519 – 5522, 10 2007.

[72] A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient single-source shortest path and distance queries on large graphs. In *ACM SIGKDD*, pages 998–1006, 2013.