TDB: Breaking All Hop-Constrained Cycles in Billion-Scale Directed Graphs

You Peng[†], Xuemin Lin^{‡*}, Michael Yu[†], Wenjie Zhang[†], Lu Qin[§]

[†]The University Of New South Wales,

[‡]Antai College of Economics & Management, Shanghai Jiao Tong University, Shanghai, China,

[§]QCIS, University of Technology, Sydney

{unswpy, xuemin.lin}@gmail.com, {mryu,zhangw}@cse.unsw.edu.au, lu.qin@uts.edu.au

Abstract—The Feedback vertex set with the minimum size is one of Karp's 21 NP-complete problems targeted at breaking all the cycles in a graph. This problem is applicable to a broad variety of domains, including E-commerce networks, database systems, and program analysis. In reality, users are frequently most concerned with the hop-constrained cycles (i.e., cycles with a limited number of hops). For instance, in the E-commerce networks, the fraud detection team would discard cycles with a high number of hops since they are less relevant and grow exponentially in size. Thus, it is quite reasonable to investigate the feedback vertex set problem in the context of hop-constrained cycles, namely hop-constrained cycle cover problem. It is concerned with determining a set of vertices that covers all hop-constrained cycles in a given directed graph. A common method to solve this is to use a *bottom-up* algorithm, where it iteratively selects cover vertices into the result set. Based on this paradigm, the existing works mainly focus on the vertices orders and several heuristic strategies. In this paper, a totally opposite cover process topdown is proposed and bounds are presented on it. Surprisingly, both theoretical time complexity and practical performance are improved. On the theoretical side, this work is the first to achieve $O(k \cdot n \cdot m)$ time complexity, whereas the state-of-the-art method achieves time complexity of $O(n^k)$.¹ On the practical level, the proposed algorithm, namely TDB++, outperforms the state-of-the-art by 2 to 3 orders of magnitude on average while preserving the minimal property. As a result, the method in this paper outperforms the state-of-the-art approaches in terms of both running time and theoretical time complexity. This is the first time, to our best knowledge, that the hop-constrained cycle cover problem on billion-scale networks has been solved with a minimal² cover set for k > 3.

I. INTRODUCTION

The Feedback vertex set with the minimum size was one of the 21 NP-complete problems Karp [1] developed in an effort to break every cycle in a graph. It was created way back in the early 1960s (see the survey of Festa *et al.* [2]). A wide variety of applications, such as operating systems [3], database systems [4], and circuit testing [5], make use of it. Numerous studies have been conducted over many years, including those on approximation algorithms [6], [7], [8], [9], linear programming [10], parameterized complexity [11], [12], [13], etc.

Nevertheless, it is observed that individuals are primarily concerned about hop-constrained cycles in a large number of real-world applications. For instance, a team from Alibaba group recently investigated the *hop-constrained cycles* [14] in the context of financial fraud detection on e-commerce networks. As mentioned in [14], the fraud detection team disregards the cycles with a high number of hops due to their lack of relevance and exponential growth in search space. This leads us to investigate the feedback vertex set problem in terms of hop-constrained cycles, namely *hop-constrained cycle cover*. Specifically, given a directed graph G, we need a set of vertices containing all hop-constrained cycles whose length is constrained by a parameter on the graph; for each constrained cycle in the graph, at least one of its vertices must be in the feedback vertex.

The hop-constrained cycles are more practical than cycles in real applications, since the real applications are inherently constrained. Users are interested in cycles in a variety of graph analytic tasks. The cycle is a vital graph pattern that is usually associated with certain behaviors in many real-life applications, such as financial fraud detection, program analysis, and compiler optimization. Significant research efforts have been devoted to the cycle-related studies such as cycle enumeration (e.g., [15], [16], [17], [18]) and real-time cycle detection (e.g., [14]).

The problem of *hop-constrained cycle cover* can be regarded as an important extension of the well-known feedback vertex set problem [6] with specific concerns about constrained simple cycles (i.e., cycles with only the first and last vertices repeated).

Applications. The following are some compelling applications of the hop-constrained cycle cover problem.

(1) <u>Combinatorial Circuit Design</u>. As shown in [6], one typical use of the feedback vertex set is the combinatorial circuit design. The circuits are depicted by graphs in which each cycle denotes a possible "racing condition". Certain circuit components may receive new inputs prior to stabilization. One method to avoid such a condition is to include a clocked register at each cycle in the circuit. Due to the fact that the "racing condition" is negligible for a long cycle [19], [6], the hop-constraint is imposed automatically in this application. As a consequence, this application enforces hop-constraint by default.

(2) <u>E-commerce Networks</u>. Each node represents an account in an E-commerce Network, and each directed edge represents a money transfer between two accounts. Figure 1 depicts an

Xuemin Lin is the corresponding author.

 $^{{}^{1}}m$ and n denote the number of edges and vertices, respectively. The k denotes the hop constraint.

²"Minimal" indicates local optimal in this paper.



Fig. 1: An example of the e-commerce network, where vertices represent accounts, and edges represent transactions.

example of this kind. According to Alibaba Group specialists in [14], a hop-constrained cycle is a strong indicator of fraudulent activity or financial crime such as money laundering [20]. Figure 1 depicts three simple cycles, i.e., potential money laundering behaviors. By using a minimal hop-constrained cycle vertex cover, we can identify a group of critical individuals who are more likely to participate in fraudulent activities. For instance, hop-constrained cycle cover $\{a\}$ with the constraint $hop \leq 5$ is the most suspicious individual since it covers (i.e., becomes involved in) all three simple cycles with a length limitation of 5.

(3) <u>Program Analysis</u>. The hop-constrained cycle cover could also be applied to identify and resolve deadlock potentials in program analysis, especially for concurrent applications [21]. Deadlock is a frequent concurrency error occurs when a set of threads are blocked, and a constrained cycle in a lock graph signals the possibility of a deadlock [22]. As a consequence, building a minimal hop-constrained cycle cover is crucial in this field.

Constraints. Various constraints might be placed on the cycle computation in real-world applications. We concentrate on two typical constraints in this paper, which follows the settings in [14]: the simple and the hop constraint. Notably, these two constraints are beneficial in reality since they might result in considerably smaller results and fewer relevant cycles (e.g., a cycle with a high number of hops implies a weak connection between the vertices).

Self-loops and bidirectional edges are not considered as cycles in this paper since they are uninteresting and substantially increase the result size. Nota bene, the self-loop and bidirectional edge may be promptly verified if required.

Contributions. The main contributions are listed as follows:

- *Scalability*. To the best of our knowledge, this is the first work to answer the problems of hop-constrained cycle cover on billion-scale directed graphs, both of which have a wide range of real-life applications. To achieve this, we use an opposite cover process from existing works. In addition, delicate upper and lower bounds are proposed to achieve better theoretical time complexity and practical performance.
- Theoretical Analysis. For a given directed, unweighted graph G, we demonstrate that approximating the hop-constrained cycle cover problem with lengths ranging from 3 to k within $(k 1 \epsilon)$ is UGC-hard (Unique Games Conjecture).
- *Comprehensive Experiments*. Compared to the baselines, our comprehensive experiments demonstrate the efficiency and effectiveness of our proposed method.

Organization of the paper. The remainder of this paper is structured as follows. Section II is devoted to related work. In Section III, we introduce the preliminary. Section IV provides an examination of these problems from a theoretical standpoint. Bottom-up and Top-Down algorithms are conducted in Sections V and VI, respectively. Extensive experiments are conducted in Section VII. Finally, Section VIII concludes this paper.

II. RELATED WORK

We review closely related works in this section.

A. K-Cycle Traversal and K-Cycle-Free Subgraph

The k-cycle transversal problem (K-cycle problem for short) is to find a minimum-size set of edges that intersects all simple cycles of length k in a network. K-cycle problems are crucial in the disciplines of extremal combinatorics, combinatorial optimization, and approximation algorithms, as shown in [23], [24], [25], [26]. [26] investigates the 3-Cycle Traversal. [25] addresses 3-Cycle Traversal, 3-Cycle-Free Subgraphs, and their connections to related issues. In [27], the problem of locating a maximum subgraph devoid of cycles of length $\leq k$ is examined in the context of computational biology, and different heuristics are presented without analyzing their approximation ratio.

The most closely related problem is [28], which tackles both the problem of discovering a minimal edge subset of Ethat intersects every hop-constrained cycle, and the problem of discovering a maximum edge subset of E that does not intersect any hop-constrained cycles. They provide a (k - 1)approximation for this problem, when k is odd. [29] investigates the kernelization for the cycle traversal problems. The most efficient method is described in [30]. We use their method as our baseline.

B. Feedback Vertex Set

Another related work is the feedback vertex set (FVS) problem, which seeks to intersect a minimum-size of vertices with all cycles of any length in the network. [6] proposes a 2-approximation method for FVS in undirected graphs. The analogous edge version issue may be reduced to the minimum spanning tree problem [31]. As a result, it can be solved in polynomial time.

C. Cycle and Path Enumeration

With the rapid development of information technology, a growing number of applications represent data as graphs [32], [33], [34], [35], [36], [37], [38], [39], [40]. Numerous research has been conducted on the subject of enumerating *s*-*t* simple paths (e.g., [41], [42], [43], [44], [45], [46], [47], [48]), with the *simpath* algorithm proposed in [42] being one among them. Due to the huge amount of results, [14] applies the hop-constrained path enumeration problem on the dynamic graphs and analyzes them. An indexing technique named HP-index is proposed to continuously maintain the pairwise hc-*s*-*t* paths among a set of hot points (i.e., vertices with a high degree).

There is a long history of study on enumerating all simple paths or cycles on a graph [16], [17], [18], [37]. Another area

TABLE I: The summary of notations.

Notation	Definition		
G	a given directed unweighted graph		
m, n	the number of edges(vertices) for G		
C	a hop-constrained cycle cover for given graph G		
c	a simple hop-constrained cycle, where $3 \le c \le k$		
C , c	the size(length) for a vertice set $C(c)$		
$\mathcal{H}[v]$	the hit times for vertex v		
CN	the cover node		
opt(G,k)	the optimal hop-constrained cycle cover in G		
len(p)	length (i.e., number of hops) of path p		
sd(u,v)	shortest path distance from u to v		
	i.e., the 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		
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 S		
$adj_{in}[v]$	the in-neighbors of vertex v		
$adj_{out}[v]$	the out-neighbors of vertex v		
$len(\mathcal{S})$	the length of the path associated with S		
	where $len(\mathcal{S}) = len(p(\mathcal{S})) = \mathcal{S} - 1$		

of study (e.g., [49]) is counting or estimating the number of paths connecting two given vertices, which is a well-known #P hard problem.

Hop-constrained Path Enumeration [50], [51], [52]. Peng et al. [52], [47] designed a barrier-based method, which dynamically maintains the distance from each vertex to t. Initially, they set the barrier for each $v \in V(G)$ as S(v, t|G). During the enumeration, if they find that a sub-tree rooted at a node in the search tree contains no result, then they will increase the barrier to avoid falling into the same sub-tree again. T-DFS [53] and T-DFS2 [54] are two theoretical works. They achieve polynomial delay by ensuring that each search branch in the search tree leads to a result. For example, before extending M by adding v'' in Algorithm 1, T-DFS checks whether there is a shortest path from v' to t without vertices in M whose length is bounded by k - L(M) - 1. Although all three algorithms achieve $O(k \times |E(G)|)$ polynomial delay. Peng et al. showed that their method runs much faster than T-DFS and T-DFS2 in practice because their pruning strategy incurs lower overhead [52].

III. PRELIMINARIES

This section introduces the hop-constrained cycle cover problem explicitly. Then, the state-of-the-art methods are presented. Table I provides the mathematical notations that are most frequently used throughout this paper.

A. Problem Definition

This subsection begins by formally introducing the hopconstrained cycle cover problem.

G = (V, E) is a directed graph containing the vertex set V and the edge set E. The symbol $e(u, v) \in E$ denotes a directed edge connecting the vertex u to the vertex v. When the context is clear, "neighbor" refers to the "out-going neighbor". $adj_{in}[v] (adj_{out}[v])$ designate the in-neighbors (out-neighbors) of vertex v, respectively. A path p from the vertex v to the vertex v' is a sequence of vertices $v = v_0, v_1, \ldots, v_h = v'$,

where $e(v_{i-1}, v_i) \in E$ for all $i \in [1, h]$. A *circuit* is a non-empty path with repeated start and end vertices, e.g., $(v_1, v_2, ..., v_u, v_1)$. The length of a *circuit* is equal to its number of vertices. A *cycle* or *simple circuit* is a circuit where only the initial and final vertices are repeated.

The hop-constrained cycle cover problem is defined as follows:

Definition 1 (A Constrained Cycle). Given a hop constraint k, A Constrained Cycle C is a cycle with $3 \le |C| \le k$, and there are no repeat vertices except the starting and ending vertices.

Definition 2 (Hop-Constrained Cycle Cover). Assume that k is a positive integer. A hop-constrained cycle cover of graph G(V, E) is a subset of vertices $C \subset V$ such that, for any constrained cycle $C, C \cap \pi \neq \emptyset$.

The following definitions apply to the optimal and minimal hop-constrained cycle cover.

Definition 3 (Optimal Hop-Constrained Cycle Cover). The term "an optimal hop-constrained cycle cover" refers to a collection of vertices C_0 that has the smallest size of all the hop-constrained cycle covers given a directed graph G.

Definition 4 (Minimal Hop-Constrained Cycle Cover). *Given* a directed graph G, a collection of vertices C_0 is said to be a minimal hop-constrained cycle cover if no vertex in the cover could not be deleted.

This paper covers all the simple cycles (hop-constrained cycles), which is described as Theorem 1.

Theorem 1. If $C = \{ c \mid c \text{ is a cycle} \in G \}$, $C_s = \{ c \mid c \text{ is a simple cycle} \in G \}$, and a vertex set V_0 is cover set for C_s s.t. $\forall c \in C_s$, $\exists v \in V_0$, and $v \in c$, then $\forall c \in C$, $\exists v \in V_0$, and $v \in c$.

Proof. It is proven by contradiction. Assume there is a cycle $c_0 \in C$ and there does not $\exists v \in V_0$, s.t. $v \in c_0$. Notably, any non-simple cycles could be decomposed into several simple cycles. Thus, there $\exists c_1 \subset c_0$ where c_1 is a simple cycle since V_0 is a cover set for the simple cycles. Then, there must $\exists v_2 \in V_0$ and $v_2 \in c_1$. If it chooses v_2 the vertex to cover c_0 , there is a contradiction.

Remark. This is also true for hop-constrained cycles. The proof is similar. The main idea is that a k-hop constrained cycle could be divided into several small k-hop constrained simple cycles. In fact, Theorem 1 is true for any constraint if a cycle of such constraints could be decomposed into several small simple cycles that still satisfy these constraints.

Theorem 1 indicates that once all the simple circuits are covered, all the cycles are covered. Hence, only simple cycles are considered in this paper.

B. The State-of-the-art

Theoretical Perspective. The most related theoretical works are the path covers. There are two types of them, i.e., <u>k</u>-hop <u>All</u> <u>Path Cover (k-APC) and <u>k</u>-hop <u>Shortest Path Cover (k-SPC)</u>. The properties of k-APC were theoretically analyzed in [55].</u>

Unfortunately, by reduction from the vertex cover problem, the problem of minimizing the size of k-APC has proven to be NP-hard for $k \ge 2$. Furthermore, the k-APC problem cannot be approximated for $k \ge 2$ in polynomial time within a factor of 1.3606 (unless P = NP), which is also inherited from the vertex cover problem [56]. The NP-hardness and inapproximability also apply to k-SPC, as pointed out in [55]. The upper bound on the size of a k-SPC has been discussed using the theory of VC dimensions. Please refer to [55] for details.

Practical Perspective. Following that, the state-of-the-art k-cycle algorithm DARC from [30] is introduced as the baseline. Firstly, the definition of k-cycle problem is given:

Definition 5 (K-cycle problem). Given a graph G = (V, E), determine minimum-size set $S \subseteq E$ such that for each constrained cycle $C \in C_k$, $C \cap S \neq \emptyset$, or, equivalently, $C_k(G \setminus S) = \emptyset$.

Algorithm 1 illustrates the details of this algorithm. The process is to iteratively go through all edges in E in Line 2. If the current edge is not included in the result set S, then AUGMENT(e) is called. After all edges are evaluated, it is called PRUNED function.

A	Algorithm 1: DARC				
	Input : Graph $G = (V, E)$, sets $S, W, P \subseteq E, U \subseteq C_k$				
	function $h: (S \cup W) \to \mathcal{U}$				
	k: the hop constraint				
1	$S \leftarrow \emptyset, h \leftarrow \emptyset, P \leftarrow \emptyset, W \leftarrow \emptyset, \mathcal{U} \leftarrow \emptyset;$				
2	for $e \in E$ do				
3	if $e \notin S$ then				
4	$\begin{bmatrix} AUGMENT(e); \end{bmatrix}$				
5	PRUNE();				

Thus, the DARC consists of two phases.

AUGMENT. In this stage, it augments uncovered vertices and discovers uncovered hop-constrained cycles. The final cover is constructed entirely from its vertices. The details are shown in Algorithm 2. If $e \in S$, then this terminates in Line 1. It is noted that $W = \emptyset$ in the initial stage, thus it can ignore it. Then, in Line 7 it checks every constrained cycle going through e and adds all edges to the result set R and P. In addition, the relationship between constrained cycles and edges is recorded in Line 11.

PRUNE. At this step, it removes any extraneous vertices to ensure that the result set remains feasible. The details are in Algorithm 3. For every candidate edge $e \in P$, it tries to remove it from the result set S in Line 4. If yes, it will delete e from S and add it to W.

The specifics of these two phases may be seen in [30], the source code for which is publicly accessible.

Modification to the vertex version. Since DARC is a method developed for discovering a minimum edge subset that does not intersect with any hop-constrained cycles, it is modified as the baseline. The modification is as follows: for the original graph G(V, E), it is converted to a new graph G'(V', E'). For every $e_{u,v} \in E$, it generates a corresponding $v_{u,v} \in V'$, an edge e' is added from vertices $v_{u,v}$ to $v_{v,w}$ since

Algorithm 2: AUGMENT(e)

```
Input : Graph G = (V, E), sets S, W, P \subseteq E, U \subseteq C_k
              function h: (S \cup W) \to \mathcal{U}
              k: the hop constraint
1 if e \in S then
2 Return;
3 else if e \in W then
       Move e from W to S;
4
       Add e to P;
6
       Return;
7 for C \in \Delta_k(e) do
       if C \cap S = \emptyset then
8
            if C \cap W = \emptyset then
9
                 Add all edges of C to S and P;
10
                 Add C to \overline{\mathcal{U}} and set h(e) = C for all e \in C;
11
12
            else
                Move any edge in C \cap W to S and P;
13
```

Algorithm 3: PRUNE()

Ir	uput : Graph $G = (V, E)$, sets $S, W, P \subseteq E, U \subseteq C_k$
	function $h: (S \cup W) \to \mathcal{U}$
	k: the hop constraint
1 fo	$\mathbf{r} \ e \in P \ \mathbf{do}$
2	$P \leftarrow P \backslash \{e\};$
3	if $e \in S$ then
4	if $S \setminus e$ remains feasible to a cover then
5	$ S \leftarrow S \setminus \{e\};$
6	$ W \leftarrow W \cup \{e\}; $
	_ L

there is a common v between them. Then, the edge set could be converted to the vertex result set. The modified algorithm is called DARC-DV. As demonstrated in [30], the worse case time complexity of DARC-DV is $O(n^k)$. This paper intends to enhance the findings in terms of cover size and efficiency.

IV. THEORETICAL ANALYSIS

This section begins with a theoretical study of the hopconstrained cycle cover issue.

Theorem 2 proves that it is NP-hard to decide whether there is a hop-constrained cycle cover for a given directed, unweighted graph G with size s.

Theorem 2. Deciding whether there is a hop-constrained cycle cover for a given directed, unweighted graph G with size s is NP-hard for all constrained cycles.

Proof. It reduces an NP-complete problem, the decision version of vertex cover problem to our hop-constrained cycle cover problem. Take note that the hop-constrained cycle cover issue does not include self-loops or bidirectional edges. As a result, it sets k = 3 for a hop-constrained cycle cover issue on undirected graphs. Then, it can reduce the traditional vertex cover issue to ours.

It adds a virtual vertex u' and two bidirectional edges (u, u')and (v, u') to each bidirectional edge u, v. Figure 2 illustrates the proof. The original graph is shown in Figure 2(a), whereas our constructed graph is shown in Figure 2(b). Take note that the virtual vertex b' is dominated by the matching edge (b, c), since b' only participates in only single cycle. On the built graph, the hop-constrained cycle cover problem is equivalent to the classical vertex cover on the original graph.



Fig. 2: The proof of NP-hardness.

After that, this work demonstrates the inapproximation.

Theorem 3. For a given directed, unweighted graph G, approximating the hop-constrained cycle cover problem (length from 3 to k) within $(k - 1 - \epsilon)$ is UGC-hard (Unique Games Conjecture).

Proof. Guruswami and Lee [57] established the UGC-hard for the inapproximability of the feedback set for bounded cycles. Our issue is with the version that excludes self-loops and cycles with a length of 2. Due to the fact that self-loops have a single vertex, they could be included in the result set. As a consequence, it places a premium on excluding the 2-cycle. The formal proof of inapproximability when selfloops are included is similar to the proof of 2-cycle exclusion. It is proved by contradiction. Assume that a $(k - 1 - \epsilon)$ approximation method exists for covering all cycle lengths between 3 and k.

S(G,3,k) denotes the approximation method, with G denoting the graph, and 3 and k denoting the length is between 3 and k. $|S(G,3,k)| \leq (k-1-\epsilon)|Opt(G,3,k)|$, where S is the approximation algorithm and Opt is the optimal algorithm for cycles in graph G lengths ranging from 3 to k. Furthermore, $|Opt(G,2,k)| \geq |Opt(G,2,2) + Opt(G - (Opt(G,2,2)), 3, k)|$. There is a simple 2-approximation method for 2-cycle by selecting all the vertices on them, i.e., $S(G,2,2) \leq 2|Opt(G,2,2)|$.

All the cycles could be divided into three types.

- C₂: 2-cycle (length equals to 2) and no intersection with k-cycles (k ≥ 3).
- C_3 : k-cycles and no intersection with 2-cycle.
- C_{23} : 2-cycles intersect with k-cycles ($k \ge 3$) and k-cycles intersect with 2-cycles.

It divides all the vertices on cycles into three categories.

- V_2 : only appears on the C_2 .
- V_3 : only appears on the C_3 . Note that it only includes the k-cycles, where all the vertices on it are in V_3 . If a k-cycle intersection with a 2-cycle (C_{23}), it only selects the intersection part into V_{23} . The remaining part would be covered because our approximation algorithm would select all the intersection vertices.
- V_{23} : appears on the intersection part of 2-cycles and k-cycles ($k \ge 3$). For example, if vertex v is in both a 2-cycle and a k-cycle, then it is in the V_{23} .

Similarly, Opt(G, 2, k) could be divided into V_2^{Opt} , V_3^{Opt} , and V_{23}^{Opt} . V_2^{Opt} is the intersection of Opt(G, 2, k) and C_2 . For V_3^{Opt} ,

$$S(V_3, 3, k) \le (k - \epsilon)Opt(V_3, 3, k)$$
$$\le (k - \epsilon)V_3^{Opt}$$
(1)

For V_2^{Opt} , there is a trivial 2-approximation algorithm. Thus,

$$S(V_2, 2, 2) \le 2Opt(V_2, 2, 2) \le V_2^{Opt}$$
(2)

For V_{23}^{Opt} , it selects all of them, and it is not hard to prove that $S(V_{23}^{Opt}, 2, k) = V_{23} \leq 2Opt(V_{23}^{Opt}, 2, k)$. The proof is similar to the approximation of V_2 . It could also divide these cycles into 2-cycle and k-cycle, while V_{23} is the 2-approximation solution of 2-cycle part. The Optimal solution of all cycles is no less than the 2-cycle part.

Since the limit of ϵ is close to 0,

$$S(V_{2}, 2, 2) + S(V_{3}, 3, k) + S(V_{23}, 2, k)$$

$$\leq 2Opt(V_{2}, 2, 2) + (k - \epsilon)Opt(V_{3}, 3, k) + V_{23}$$

$$\leq 2V_{2}^{Opt} + (k - \epsilon)V_{3}^{Opt} + 2V_{23}^{Opt}$$

$$\leq (k - \epsilon)Opt(G, 2, k)$$
(3)

This violates the inapproximability for Opt(G, 2, k) within $(k - 1 - \epsilon)$.

V. BOTTOM-UP APPROACH

This section introduces our bottom-up hop-constrained cycle cover algorithm, which is designed to minimize the size of the cover.

A. Motivation

This subsection discusses the rationale for the *bottom-up* hop-constrained cycle cover algorithm.

Given the *NP-hardness* of this problem, it is typical to employ the greedy heuristic. That is to determine the best cover vertex iteratively, i.e., the vertex that covers the largest number of uncovered cycles in the current iteration. Nonetheless, determining the optimal cover vertex requires enumerating all the hop-constrained cycles, which are prohibitively difficult in terms of time and space complexities. The enumerating time complexity $O(2^n \times cost_c)$ is prohibitively expensive, while $cost_c$ is the cost for the check.

The more hop-constrained cycles a vertex covered in the previous iterations, the more probable it will cover additional cycles in the remaining graph. Based on this motivation, a heuristic greedy algorithm is proposed based on the number of cycles the vertices covered.

Example 1. A motivational scenario is depicted in Figure 3. C is the graph's center vertex. Assume that in the first iteration, a cycle $A \rightarrow B \rightarrow C$ is found and it adds A to the cover set by random. C will be chosen in the second iteration since it occurred in the preceding cycle and is therefore more likely to cover more cycles.

Based on this example, a bottom-up hop-constrained cycle cover method is proposed. The key idea is that when discovering a cycle during the search, it records hit-times (\mathcal{H}) of all the vertices on it and chooses the one with the highest hit-times (\mathcal{H}).



Fig. 3: A motivation example of the bottom-up approach.

B. The Bottom-Up Approach

The greedy method is an efficient solution for solving the hop-constrained cycle cover problem. The key principle is that for each iteration, we select the vertex that covers the most hop-constrained cycles. Due to the difficulty of enumerating all the hop-constrained cycles, a heuristic algorithm is proposed to solve it.

Algorithm 4: BOTTOM-UP (G, k)					
$1 \mathcal{R} \leftarrow \emptyset;$					
2 $\mathcal{H}[v] \leftarrow 0$, for each $v \in G$;					
3 for each $v_i \in V$ do					
4 $c \leftarrow \text{FINDCYCLE}(G, k, \emptyset, v_i);$					
5 while $c \neq \emptyset$ do					
6 for each $v \in c$ do					
7 $\ \ \ \ \ \ \ \ \ \ \ \ \ $					
8 $u \leftarrow \text{FINDCOVERNODE}(v_i, \mathcal{H}, c);$					
9 Insert u into R ;					
10 Remove in-edges and out-edges of $u \in G$;					
11 $c \leftarrow \text{FINDCYCLE}(G, k, \emptyset, v_i);$					

Algorithm 4's main idea is to find a cover vertex with the \mathcal{H} array. Line 2 initializes $\mathcal{H}[v]$ to 0 for each vertex v on graph G. Then, Line 3 is a for-loop that iterates over all vertices in G. Lines 4 and 11 attempt to identify a hop-constrained cycle c beginning at v. The \mathcal{H} array is updated for each vertex on c. Line 8 specifies that it selects one vertex u from c, and eliminates all of u's associated edges in Line 10. Whenever $c \neq \emptyset$ in Line 5, the algorithm continues the procedure.

Algorithm 5 employs a recursive approach to find a hopconstrained cycle starting from the vertex v. The graph G is a reduced graph, since it has no vertex in the current result set R. The key point is to identify a hop-constrained cycle using a DFS method. Line 3 demonstrates that this method identifies a valid hop-constrained cycle, with the condition CD > 0indicating that self-loops should be avoided. Line 5 illustrates the situation where the algorithm fails to locate a valid hopconstrained cycle. Line 8 investigates all of v's out-neighbors. Then, Lines 7 and 12 are to recursively push and pop all v's out-neighbors.

Algorithm 6 employs \mathcal{H} to determine the cover vertex. Line 2 attempts to locate the vertex with the maximum \mathcal{H} .

Algorithm 5: FINDCYCLE(G, k, CP, v)

1 $\mathcal{C} \leftarrow \emptyset$: 2 $\mathcal{CD} \leftarrow \operatorname{len}(\mathcal{CP});$ /* the current distance */; 3 if $\mathcal{CD} > \mathbf{0} \land v = \mathcal{CP}[0]$ then Return CP; $s~\text{if}~\mathcal{CD} > k~\text{then}$ Return \emptyset ; 7 CP.pushBack(v); s for each vertex u of $adj_{out}[v]$ on G do $\mathcal{C} \leftarrow \text{FINDCYCLE}(G, \mathbf{k}, \mathcal{CP}, u);$ 9 if $C \neq \emptyset$ then 10 Return C; 11 12 *CP*.pop();

		_	_	
13	Ret	nr	n	\mathcal{C} :

Algorithm 6	5 :	FINDCOVERNODE (v, \mathcal{H}, c))
-------------	------------	-------------------------------------	---

Correctness. Since Algorithm 4 traverses all the vertices in V and increments the result set by one vertex until no new constrained cycle is detected, it is self-evident that the induced graph (which is constructed by eliminating all the vertices from the cover set) contains no hop-constrained cycles. Assume that there is a hop-constrained cycle c_0 , which is initially explored by Algorithm 4 with vertex v_c . When we explore vertex v_c for the first time, the *FindCycle* produces a hop-constrained cycle c_i . The c_i could be c_0 or not. There are two cases.

- If $c_i = c_0$, our algorithm covers c_0 in the subsequent steps.
- If $c_i \neq c_0$, we choose a cover vertex CN.

Additionally, there are two scenarios depending on whether $CN = v_c$ or not if $c_i \neq c_0$.

- If $CN = v_c$, the c_0 is covered by vertex v_c .
- If $CN \neq v_c$, we will continue to check vertex v_c , in accordance with Line 9 Algorithm 4.

Thus, the c_0 will be covered by the result set of Algorithm 4.

Time and Space Complexities. Algorithm 4 employs an array of \mathcal{H} and a k-step DFS procedure. As a result, the space complexity is O(m). Algorithm 4 Line 3 needs n iterations. Each iteration contains three steps: FindCycle, UpdateH, and FindCoverNode. Both UpdateH and FindCoverNodetakes a time complexity of O(k) time complexity, due to the fact that they include a for-loop on hop-constrained cycle c. FindCycle takes $O(n^k)$, since it is a DFS algorithm that determines whether there is a hop-constrained cycle within k steps. Thus, the overall time complexity is $O(n^{k+1})$.

Nonetheless, the practical performance is acceptable, due to the following reasons. To begin, *FindCycle* could early terminate when it finds a hop-constrained cycle. It is not necessary to locate all the hop-constrained cycles. Second,

each time we choose a cover vertex, the in-edges and outedges of it would be eliminated from the graph G. Thus, the graph is becoming smaller and smaller.

C. The Minimal Pruning Algorithm

This subsection proposes a minimal pruning method s.t. it can further reduce the hop-constrained cycle cover to a *minimal* result set. The main idea is to remove every possible vertex that can be eliminated until the minimal one.

Algorithm 7: FINDMINIMALCOVER (G, k, R)
1 for each $v \in \mathcal{R}$ do
2 $c \leftarrow \text{FINDCYCLE}(G - R + (v), k, \emptyset, v);$
3 if $c = \emptyset$ then
4 $\[\] Remove v from \mathcal{R};$
5 Return \mathcal{R} ;

Algorithm 7 Line 1 verifies each vertex in the R, which is the hop-constrained cycle cover generated by Algorithm 4. Line 2 attempts to find a hop-constrained cycle inside G-R+(v). Nota bene, in Algorithm 7, the input graph G-R+(v)is the reduced graph, which has no vertex in R except for the vertex v. Line 3 determines if there exists a hop-constrained cycle for v. Otherwise, v will be deleted from the R.

Theorem 4 establishes that Algorithm 7 provides a minimal solution of hop-constrained cycle cover.

Theorem 4. Algorithm 7 returns a feasible and minimal hopconstrained cycle cover *R*.

Proof. Take note that Algorithm 4 generates the input vertex set R. R is a viable hop-constrained cycle cover, as its correctness has been proven. Algorithm 7 removes a vertex v from R only if the reduced G contains no hop-constrained cycle. As a consequence, the result set R includes all the hop-constrained cycles after the termination.

As for the minimality property, if Algorithm 7 does not prune each vertex v, there will exist a witness hop-constrained cycle c_w , where $c_w \cap (R - \{v\}) = \emptyset$. Given that the result set is $R_f \subset R$, then $c_w \cap (R_f - \{v\}) = \emptyset$. Thus, if any vertex v in the final result set R_f is deleted, no vertex in $R_f - \{v\}$ will cover the witness hop-constrained cycle c_w . Thus, R_f is a hop-constrained cycle cover of G that is both feasible and minimal.

Time and Space Complexities. Since there is no index and FindCycle is a DFS algorithm, its space complexity is O(m). Following that, the time complexity is investigated. Line 1 of Algorithm 7 contains only one for-loop. Given that the size of R is no larger than n, Line 2 requires at most n iterations. In the worst-case scenario, the procedure FindCycle requires $O(n^k)$. As a result, the overall time complexity is $O(n^{k+1})$.

VI. TOP-DOWN ALGORITHM

This section presents the Top-Down method in this section, which aims to improve the efficiency.

A. Motivation

Why Costly? The most expensive aspect of the hopconstrained cycle cover problem is the repeated usage of the constrained cycle search. This paper accelerates it from two aspects.

- Decrease the Search Space. The purpose of researching top-down algorithms is to reduce search space. We must verify all of the vertices. The search spaces in the bottom-up method range from the whole graph G to the graph G R, where R is the cover set. In the top down algorithm, the search areas would range from \emptyset to G R. As a result, the top-down approach has the potential to substantially shrink the search space.
- Increase the Speed of the Cycle Search Function. Since the cycle search function is frequently utilized in the whole process, it is one of the bottlenecks. Thus, this work proposes a delicate block-based and BFS-filterbased method to accelerate this process.

B. Top-Down Algorithm Description

The key idea of Top-Down algorithm is different from that of the bottom-up algorithm. The method begins with an empty graph G_0 and a full cover set with all vertices in it. Then, it evaluates each vertex v in it. It determines whether or not to remove v from the result set. If true, all in-edges and out-edges of v are inserted into G_0 . The cycle validation algorithm is adapted from [52]. The general idea is as follows: It conducts a DFS search to validate whether there is a cycle including the query vertex. In the DFS search, for each vertex u, if it has been searched before and we can guarantee that it is not included in a cycle with a certain length threshold, we can avoid searching u. The length threshold is at most t, and it is maintained during the search. In this way, the threshold value is updated at most k times and every time it explores at most m edges. Therefore, the Top-Down algorithm runs in O(kmn)time, where n is the number of vertices to be validated.

Algorithm 8 Line 1 initializes the graph for verifying the node necessary. Line 2 is a for-loop that verifies all of the cover set's vertices. Lines 3 and 4 attempt to insert all of the vertex v's edges and determine whether there exists a constrained cycle. If not, the vertex v is deleted from the cover set in Line 6. Otherwise, vertex v is maintained, but all of its edges are removed in Line 8.

Algorithm 8: TOP-DOWN (G, k, R)				
$1 \ G_0 \leftarrow \emptyset, \ \mathcal{R} \leftarrow G ;$				
2 for each $v \in \mathcal{R}$ do				
3 Insert all in-edges and out-edges of v into G_0 ;				
4 $c \leftarrow \text{FINDCYCLE}(G_0, k, v);$				
5 if $c = \emptyset$ then				
6 Remove v from \mathcal{R} ;				
7 else				
8 Delete all in-edges and out-edges of v ;				
9 Return \mathcal{R} ;				

C. Node Necessary Validation

A frequent operation is to validate whether a vertex v is in a constrained cycle in the current graph G_0 . A straightforward method is modified DFS BFS. As for modified BFS, Figure 4 demonstrates counter-examples.

Example 2. Specifically, by executing BFS from vertex v and assigning a new color to each neighbor. When it explores a vertex further and locates a vertex with a neighbor of a different color for the first time, it discovered a shortest cycle through v. Nevertheless, when beginning from vertex a, the modified BFS was unable to distinguish between Figure 4(a) and 4(b). When the modified BFS is employed, b and c are marked as visited during the first iteration. Then, in the third iteration, it returned to the edge (d, c) of c. In such an instance, it is unable to distingue 4(a).



Fig. 4: Counter-examples for the modified BFS.

To accelerate Node Necessary Validation, an O(km) time complexity method is proposed, which is inspired by the barrier technique in [52]. Firstly, the block for a given vertex u during the search procedure is formally defined. It could be regarded as the lower bound of dis(u, s) in the current stack S. s is the starting vertex. The block is utilized to prune unnecessary candidates.

Definition 6. (*u.block*) For a given vertex u, *u.block is correct if and only if given the current stack* S, *if there is a path* $p(u \rightarrow s)$, *not containing any vertex in* S, *we have* $len(p) \ge u.block$, *i.e.*, $sd(u, s|S) \ge u.block$.



Fig. 5: An example of the block idea.

An example of the *block* is illustrated in Figure 5.

Example 3. Assume vertex a is the starting vertex with k = 5. With a DFS search, it validates whether there exists a constrained cycle containing vertex a. In the first iteration, path $a \rightarrow b_1 \rightarrow c \rightarrow d$ is checked. In this path, vertex c cannot reach vertex a in 5 - 2 = 3 hops. Then, c.block is set to 3 + 1 = 4. This block information may be utilized in the subsequent DFS exploration, e.g., $a \rightarrow b_2 \rightarrow c \rightarrow d$. It may end prematurely when exploring $a \rightarrow b_i \rightarrow c$. In this

path, there are 5-2=3 remaining depths for DFS, which is smaller than c.block.

The idea of *block* is to utilize the failure information to prune invalid search space. The algorithm is formally described as follows: S is used to denote the stack of the currently explored path. S_{old} indicates the past exploration path, whereas S_{new} denotes the current exploration path. Algorithm 9 illustrates the whole process.

Algorithm 9: NODENECESSARY (s, u, S, R, G')				
1 if $\mathcal{R} \neq \emptyset$ then				
2 Return R ; /* Vertex s is necessary */;				
3 $u.block \leftarrow k - len(\mathcal{S}) + 1$;				
4 \mathcal{S} .pusn(u); : if $u = a \wedge lon(S) > 2$ then				
5 If $u = S \land ien(S) \ge 2$ then 6 $\downarrow u$ is unstacked from S:				
7 UNBLOCK $(S.top(), S, 1)$:				
s if $len(\mathcal{S}) > 2$ then				
9 insert $p(S)$ into \mathcal{R} ;				
10 return \mathcal{R}				
11 else if $len(S) < k$ then				
12 for vertex v of $adj_{out}[u]$ where $v \notin (S - \{s\})$ do				
13 if $len(S) + 1 + v.block \le k$ then				
14 $\mathcal{R} \leftarrow \text{NODENECESSARY}(s, v, S, \mathcal{R}, G');$				
15 if $\mathcal{R} \neq \emptyset$ then				
16 Return \mathcal{R} ;				
17 u is unstacked from S ;				
18 return R				

Algorithm 10:	UNBLOCK (u, \mathcal{S}, l)
1 $u.block = l$; 2 for each vertex 3 $if v.block > d$ 4 UNBLOO	$x v \text{ of } adj_{in}[u] \text{ with } v \notin S \text{ do}$ > $l + 1 \text{ then}$ CK $(v, S, l + 1);$

Details. Algorithm 9 Line 1 terminates the recursive algorithm if a valid constrained cycle is found. Line 3 initializes the block value associated with the current vertex u to k - len(S + 1). Line 5 is the condition for a valid hop-constrained cycle. Line 8 determines whether it is a valid constrained cycle.

If *true*, the cycle will be inserted into the result set \mathcal{R} . Line 13 is the condition for blocking. When $len(\mathcal{S}) + 1 + v.block > k$, vertex v is blocked. The worst-case time complexity is O(km) due to the block level of each vertex ranging from 0 to k. Algorithm 10 iteratively updates the block values for the vertices whose block is larger than l.

Modification to Cycle Cover without Constraints. To cope with the variant without hop constraint, we only need to modify the node necessary function. The modification can be summarized as the following steps:

- i) Replace u.block ← k − len(S) + 1 to u.block ← ∞ in Line 3 Algorithm 9.
- ii) Remove condition len(S) < k in Line 11 Algorithm
 9.
- iii) Replace len(S) + 1 + v.block ≤ k with v.block ≠ ∞ in Line 13 Algorithm 9.

D. Analysis

This part proves the correctness of Algorithm 9. To begin, the condition of the correct u.block value is given.

Lemma 1. *u.block is correct iff given the stack* S*, there is a path* $p(u \rightarrow s)$ *without any vertex in* S*,* $len(p) \ge u.block$, *i.e.,* $sd(u, s|S) \ge u.block$.

The *budget* of a vertex u is defined as follows:

Definition 7 (budget). *budget[u] is the number of hops remaining for u to continue the search.*

$$budget[u] = k - len(\mathcal{S}), when \ u = S[0]$$
(4)

The following lemma gives the condition under which a vertex u may reach the target vertex s in Algorithm 9.

Lemma 2. Assume that the top vertex of the current stack S is u. There is a path $p(u \rightarrow s)$. The vertex u could reach the vertex s in Algorithm 9 only if $k - len(S) \ge len(p)$ and every vertex in the path (except u) is not included by S.

Proof. The inequation $k - len(S) \ge len(p)$ indicates that the vertex u has a sufficient *budget* to reach t. Since S does not include all vertices $\{x\}$ along the path, the search can only be early terminated due to their block values. Because x is not in the S and x.block is correct w.r.t S, $x.block \le p[x]$, as x can reach t within p[x] hops. Thus, x cannot use the block value to terminate the search. Thus, u can reach t in Algorithm 9. \Box

The correctness of Algorithm 9 is proved by demonstrating that block values are correct throughout it.

Theorem 5. All the block values are correctly computed, and they remain correct in Algorithm 9.

Proof. Firstly, *u.block* is correct if $u \in S$. When *u.block* is set at Line 3, the value is correct w.r.t S.

It is demonstrated that u.block is properly specified in the following search. If a new vertex v is pushed into S, then u.block is immediately correct since $S = S \cup \{v\}$ leads to a strictly smaller search space.

Consequently, the sole remaining scenario is vertexs' unstacking. The vertex v denotes the first vertex that leads u.block to be incorrect. If unstack of v does not affect u.block, u.block is still correct for the new stack $S \setminus \{v\}$. Alternatively, u.block may be updated in two cases:

- (1) If v = u, *u.block* is still *true*. When a valid hop-constrained cycle containing u exists, the algorithm terminates. Then, u will not be unstacked due to the early termination. Thus, the unstack of vertex u indicates that the current S does not include any valid hop-constrained cycle.
- (2) Given v ≠ u and that the unstack of v impacts the u.block. Thus, there exists a path p(u → v → s), which does not include any vertex in S, s.t. len(p) < u.block. That is, u can reach t with fewer hops due to v's unstack from S. Assume that vertices' block values are properly maintained before vertex v's unstack.

Assume v cannot reach t in Algorithm 9, and u.block > len(p) = p(u) with respect to the current stack S. It indicates that although v.block was correctly maintained in the previous step, it is incorrect due to unstack of v and u.block > p(u).

Three vital timestamps occurred throughout the proof. They are T_{inU} , T_{outU} , and T_{outV} . T_{inU} indicates the time when u is added into the stack. T_{outU} and T_{outV} indicate the time when the first³ unstack of u and v, respectively, after T_{inU} . $S_0(y)$ indicates the stack size when the vertex y is pushed to the stack S_0 . Note that $T_{inU} < T_{outV} < T_{outU}$, and $S_0(v) < S_0(u)$.

If v is the sole vertex that blocks $u \to s$ at T_{inU} , the algorithm will terminate before T_{outV} . Note that $S_0(v) < S_0(u)$. According to Lemma 2, the vertex v can reach the vertex s, and according to Algorithm 9 Line 1 and 15, the unstack of v is early terminated. It violates the assumption that there exists unstack of v. The proof is similar if there exist more than one vertices.

Correctness. According to Theorem 5, the vertices' block values are correct. Algorithm 9 is a hop-constrained DFS that utilizes a block-based technique. If the hop-constrained DFS and the block-based method are valid, the algorithm is correct. Notably, Algorithm 9 guarantees the simple cycle property by default.

Time Complexity. The following theorem indicates that Algorithm 9 is O(km). The time complexity of TDB++ is $O(k \cdot m \cdot n)$.

Theorem 6. Algorithm 9 is capable of returning a valid answer in O(km).

Proof. Assume that Algorithm 9 unstacked a vertex u twice. This implies that none of these two unstacks is early terminated. Let S_1 and S_2 denote the stacks after the first and second times that u is pushed into the stack, respectively. After unstack u for the first time, $u.block = k - S_1 + 1$. When u is pushed to the stack at the second time, u.block remains unchanged. As u passes block conditions in the second visit with $S_2 + u.block \leq k$, and thus $S_2 < S_1$. Consequently, u.block will be increased by at least one every time u is unstacked.

This indicates that a vertex may be pushed to stack no more than k times. When u is added into the stack, an edge (u, v)is visited. An edge is visited at most k + 1 times. Hence, the time complexity of it is O(km). It is worth noting that omitting the bidirectional edges as cycles has no effect on the time complexity. Assume the start vertex is v, and u, w are its bidirectional out-neighbors. Assume that the unblock of u and w would have an effect on the vertex sets A_u and A_w . Then either $A_u \cap A_w = \emptyset$, or there exists a constrained cycle when u and w are explored. Both of them show that Algorithm 9 has an O(km) time complexity.

Space Complexity. Since the stack size is always bounded by k, the space complexity of Algorithm 9 is O(m + k).

Theorem 7. Algorithm 8 produces a hop-constrained cycle cover R that is both valid and minimal.

³A vertex may be pushed and unstack for many times.

The proof is similar to the proof of Theorem 4.

Comparison with Barrier Technique. Firstly, the hopconstrained path enumeration problem focuses on how to efficiently enumerate all the paths, but in the constrained cycle cover problem, the point is how to efficiently detect the existence of any constrained cycle. There are many algorithms in the problem of hop-constrained path enumeration, IDX-DFS [58], IDX-JOIN [58], PathEnum [58], BC-DFS [52], T-DFS [53], T-DFS2 [54], and JOIN [52]. IDX-DFS, IDX-JOIN, PathEnum, and JOIN are more efficient than BC-DFS in terms of hop constrained path enumeration, but their technique is not suitable to adapt to the constrained cycle cover problem due to the different focuses of these two different problems. They either need preprocessing costs to construct a light-weighted index [58] or find the middle cut (JOIN [52]) to accelerate the whole enumerate process. In the context of hop-constrained path enumeration, such cost is affordable since the bottleneck is the heavy enumeration stage due to a large number of results. Nevertheless, for the constrained cycle cover problem, only one cycle is needed, then the bottleneck is altered.

E. Upper Bounds Filtering

This subsection introduces an upper bound to filter some unnecessary vertices.

BFS-filter technique. According to Example 2, a modified BFS could not examine whether a vertex is necessary in the *Top-Down* algorithm. Nevertheless, for a vertex v, if its distance to itself is larger than k in the modified BFS, then it could be safely excluded in the current iteration.

Details. The details are presented as Algorithm 11. Line 1

1	Algorithm 11: BFS-FILTER(G_0, k, v)
1	$\mathcal{U} \leftarrow$ the upper bound distance from v to v using the
	modified BFS;
2	if $\mathcal{U} > k$ then
3	Prune vertex v ;
4	else
5	Vertex v needs further verify.;

computes the upper bound of the distance of v to itself using the modified BFS (see Example 2 for details) and represents it with \mathcal{U} . Two cases for the BFS-filter technique method are shown in Lines 2 to 4. When $\mathcal{U} > k$, the vertex is pruned safely. Line 4 represents the situation where $\mathcal{U} \le k$. Then, the vertex has to be verified further using Algorithm 9.

VII. EXPERIMENTAL RESULTS

This section evaluates the effectiveness and efficiency of the proposed techniques on comprehensive experiments.

A. Experimental Settings

Compared Algorithms. The following baselines are compared in the experimental part.

- *DARC-DV*. The state-of-the-art algorithm [30] introduced in Section III-B.
- BUR. The bottom-up approach introduced in Section V-B.
- *BUR+*. The bottom-up approach with the minimal technique introduced in Section V-C.

TABLE II: Statistics of datasets. K indicates 10^3 . M indicates 10^6 . B indicates 10^9 .

Name	Dataset	V	E	d_{avg}
WKV	Wiki-Vote	7K	104K	29.1
ASC	as-caida	26K	107K	8.1
GNU	Gnutella31	63K	148K	4.7
EU	Email-Euall	265K	420K	3.2
SAD	Slashdot0902	82K	948K	23.1
WND	web-NotreDame	325K	1.5M	9.2
CT	citeseer	384K	1.7M	9.1
WST	webStanford	281K	2.3M	16.4
LOAN	prosper-loans	89K	3.4M	76.1
WIT	Wiki-Talk	2.4M	5.0M	4.2
WGO	webGoogle	875K	5.1M	11.7
WBS	webBerkStan	685K	7.6M	22.2
FLK	Flickr	2.3M	33.1M	28.8
LJ	LiverJournal	10.6M	112M	21.0
WKP	Wikipedia	18.2M	172M	18.85
TW	Twitter(WWW)	41.6M	1.47B	70.5

- *TDB*. The <u>Top-Down</u> <u>Blocks</u> algorithm introduced in Section VI.
- *TDB*+. The Top-Down Blocks algorithm with the block technique introduced in Section VI.
- *TDB*++. The Top-Down Blocks algorithm with the block and BFS-filter techniques introduced in Section VI.

Datasets. Table II summarizes the key statistics about the real graphs used in the experiments. Most of these graphs are from either SNAP [59] or KONECT [60].

Settings. All programs were implemented in standard C++11 and compiled using G++4.8.5.

All experiments were performed on a machine with 36X Intel Xeon 2.3GHz and 385GB main memory running Linux (Red Hat Linux 7.3 64 bit).

TABLE III: The cover size (the number of vertices) and runtime (seconds) for algorithms when k = 5.

Name		DARC-DV		BUR+		TDB++
	Size	Time	Size	Time	Size	Time
WKV	490	53.8	469	402.8	491	0.41
ASC	620	2.42	607	44.01	612	0.11
GNU	184	1.3	180	1.49	193	0.69
EU	622	114.7	609	702.1	627	1.25
SAD	6,377	440.1	6,005	4,717	6,380	3.13
WND	27,067	29,916.8	23,853	28,953.3	24,290	2.67
CT	1,621	37.03	1,610	43	1,611	16.2
WST	31,253	140.7	30,811	275.6	31,148	2.99
LOAN	332	184.5	320	450.7	347	127.9
WIT	7,040	2,296.8	6,923	4,708.3	6,894	56.3
WGO	130,382	42.2	129,009	110.8	129,421	5.99
WBS	98,570	3,571.4	94,817	12,739	100,668	6.96
FLK	-	-	-	-	206,912	92.3
LJ	-	-	-	-	39,183	20,466.8
WKP	-	-	-	-	685,759	4,132
TW	-	-	-	-	3,731,522	89,634

B. The Speedup Effects

In this subsection, all the techniques in Top-Down are evaluated. Figure 10 illustrates the speed-up benefits of all the techniques in WKV and WGO, varying k from 3 to 7. What is remarkable about the figures is that when k is large, the BFS-filter technique contributes more speedup effect than the block technique. The insight is that the BFS-filter technique is a linear filter technique and is effective in a wide variety of situations. Nonetheless, both the block technique and BFSfilter technique comparable speed-up effects in both



Fig. 7: Cover size (# of vertices).



Fig. 9: Cover size (# of vertices).

datasets when k is small. Since the result sets generated by all three methods are identical, their cover sizes are not reported.



Fig. 10: Runtime (s) for Top-Down techniques.

C. Effectiveness and Efficiency on hop-constrained cycle cover

In this part, the effectiveness and efficiency of all the experiments are discussed as follows. On real datasets, Table III presents the cover size and runtime for BUR+, DARC-DV, and TDB++. The k is set to 5 in this experiment. As demonstrated in Table III, BUR+ consistently returns the smallest cover size in 11 datasets except for WIT, despite using more time among all the algorithms. It is apparent from this table that TDB++ method produces the smallest cover size in WIT among 12 datasets. In the remaining 11 datasets, it provides a cover size that is comparable to BUR+, with an average difference of less than 4%. TDB++, on the other hand, runs 3 orders faster than BUR+ and up to 4 orders in WND. When compared to DARC-DV, what stands out in the table is that TDB++ runs about 2-3 orders of magnitude faster while returning a comparable cover size. Notably, only TDB++ was capable of producing results on large graphs, i.e., FLK, LJ, WKP, and TW.

D. Tuning the Parameter k

Additionally, experiments are conducted by varying the parameter k. This experiment is conducted with tuning the value of the parameter k from 3 to 7 on 12 distinct datasets

to determine the cover size and runtime. As shown in Figure 6, TDB++ is the fastest algorithm across all the datasets, followed by DARC-DV. The figure demonstrates that BUR+ runs slowest. Nevertheless, what stands out in Figure 7 is BUR+ generates the smallest cover size. TDB++ produces a comparable cover size as BUR+ but has the fastest runtime. As for DARC-DV, it returns the worst cover size among these three methods.

E. The Pruning Effects

This subsection conduct experiments to demonstrate the pruning effects. As shown in Figure 8, BUR and BUR+ have a similar runtime in both WKV and WGO. Nevertheless, it is shown in Figure 9 that BUR+ has a smaller cover size owing to the minimal pruning approach in both datasets. WKV and WGO vary in that WKV has a higher average degree than WGO. In WKV, it could prune more percentage results. As for WGO, the cover size difference between BUR+ and BUR stays steady when k grows.

F. Cover Size including 2-cycles

Table IV illustrates the cover size for our algorithm when including 2-cycles or not. What stands out in this table is that the cover size would be 3 times larger on average when including 2-cycles. For some graphs, e.g., GNU, the cover size does not grow too much. Nevertheless, for graphs ASC, SAD, WND, CT, WST, WIT, WGO and WBS, the cover size significantly grows. Since 2-cycles could be efficiently verified separately, our problem concentrates on constrained cycles without 2-cycles.

TABLE IV: The cover size (the number of vertices) k = 5.

Name	No 2-cycle	With 2-cycle	Ratio
WKV	491	714	1.45
ASC	612	5,285	8.64
GNU	193	222	1.15
EU	627	1,270	2.03
SAD	6,380	27,461	4.30
WND	24,290	51,466	2.12
CT	1,611	7,615	4.73
WST	31,148	116,065	3.73
LOAN	347	568	1.64
WIT	6,894	21,781	3.16
WGO	129,421	217,799	1.68
WBS	100,668	256,281	2.55

VIII. CONCLUSION

This paper introduced the hop-constrained cycle cover problem, whose objective is to discover a collection of vertices that covers all hop-constrained cycles in a given directed graph. On the theoretical side, this work demonstrates that approximating the hop-constrained cycle cover issue with length between 3 and k is UGC-hard (Unique Games Conjecture) for a given directed, unweighted graph G. Our comprehensive experiments show the effectiveness and efficiency of our proposed methods in terms of cover size and runtime when compared to the stateof-the-art k-cycle traversal algorithm DARC-DV.

ACKNOWLEDGMENT

Wenjie Zhang is supported by ARC Future Fellowship FT210100303. Lu Qin is supported by ARC FT200100787 and DP210101347.

REFERENCES

- R. M. Karp, "Reducibility among combinatorial problems," in *Complex*ity of computer computations, pp. 85–103, Springer, 1972.
- [2] P. Festa, P. Pardalos, and M. Resende, "Feedback set problems, handbook of combinatorial optimization," *Supplement Vol. A, Kluwer Academic Publishers*, 1999.
- [3] P. B. Galvin, G. Gagne, A. Silberschatz, et al., Operating system concepts. John Wiley & Sons, 2003.
- [4] G. Gardarin and S. Spaccapietra, "Integrity of data bases: A general lockout algorithm with deadlock avoidance.," in *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 395–412, 1976.
- [5] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1-6, pp. 5–35, 1991.
- [6] V. Bafna, P. Berman, and T. Fujito, "A 2-approximation algorithm for the undirected feedback vertex set problem," *SIAM Journal on Discrete Mathematics*, vol. 12, no. 3, pp. 289–297, 1999.
- [7] R. Bar-Yehuda, D. Geiger, J. Naor, and R. M. Roth, "Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference," *SIAM journal on computing*, vol. 27, no. 4, pp. 942–959, 1998.
- [8] G. Even, J. Naor, B. Schieber, and L. Zosin, "Approximating minimum subset feedback sets in undirected graphs with applications," *SIAM Journal on Discrete Mathematics*, vol. 13, no. 2, pp. 255–267, 2000.
- [9] J. Kleinberg and A. Kumar, "Wavelength conversion in optical networks," *Journal of algorithms*, vol. 38, no. 1, pp. 25–50, 2001.
- [10] F. A. Chudak, M. X. Goemans, D. S. Hochbaum, and D. P. Williamson, "A primal-dual interpretation of two 2-approximation algorithms for the feedback vertex set problem in undirected graphs," *Operations Research Letters*, vol. 22, no. 4-5, pp. 111–118, 1998.
- [11] F. Dehne, M. Fellows, M. Langston, F. Rosamond, and K. Stevens, "An o (2 o (k) n 3) fpt algorithm for the undirected feedback vertex set problem," *Theory of Computing Systems*, vol. 41, no. 3, pp. 479–492, 2007.
- [12] R. G. Downey and M. R. Fellows, *Parameterized complexity*. Springer Science & Business Media, 2012.
- [13] J. Guo, R. Niedermeier, and S. Wernicke, "Parameterized complexity of generalized vertex cover problems," in *Workshop on Algorithms and Data Structures*, pp. 36–48, Springer, 2005.
- [14] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Realtime constrained cycle detection in large dynamic graphs," *Proceedings* of the VLDB Endowment, vol. 11, no. 12, pp. 1876–1888, 2018.
- [15] E. Birmelé, R. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto, "Optimal listing of cycles and st-paths in undirected graphs," in *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1884–1896, Society for Industrial and Applied Mathematics, 2013.
- [16] F. Rubin, "Enumerating all simple paths in a graph," *IEEE Transactions on Circuits and Systems*, vol. 25, no. 8, pp. 641–642, 1978.
- [17] A. A. Khan and H. Singh, "Petri net approach to enumerate all simple paths in a graph," *Electronics Letters*, vol. 16, no. 8, pp. 291–292, 1980.
- [18] S. RAI and A. KUMAR, "On path enumeration," *International Journal of Electronics*, vol. 60, no. 3, pp. 421–425, 1986.
 [19] C. Liu, X. He, B. Liang, and Y. Guo, "Detailed placement for pulse
- [19] C. Liu, X. He, B. Liang, and Y. Guo, "Detailed placement for pulse quenching enhancement in anti-radiation combinational circuit design," *Integration*, vol. 62, pp. 182–189, 2018.
- [20] D. Yue, X. Wu, Y. Wang, Y. Li, and C.-H. Chu, "A review of data mining-based financial fraud detection research," in 2007 International Conference on Wireless Communications, Networking and Mobile Computing, pp. 5519–5522, Ieee, 2007.
- [21] Y. Cai and W.-K. Chan, "Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *IEEE Transactions* on Software Engineering, vol. 40, no. 3, pp. 266–281, 2014.
- [22] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang, "Detection of deadlock potentials in multithreaded programs," *IBM Journal of Research and Development*, vol. 54, no. 5, pp. 3–1, 2010.
- [23] N. Alon, "Bipartite subgraphs," *Combinatorica*, vol. 16, no. 3, pp. 301– 311, 1996.
- [24] N. Alon, B. Bollobás, M. Krivelevich, and B. Sudakov, "Maximum cuts and judicious partitions in graphs without short cycles," *Journal* of Combinatorial Theory, Series B, vol. 88, no. 2, pp. 329–346, 2003.
- [25] P. Erdös, T. Gallai, and Z. Tuza, "Covering and independence in triangle structures," *Discrete Mathematics*, vol. 150, no. 1-3, pp. 89–101, 1996.

- [26] M. Krivelevich, "On a conjecture of tuza about packing and covering of triangles," *Discrete Mathematics*, vol. 142, no. 1-3, pp. 281–286, 1995.
 [27] P. A. Pevzner, H. Tang, and G. Tesler, "De novo repeat classification and
- [27] P. A. Pevzner, H. Tang, and G. Tester, "De novo repeat classification and fragment assembly," *Genome research*, vol. 14, no. 9, pp. 1786–1796, 2004.
- [28] G. Kortsarz, M. Langberg, and Z. Nutov, "Approximating maximum subgraphs without short cycles," in *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pp. 118–131, Springer, 2008.
- [29] G. Xia and Y. Zhang, "Kernelization for cycle transversal problems," Discrete Applied Mathematics, vol. 160, no. 7-8, pp. 1224–1231, 2012.
- [30] A. Kuhnle, V. G. Crawford, and M. T. Thai, "Scalable approximations to k-cycle transversal problems on dynamic networks," *Knowledge and Information Systems*, vol. 61, no. 1, pp. 65–84, 2019.
- [31] V. V. Vazirani, Approximation algorithms. Springer Science & Business Media, 2013.
- [32] 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), pp. 1192–1203, IEEE, 2018.
- [33] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng, "Fast: Fpga-based subgraph matching on massive graphs," arXiv preprint arXiv:2102.10768, 2021.
- [34] Y. Peng, X. Lin, Y. Zhang, W. Zhang, and L. Qin, "Answering reachability and k-reach queries on large graphs with label-constraints," *The VLDB Journal*, pp. 1–25, 2021.
- [35] Z. Yuan, Y. Peng, P. Cheng, L. Han, X. Lin, L. Chen, and W. Zhang, "Efficient k-clique listing with set intersection speedup," in *ICDE*, IEEE, 2022.
- [36] Y. Peng, S. Bian, R. Li, S. Wang, and J. X. Yu, "Finding top-r influential communities under aggregation function," in *ICDE*, IEEE, 2022.
- [37] X. Chen, Y. Peng, S. Wang, and J. X. Yu, "Dlcr : Efficient indexing for label-constrained reachability queries on large dynamic graphs," *Proceedings of the VLDB Endowment*, 2022.
- [38] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang, "Huge: An efficient and scalable subgraph enumeration system," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 2049–2062, 2021.
- [39] L. Qin, W. Zhang, Y. Zhang, Y. Peng, H. Kato, W. Wang, and C. Xiao, Software Foundations for Data Interoperability and Large Scale Graph Data Analytics: 4th International Workshop, SFDI 2020, and 2nd International Workshop, LSGDA 2020, Held in Conjunction with VLDB 2020, Tokyo, Japan, September 4, 2020, Proceedings, vol. 1281. Springer Nature, 2020.
- [40] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang, "Pefp: Efficient k-hop constrained s-t simple path enumeration on fpga," in *ICDE*, IEEE, 2021.
- [41] 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*, vol. 62, no. 3, pp. 600–621, 2018.
- [42] D. E. Knuth, The art of computer programming, volume 4A: combinatorial algorithms, part 1. Pearson Education India, 2011.
- [43] M. Nishino, N. Yasuda, S.-i. Minato, and M. Nagata, "Compiling graph substructures into sentential decision diagrams," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- [44] N. Yasuda, T. Sugaya, and S.-I. Minato, "Fast compilation of st paths on a graph for counting and enumeration," in Advanced Methodologies for Bayesian Networks, pp. 129–140, 2017.
- [45] Y. Peng, W. Zhao, W. Zhang, X. Lin, and Y. Zhang, "Dlq: A system for label-constrained reachability queries on dynamic graphs," in *Proceedings of the 230th ACM International Conference on Information & Knowledge Management*, 2021.
- [46] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang, "Answering billionscale label-constrained reachability queries within microsecond," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 812–825, 2020.
- [47] Y. Peng, X. Lin, Y. Zhang, W. Zhang, L. Qin, and J. Zhou, "Efficient hop-constrained s-t simple path enumeration," *The VLDB Journal*, pp. 1– 24, 2021.
- [48] Q. Feng, Y. Peng, W. Zhang, Y. Zhang, and X. Lin, "Towards real-time counting shortest cycles on dynamic graphs: A hub labeling approach," in *ICDE*, IEEE, 2022.
- [49] B. Roberts and D. P. Kroese, "Estimating the number of s-t paths in a graph," J. Graph Algorithms Appl., vol. 11, no. 1, pp. 195–214, 2007.
- [50] K. Hao, L. Yuan, and W. Zhang, "Distributed hop-constrained s-t simple path enumeration at billion scale," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 169–182, 2021.

- [51] X. Li, K. Hao, Z. Yang, X. Cao, and W. Zhang, "Hop-constrained s-t simple path enumeration in large uncertain graphs," in *Databases Theory* and Applications - 33rd Australasian Database Conference, ADC 2022, Sydney, NSW, Australia, September 2-4, 2022, Proceedings (W. Hua, H. Wang, and L. Li, eds.), vol. 13459 of Lecture Notes in Computer Science, pp. 115–127, Springer, 2022.
- [52] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Towards bridging theory and practice: hop-constrained st simple path enumeration," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 463–476, 2019.
- [53] R. Rizzi, G. Sacomoto, and M.-F. Sagot, "Efficiently listing bounded length st-paths," in *International Workshop on Combinatorial Algorithms*, pp. 318–329, Springer, 2014.
- [54] R. Grossi, A. Marino, and L. Versari, "Efficient algorithms for listing k disjoint st-paths in graphs," in *Latin American Symposium on Theoretical Informatics*, pp. 544–557, Springer, 2018.
- [55] S. Funke, A. Nusser, and S. Storandt, "On k-path covers and their

applications," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 893–902, 2014.

- [56] B. Brešar, F. Kardoš, J. Katrenič, and G. Semanišin, "Minimum k-path vertex cover," *Discrete Applied Mathematics*, vol. 159, no. 12, pp. 1189– 1195, 2011.
- [57] V. Guruswami and E. Lee, "Inapproximability of feedback vertex set for bounded length cycles.," in *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 21, p. 2, 2014.
- [58] S. Sun, Y. Chen, B. He, and B. Hooi, "Pathenum: Towards realtime hop-constrained st path enumeration," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 1758–1770, 2021.
- [59] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," ACM Transactions on Intelligent Systems and Technology (TIST), vol. 8, no. 1, p. 1, 2016.
- [60] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, pp. 1343–1350, ACM, 2013.