Towards Real-Time Counting Shortest Cycles on Dynamic Graphs: A Hub Labeling Approach

Qingshuai Feng[†], You Peng[†*], Wenjie Zhang[†], Ying Zhang[§], Xuemin Lin[†]

[†]*The University of New South Wales;* [§]*University of Technology Sydney*
q.feng.1@student.unsw.edu.au; unswpy@gmail.com;
{zhangw,lxue}@cse.unsw.edu.au; ying.zhang@uts.edu.au

Abstract—With the ever-increasing prevalence of graph data in a wide spectrum of applications, it becomes essential to analyze structural trends in dynamic graphs on a continual basis. The shortest cycle is a fundamental pattern in graph analytics. In this paper, we investigate the problem of shortest cycle counting for a given vertex in dynamic graphs in light of its applicability to problems such as fraud detection. To address such queries efficiently, we propose a 2-hop labeling based algorithm called <u>C</u>ounting <u>S</u>hortest <u>C</u>ycle (CSC for short). Additionally, techniques for dynamically updating the CSC index are explored. Comprehensive experiments are conducted to demonstrate the efficiency and effectiveness of our method. In particular, CSC enables query evaluation in a few hundreds of microseconds for graphs with millions of edges, and improves query efficiency by two orders of magnitude when compared to the baseline solutions. Also, the update algorithm could efficiently cope with edge insertions (deletions).

I. INTRODUCTION

With the rapid development of information technology, a growing number of applications represent data as graphs [1]–[6]. The graph structure naturally encodes complex relationships among entities in real-world networks such as social networks, e-commerce networks, and electronic payments networks. Sophisticated analytics over such graphs provides insights into the underlying dataset and interactions [7]–[9]. Such networks often include millions of edges and vertices. Additionally, the structural changes to these networks are constant in nature, which makes them extremely dynamic. At scale, it is imperative and challenging to support real-time analytics on rapidly changing structural patterns in dynamic graphs [10], [11].

Cycles are fundamental to graph analytics. A cycle is a path with at least 3 vertices and returns to its starting vertex. A cycle captures the circular structural pattern from the starting vertex to itself and is an informative indicator for graph analytics. In light of this, cycle detection has been extensively investigated in the literature for both static and dynamic graphs [11], [12]. The shortest cycle refers to the cycle in a graph with the shortest length (i.e., the minimum number of edges) and the length is also called *girth* of the graph. The shortest cycle is often employed in graph structure analysis, for example, when solving the graph coloring problem [13], [14].

The shortest cycles across a vertex v denotes the cycles through v with the shortest length [12], [15]. For a particular vertex, the shortest cycles provide the quickest feedback routes originated at the vertex [16]. The lengths of the shortest cycles have been studied to determine the frequencies of
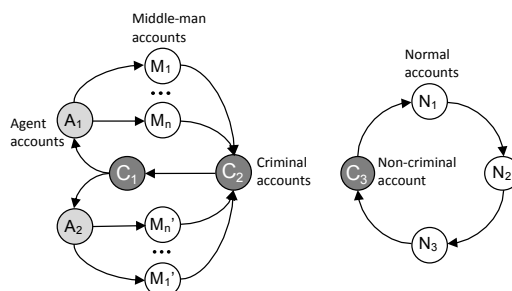
Fig. 1. A Graph of a Money-Laundering Example.

oscillations in neural circuits [17]. The distribution of shortest cycle lengths supports the structure analysis on chemical networks and biological networks [18], [19], as well as feedback processes [20] and synchronization [21] in various networks. While previous works mostly focus on computing the length of shortest cycles, in many real-world graphs, the diameter of the graphs tends to be small due to the small-world phenomenon. This leads to the fact that the shortest cycles of many vertices share the same length. For instance, Figure 1 depicts a transaction network involved in money laundering. The shortest cycle length through both vertex C_1 and vertex C_3 is 4. Nonetheless, there are significantly more shortest cycles via C_1. The number of shortest cycles via a vertex becomes a more informative metric for identifying suspicious users in transaction networks. Motivated by this, we investigate the problem of calculating shortest cycles for any given vertex in a graph.

Applications. Shortest cycle counting is required in a broad variety of applications. Two example applications are given below.

Application 1: Fraud Detection. Figure 1 illustrates an example of money-laundering crime. Dark gray vertices C_1 and C_2 are both criminal accounts. Other vertices represent intermediary accounts (white nodes) or agents (light gray nodes) which assist the criminals. Directed edges depict forged transactions or other money laundering activities. The shortest cycles reflect criminals' most efficient ways for money laundering. As a result, a high number of such cycles travel through the criminal accounts. For example, a cycle, starting from and ending with C_1 via A_1 M_1 C_2, demonstrates such a route. The more such cycles that pass through a vertex, the more probable it is that the individual will engage in money-laundering activities. A pre-screening criterion for offenders might include a specified number of shortest cycles [11], [22]. Additionally, such data can also be utilized to detect fraudulent transactions

on an e-commerce platform [11], [23]. [24] develops an expert system for automobile insurance fraud detection. A collision graph is constructed by considering automobiles as vertices and edges between two vehicles as collisions. Fraudulent components share some structural characteristics, one of which is the occurrence of short cycles. [25] studies the similar problem. Their experiments demonstrate that detecting cycles is more effective in finding potential fraudulent groups than using other methods, such as leading eigenvector. [26] applies the cycle detection problem to a fraud detection system of financial institutions and demonstrates its effectiveness.

Application 2: File Sharing Optimization. Consider another use case in peer-to-peer file-sharing networks (e.g., Gnutella) [27]. The vertices in the network represent hosts, while edges indicate the interactions, such as file request or transfer, between the hosts. A file-sharing cycle via a host signals the end of the file-sharing activity. For instance, host A sends a file request, which is propagated until it reaches host B. B is the location of the requested file and sends the file to host A. Generally, the most efficient routing is chosen with the minimum hop-count. The number of shortest cycles through host A along with the length might be used to determine the host's location. For instance, if host A has longer and more shortest cycles, a proxy server is probably required to decrease the transmission costs. [28] introduces an index-server optimization problem for P2P file sharing. They analyzed a flooding scheme and two index-server schemes for history and latest queries. In this problem, they need to set the index-server, while a machine with a high number of shortest cycles is preferred due to 1) these systems need to be failure-tolerant and 2) a needed file is easy to locate. The shortest cycles naturally satisfied these two requirements.

In each of the aforementioned applications, the networks are extremely dynamic in nature. For instance, in an e-commerce network, fraud accounts often initiate new transactions and transfer funds to others. New file requests and file transfers often occur in a file-sharing network. An update will be reflected in the graph as an edge insertion or deletion and will impact the query results for many vertices in the graph. In many scenarios, a static graph or a snapshot of a dynamic graph is insufficient, since continuous monitoring of shortest cycles numbers is needed. As a consequence, the dynamic issue is essential for shortest cycle counting.

Challenges and Contributions. Numerous applications of the shortest cycle counting, particularly in online situations, demand real-time response to the query [11], [23]. Thus, it is critical to develop a real-time algorithm capable of counting the shortest cycles. A straightforward solution is to calculate the number of shortest cycles for each vertex in advance and record the values. Then, any query can be answered with $O(1)$ time complexity. Nevertheless, such a simple approach cannot handle dynamic graphs well since it requires to re-compute the shortest cycles for all vertices regarding graph updates. This is because of the lack of awareness for distance information among vertices within the graph. Hence, it is challenging to build an index that supports both real-time query answering and fast updates in the face of dynamic changes.

We address the aforementioned issues in this paper and offer the following contributions.

(1) Hub Labeling algorithm (CSC) for Counting Shortest Cycles. We devise a new and effective bipartite conversion-based technique for reshaping the initial graph and then constructing its hub labeling index CSC to enable real-time shortest cycle counting. Efficient pruning strategies are developed to expedite the index building process. An index merge mechanism is applied to aggregate related label entries and minimize the label size. Even if the bipartite conversion doubles the number of vertices in the reshaped graph, the new index remains a similar size compared with the baseline.

(2) Dynamic Maintenance of CSC. We provide the first incremental (decremental) updating algorithm for an added (removed) edge in order to maintain our CSC index. Rather than rebuilding the whole index, only the portions impacted by the new edge are updated, substantially lowering the cost of maintenance.

(3) Comprehensive Experimental Evaluation. We perform experiments on nine graphs to validate the effectiveness and efficiency of our algorithms. The experimental results demonstrate that CSC is comparable to the baseline in terms of index construction time and index size but is up to two orders of magnitude faster in terms of query processing. Also, the update algorithm could efficiently cope with edge insertions (deletions).

Roadmap. The rest of the paper is organized as follows. Section II introduces some preliminaries and Section III introduces baselines. Our 2-hop based method is proposed in Section IV. Section V investigates the index maintenance for edge insertion, followed by empirical studies in Section VI. Section VII surveys important related work. Section VIII concludes the paper.

II. PRELIMINARIES

TABLE I
NOTATIONS

Notations	Description
$G = (V, E)$	directed graph with vertex set V and edge set E
SP(s, t)	all the shortest paths from s to t
SPV(s, t)	all the corresponding vertices in SP(s, t)
SPCnt(s, t)	the number of shortest paths from s to t
SCCnt(v)	the number of shortest cycles through v
G_0	the original graph
G_+/G_-	the graph after an edge insertion/deletion

A. Preliminaries

$G = (V, E)$ denotes a directed graph where V is the set of its vertices and $E \subseteq V \times V$ is the set of edges. $e(v, u)$ represents a directed edge connecting the vertex v to vertex u. Let n and m represent the number of vertices and edges, respectively. The degree of vertex v is defined as degree(v), which is the sum of its in- and out-degrees. We use $nbr_{out}(v)$ to refer to its out neighbors or successors and $nbr_{in}(v)$ to refer to its in neighbors or ancestors. A path p from vertex s to vertex t is a sequence of vertices in the form of $< s = v_0, v_2, ..., v_k = t >$ where for every $i \in [1, k], e(v_{i-1}, v_i) \in E$.

$p(u, v)$ denotes a path from vertex u to v. The length of a path p, indicated by $len(p)$, is the total number of its edges. A path is considered *simple* if it has no repeating vertices or edges. A path from vertex s to vertex t is said to be shortest if its length is equal to or less than the length of any other path from s to t. The length of the shortest path from s to t is represented by $sd(s, t)$, and one such path is denoted by $sp(s, t)$. $SP(s, t)$ denotes the collection of all such shortest paths. $SPV(s, t)$ is the set of all vertices that correspond to $SP(s, t)$. Let $SPCnt(s, t)$ represent the number of shortest paths connecting s and t, i.e., $|SP(s, t)|$.

Similarly, we present the definitions for cycles. A *cycle* is a path p with $v_0 = v_k$ and $len(p) \geq 3$. A cycle is said to be *simple* if it has no repeats of vertices or edges other than the starting and ending vertex. The length of a cycle is equal to its number of edges. A cycle through a vertex v is said to be shortest if its length is not larger than any other cycle through v. The number of shortest cycles through v is denoted by $SCCnt(v)$.

To tackle the dynamic graph, we examine only edge updates in the paper, since the insertion or deletion of a vertex can be represented by a series of edge insertions or deletions. We refer to the original graph as G_0, and use G_+ (G_-) to denote the graph after an edge insertion (deletion). Table I summarizes the key mathematical notations used throughout this paper.

Problem Statement. Given a graph G and a query vertex v, the shortest cycle counting query, denoted by $SCCnt(v)$, returns the number of shortest cycles through vertex v.

Example 1. *There are three shortest cycles in Figure 2 with length 6 through v_7. Thus, $SCCnt(v_7)=3$.*

B. Hub Labeling for Shortest Path Counting Queries

Next, we present the hub labeling method for the shortest path counting between vertices s and t, $SPCnt(s, t)$. In a directed graph, the shortest path counting between vertices s and t seeks to determine the total number of all the shortest paths from s to t. [29] proposed a 2-hop labeling scheme and index construction algorithm to build the index efficiently and enabling real-time shortest path counting queries. The hub labeling scheme supports the cover constraint, <u>E</u>xact <u>S</u>hortest <u>P</u>ath <u>C</u>overing (ESPC), which implies that it not only encodes the shortest distance between two vertices but also ensures that such shortest paths are correctly counted. HP-SPC is their proposed algorithm to construct the SPC label index that satisfies ESPC.

Formally, given a directed graph G, HP-SPC assigns each vertex $v \in G$ an in-label $L_{in}(v)$ and an out-label $L_{out}(v)$, which are composed of entries of the form $(w, sd(v, w), \theta_{v,w})$. The shortest distance between v and w is denoted by $sd(v, w)$, and the number of shortest paths between v and w is denoted by $\theta_{v,w}$. If $w \in L_{in}(v)$ or $w \in L_{out}(v)$, we say that w is a hub of v. Essentially, the in-label $L_{in}(v)$ keeps track of the distance and counting information from its hubs to itself, while the out-label $L_{out}(v)$ records distance and counting information from v to its hubs. HP-SPC adheres to the cover constraint, which states that for any given starting vertex s and ending vertex t, there exists a vertex $w \in L_{out}(s) \cap L_{in}(t)$ that lies on the shortest path from s to t. Additionally, HP-SPC guarantees

the correctness of shortest path counting by including the shortest path from s to t via a hub vertex once during the label construction. $SPCnt(s, t)$ is evaluated by scanning the $L_{out}(s)$ and $L_{in}(t)$ for the shortest distance via common hubs and adding the multiplication of the corresponding counting. Equation (1) identifies all common hubs (on shortest paths) from $L_{out}(s)$ and $L_{in}(t)$. Equation (2) determines the result of $SPCnt(s, t)$.

$$H = \{h| \underset{h \in L_{out}(s) \cap L_{in}(t)}{\arg\min} \{sd(s, h) + sd(h, t)\}\} \quad (1)$$

$$SPCnt(s, t) = \sum_{h \in H} \theta(s, h) \cdot \theta(h, t) \quad (2)$$

Example 2. *Figure 2 depicts a directed graph with 10 vertices, and Table II provides its hub labeling index for SPCnt queries. We use $SPCnt(v_{10}, v_8)$ as an example to determine the shortest paths counting from v_{10} to v_8. Two common hubs $\{v_1, v_7\}$ are discovered by scanning $L_{out}(v_{10})$ and $L_{in}(v_8)$. The shortest distance through v_1 is $1 + 3 = 4$, while the counting is $1 \cdot 2 = 2$; The shortest distance via v_7 is $3 + 1 = 4$, and the counting is $1 \cdot 1 = 1$. Thus, the number of shortest paths from v_{10} to v_8 is $2 + 1 = 3$ with length 4.*

III. BASELINES

In this section, we introduce two baseline solutions for the shortest cycle counting problem in this paper.

A. HP-SPC for SCCnt by Neighborhood Information

By selecting the identical beginning and ending vertex, an obvious solution for shortest cycle counting is to utilize the existing hub-labeling techniques HP-SPC for shortest path counting. To compute $SCCnt(v_q)$ through query vertex v_q, we resort to HP-SPC based hub labeling described in [29] by specifying both starting and ending vertex to be v_q, i.e., $SPCnt(v_q, v_q)$. Nevertheless, this will lead to erroneous query results since the shortest distance is always 0, as determined by identifying a self-loop of v_q in (1). For instance, if we aim to count the number of shortest cycles via v_1 in Figure 2, $SPCnt(v_1, v_1)$ will search up $L_{in}(v_1)$ and $L_{out}(v_1)$ of the hub labeling in Table II, and return 0.

To remedy this, we convert the shortest cycle counting query to the shortest path counting query between query vertex v_q and the in-neighbors (or out-neighbors) of v_q. Formally, to compute $SCCnt(v_q)$, we compute the SPCnt from v_q to each of the in-neighbor w of v_q, or from each out-neighbor to v_q. To reduce query costs, we choose out-neighbors of v_q when $|nbr_{out}(v_q)| < |nbr_{in}(v_q)|$, and in-neighbors of v_q otherwise. Equation (3) and (4) demonstrate the evaluation of $SCCnt(v_q)$ when $|nbr_{out}(v_q)| < |nbr_{in}(v_q)|$. The first step is to get all vertices with the shortest distance to v_q using Equation (3). Clearly, the length of the shortest cycles via v_q is increasing the shortest distance in (3) by 1. The number of shortest cycles passing through v_q is derived by summing up the number of shortest paths between v_q and each of the vertices acquired in the first step. Notably, if there is no path from v_q to its

Fig. 2. A Directed Graph.

TABLE II
SHORTEST PATH COUNTING LABELS OF FIG. 2
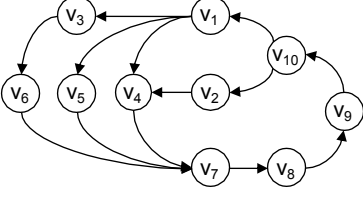
Vertex	$L_{in}(\cdot)$	$L_{out}(\cdot)$
v_1	$(v_1, 0, 1)$	$(v_1, 0, 1)$
v_2	$(v_1, 6, 2)$ $(v_7, 4, 1)$ $(v_{10}, 1, 1)$ $(v_2, 0, 1)$	$(v_1, 6, 1)$ $(v_7, 2, 1)$ $(v_4, 1, 1)$ $(v_2, 0, 1)$
v_3	$(v_1, 1, 1)$ $(v_3, 0, 1)$	$(v_1, 6, 1)$ $(v_7, 2, 1)$ $(v_3, 0, 1)$
v_4	$(v_1, 1, 1)$ $(v_7, 5, 1)$ $(v_4, 0, 1)$	$(v_1, 5, 1)$ $(v_7, 1, 1)$ $(v_4, 0, 1)$
v_5	$(v_1, 1, 1)$ $(v_5, 0, 1)$	$(v_1, 5, 1)$ $(v_7, 1, 1)$ $(v_5, 0, 1)$
v_6	$(v_1, 2, 1)$ $(v_3, 1, 1)$ $(v_6, 0, 1)$	$(v_1, 5, 1)$ $(v_7, 1, 1)$ $(v_6, 0, 1)$
v_7	$(v_1, 2, 2)$ $(v_7, 0, 1)$	$(v_1, 4, 1)$ $(v_7, 0, 1)$
v_8	$(v_1, 3, 2)$ $(v_7, 1, 1)$ $(v_8, 0, 1)$	$(v_1, 3, 1)$ $(v_7, 5, 1)$ $(v_4, 4, 1)$ $(v_{10}, 2, 1)$ $(v_8, 0, 1)$
v_9	$(v_1, 4, 2)$ $(v_7, 2, 1)$ $(v_8, 1, 1)$ $(v_9, 0, 1)$	$(v_1, 2, 1)$ $(v_7, 4, 1)$ $(v_4, 3, 1)$ $(v_{10}, 1, 1)$ $(v_9, 0, 1)$
v_{10}	$(v_1, 5, 2)$ $(v_7, 3, 1)$ $(v_{10}, 0, 1)$	$(v_1, 1, 1)$ $(v_7, 3, 1)$ $(v_4, 2, 1)$ $(v_{10}, 0, 1)$

neighbors, Equation (3) returns an empty set, indicating that there is no cycle via v_q.

$$W = \{w| \underset{w \in nbr_{out}(v_q)}{\operatorname{argmin}} sd(w, v_q)\} \qquad (3)$$

$$SCCnt(v_q) = \sum_{w \in W} SPCnt(w, v_q) \qquad (4)$$

Example 3. *Consider the example of SCCnt(v_7). v_7 has three in-neighbors $\{v_4, v_5, v_6\}$. According to the index in Table II, SPCnt(v_7, v_4) = 2 and sd(v_7, v_4) = 5; SPCnt(v_7, v_5) = 1 and sd(v_7, v_5) = 5; SPCnt(v_7, v_6) = 1 and sd(v_7, v_6) = 6. Thus, the shortest cycles via v_7 passing through v_4 and v_5. SCCnt(v_7) is 2 + 1 = 3.*

B. Breadth-First Search (BFS) Based Shortest Cycle Counting

The second baseline solution applies the Breadth-First Search (BFS) from the query vertex v_q. In Algorithm 1, we record the shortest distance from v_q ($D[\cdot]$) and the counting of the shortest paths ($C[\cdot]$) for each accessed vertex throughout the BFS. Lines 11-16 compare the recorded distance of accessed vertex w_n and the derived distance from its ancestor w. Particularly, if the recorded distance is larger, $D[w_n]$ will be adjusted to reflect $D[w]$ and will inherit the shortest path counting from w (lines 12-14). If the recorded distance equals the distance from w, only the shortest path counting will be updated (lines 15-16). Once v_q is popped out from the queue, all of its shortest cycles are encountered. Otherwise, if v_q is never visited until the queue is empty, then it indicates that no cycle through v_q occurs. Clearly, the time complexity of BFS based solution is $O(m + n)$ and the space complexity is also $O(m + n)$.

IV. BIPARTITE HUB LABELING FOR SHORTEST CYCLE COUNTING

This section introduces our bipartite conversion based hub labeling scheme, then describes labeling computation and processing shortest cycle counting based on the labeling.

A. Bipartite Hub Labeling Scheme

We seek to develop a hub labeling scheme for shortest cycle counting by effectively implementing hub labeling for shortest path counting. To this end, the labeling must adhere to the cover constraint of hub labeling for shortest path counting. The following constraint is imposed in [29] to ensure correctly maintaining the counting of shortest paths.

Algorithm 1: BFS-CYCLE(G, v_q)

1 **for each** $v \in V$ **do**
2      $D[v] \leftarrow \infty$; $C[v] \leftarrow 0$;
3 $Q \leftarrow \emptyset$;
4 **for each** $u \in nbr_{out}(v_q)$ **do**
5      $D[u] \leftarrow 1$; $C[u] \leftarrow 1$;
6   Q.enqueue(u);
7 **while** Q is not empty **do**
8      $w \leftarrow Q$.dequeue();
9   **if** $w = v_q$ **then**
10   **return** $(D[v_q], C[v_q])$;
11   **for each** $w_n \in nbr_{out}(w)$ **do**
12   **if** $D[w_n] > D[w] + 1$ **then**
13             $D[w_n] \leftarrow D[w] + 1$; $C[w_n] \leftarrow C[w]$;
14   Q.enqueue(w_n);
15   **else if** $D[w_n] = D[w] + 1$ **then**
16             $C[w_n] \leftarrow C[w_n] + C[w]$;
17 **return** $(\infty, 0)$;

Cover Constraint. Given a total ordering $\prec$ over all the vertices in graph G where for any two vertices (v, w), $v \prec w$ indicates that v has a rank higher than w. A label entry (v, d, c) is added to the in-label of vertex w $L_{in}(w)$ if there exists at least one shortest path $sp(v, w)$ between v and w, where v has the highest rank along $sp(v, w)$. Here, d represents the shortest distance $sd(v, w)$, and c is the number of all such shortest paths. Notably, if $c = |SP(v, w)|$, namely, c is the number of all shortest paths between v and w, (v, d, c) is referred to as a **canonical label** of $L_{in}(w)$. Otherwise, if $c < |SP(v, w)|$, namely only a proper subset of $SP(v, w)$ is counted, (v, d, c) it is a **non-canonical label**.

Example 4. *In Figure 2, assume the ordering of vertices follows the degree order: $v_1 \prec v_7 \prec v_4 \prec v_{10} \prec v_2 \prec v_3 \prec v_5 \prec v_6 \prec v_8 \prec v_9$. There are two shortest paths in the reverse graph (a reverse graph is obtained by reversing the orientations of all edges in the graph) from v_4 to v_{10} with length 2. One of the paths v_4, v_1, v_{10} passes through v_1 which has a higher rank than v_4. Thus, the count c is 1 rather than 2 for $(v_4, 2, c)$ in $L_{out}(v_{10})$. This label entry is an example of a non-canonical one.*
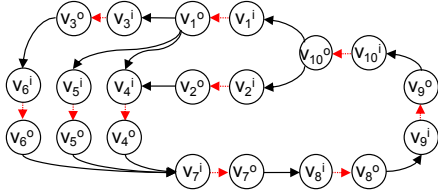
Fig. 3. Bipartite Conversion of Figure 2

B. Bipartite Conversion

It is insufficient to provide shortest cycle counting just by following the cover constraint of shortest path counting. As shown in Section III-A, current hub labeling techniques are incapable of supporting shortest cycle counting. We offer a new bipartite conversion approach to circumvent this limitation. Given a directed graph G, the following procedure is used to build its bipartite graph G_b conversion.

Algorithm 2 illustrates the conversion. From line 2 to 6, each vertex $v \in G$ is decomposed into a pair of *couple vertices* (v^i, v^o) where v^i represents the incoming vertex of v, and v^o denotes the outgoing vertex of v, along with an edge $e(v^i, v^o)$. V_{in} (V_{out}) denotes the set of all incoming (outgoing) vertices. The edge $e(v, w)$ from the original graph G is replaced by an edge $e(v^o, w^i)$ in the new graph G_b. G_b is a bipartite graph with two vertex sets V_{in} and V_{out}. G_b has a vertex and edge count of $2 \cdot n$ and $n + m$, respectively. The vertices in V_{in} contain all the in-edges of G, and the vertices in V_{out} contain all the out-edges. Figure 3 illustrates the bipartite conversion of the graph in Figure 2.

Algorithm 2: BI-G(G)

1 $V_{in} \leftarrow \emptyset; V_{out} \leftarrow \emptyset; E_b \leftarrow \emptyset;$
2 **for each** $v \in V$ **do**
3 $\quad \{v^i, v^o\} \leftarrow v;$
4 $\quad V_{in} \leftarrow V_{in} \cup \{v^i\};$
5 $\quad V_{out} \leftarrow V_{out} \cup \{v^o\};$
6 $\quad E_b \leftarrow E_b \cup \{e(v^i, v^o)\};$
7 **for each** $e(v, w) \in E$ **do**
8 $\quad E_b \leftarrow E_b \cup e(v^o, w^i);$
9 **return** $G_b = (V_{in}, V_{out}, E_b);$

Couple-Vertex Skipping. The consecutive order of each pair of couple vertices allows the correct hub labeling by avoiding some computation associated with index construction. According to the cover constraint, only higher-ranked vertices are eligible to be the hub of lower-ranked ones. Thus, $(v^i, 1, 1)$ must be a label entry of $L_{in}(v^o)$. Any vertex that is qualified to serve as an in-label hub for v^i should also be an in-label hub of v^o. When an in-label hub is identified for v^i, we can simply increase the distance by 1 and add the label to v^o. As a result, we can exclude v^o from the in-label construction. Regarding out-labels, the graph is traversed in a reverse direction. Similarly, if an out-label hub is identified for v^o, it will be immediately added to v^i, obviating the need for v^i's out-label construction. Additionally, if both coupled vertices v^i and v^o serve as hubs for a vertex w, it is acceptable to retain just the label entry of the higher-ranked hub v^i, since v^o contains redundant information from the pair.

C. Bipartite Hub Labeling Construction

The index construction algorithm for in-label generation is illustrated in Algorithm 3, which employs the couple-vertex skipping technique. The algorithm begins with the vertex v and pushes it to the vertices that share v as a hub. It explores from higher to lower rank in accordance with the vertex ordering.

Algorithm 3: CSC($G_b, \bar{G}_b$)

1 **for each** $v \in V_{in} \cup V_{out}$ **do**
2 $\quad L_{in}^c(v) \leftarrow \emptyset; L_{in}^{nc}(v) \leftarrow \emptyset;$
3 $\quad L_{out}^c(v) \leftarrow \emptyset; L_{out}^{nc}(v) \leftarrow \emptyset;$
4 $\quad D[v] \leftarrow \infty; C[v] \leftarrow 0;$
5 **for each** $v \in V_{in} \cup V_{out}$ in descending order **do**
6 $\quad$ **if** $v \in V_{out}$ **then**
7 $\quad\quad$ append $(v, 0, 1)$ to $L_{in}^c(v)$; append $(v, 0, 1)$ to $L_{out}^c(v);$
8 $\quad\quad$ **continue**;
9 $\quad$ Queue $Q \leftarrow \emptyset; Q.\text{enqueue}(v);$
10 $\quad D[v] \leftarrow 0; C[v] \leftarrow 1; V_{is} \leftarrow \{v\};$
11 $\quad$ **while** $Q \neq \emptyset$ **do**
12 $\quad\quad w \leftarrow Q.\text{dequeue}();$
13 $\quad\quad d \leftarrow min_{u \in L_{out}^c(v) \cap L_{in}^c(w)} sd(v, u) + sd(u, w);$
14 $\quad\quad$ **if** $d < D[w]$ **then**
15 $\quad\quad\quad$ **continue**;
16 $\quad\quad$ INSERTLABEL($v, d, w, D[w], C[w]$);
17 $\quad\quad w' \leftarrow$ couple of w; $V_{is} \leftarrow V_{is} \cup \{w'\};$
18 $\quad\quad D[w'] \leftarrow D[w] + 1; C[w'] \leftarrow C[w];$
19 $\quad\quad$ **for each** $w_n \in nbr_{out}(w')$ **do**
20 $\quad\quad\quad$ **if** $D[w_n] = \infty \wedge v \prec w_n$ **then**
21 $\quad\quad\quad\quad D[w_n] \leftarrow D[w'] + 1; C[w_n] \leftarrow C[w'];$
22 $\quad\quad\quad\quad Q.\text{enqueue}(w_n); V_{is} \leftarrow V_{is} \cup \{w_n\};$
23 $\quad\quad\quad$ **else if** $D[w_n] = D[w'] + 1$ **then**
24 $\quad\quad\quad\quad C[w_n] \leftarrow C[w_n] + C[w'];$
25 $\quad$ **for each** $v \in V_{is}$ **do**
26 $\quad\quad D[v] \leftarrow \infty; C[v] \leftarrow 0;$
27 $\quad$ /* Out-labels Generation */
28 **for each** $v \in V_{in} \cup V_{out}$ **do**
29 $\quad L_{in}(v) \leftarrow L_{in}^c(v) \cup L_{in}^{nc}(v);$
30 $\quad L_{out}(v) \leftarrow L_{out}^c(v) \cup L_{out}^{nc}(v);$

Initialization. To begin, the canonical and non-canonical in(out)-label sets $(L_{in}^c(\cdot), L_{in}^{nc}(\cdot), L_{out}^c(\cdot), L_{out}^{nc}(\cdot))$ for each vertex are initialized (lines 2-3). Arrays $D[\cdot]$ and $C[\cdot]$ store tentative distances and counting of such distances (line 4). Then, for each vertex v in descending order of rank, a breadth-first search is performed to identify the vertices w that have v as a hub in $L_{in}(w)$ or $L_{out}(w)$. If v comes from V_{out}, we can safely skip the current loop and add the label entries for itself (lines 6-8). If v is from V_{in}, the label construction process begins with a queue Q containing the current hub. Vis maintains the set of visited vertices.

Label Generation. The following procedure generates all label entries for the current hub v (lines 12-16). From v, we visit vertex w in a BFS fashion. We calculate the distance from
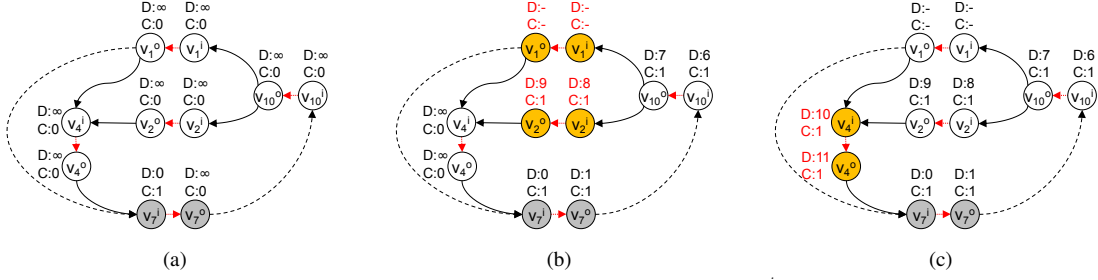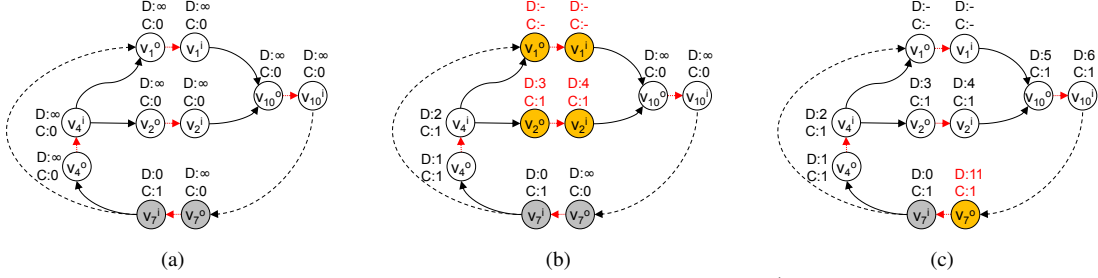
Fig. 4. In-Labels Construction Example (Hub $v_7{}^i$).

Fig. 5. Out-Labels Construction Example (Hub $v_7{}^i$).

v to w using the hub labels prior to v. This is the shortest distance d in line 13 from v to w through a higher-ranked vertex. If d is less than the tentative distance $D[w]$, the BFS is superfluous, since v is not the highest-ranked vertex along any shortest path from v to w (lines 14-15). Otherwise, a new label of w with hub v is generated by calling Algorithm 4. If d equals to the tentative distance D, then v has the highest rank along **some** of the shortest paths from v to w. A non-canonical label is generated in this scenario (lines 2-3). According to the Couple-Vertex Skipping, w', the couple of w, also has a new non-canonical label with the distance of $D + 1$ (line 4). If $d > D$, then v has the highest rank along **all** the shortest paths from v to w. In this case, a canonical label is generated for both w and its couple w' (lines 5-7).

Algorithm 4: INSERTLABEL(v, d, w, D, C)

---

1   $w' \leftarrow$ couple of w;
2   **if** $d = D$ **then**
3   append (v, D, C) to $L_{in}^{nc}(w)$;
4   append $(v, D + 1, C)$ to $L_{in}^{nc}(w')$;
5   **else if** $d > D$ **then**
6   append (v, D, C) to $L_{in}^{c}(w)$;
7   append $(v, D + 1, C)$ to $L_{in}^{c}(w')$;

Update Tentative Distance and Counting. $w \in V_{in}$ is constantly established during the in-label generation process. Due to the fact that the first vertex in Q is from V_{in}, then we have its couple w' in V_{out} (line 17). By using the couple-vertex skipping technique, w is skipped. We continue to check each neighbor w_n from $nbr_{out}(w')$, where w_n must be from V_{in} (lines 19-24). If w_n is not visited and has a lower rank than the current hub v, its tentative distance and counting are updated. After that, it is pushed into Q (lines 20-22). Thus, only those vertices from V_{in} are eligible to be included in Q. Otherwise, if the next-step distance to w_n is identical to its

tentative distance, the counting is accumulated due to the new shortest paths (lines 23-24). Finally, the label of each vertex is the concentration of the canonical and non-canonical labels (lines 28-30).

The process for out-label generation is similar to the procedure described in lines 9-26 but in a reverse direction. The primary distinctions are as follows: (1) The distance calculated using an existing index should be the distance from w to v (line 13); (2) Replace nbr_{out} with nbr_{in} (line 19); (3) The first time w popped from Q is v (line 12), only $(v, 0, 1)$ should be inserted into $L_{out}^{c}(v)$ in this loop. Then skip lines 17-18 and verify each of v's in-neighbor. From the second loop, w should always be from V_{out}; (4) In Algorithm 4, label entries should be inserted into out-labels. If w is the couple of v, only insert label entry into $L_{out}(w)$ and prune at this point. The Algorithm CSC takes the graph G_b and reverse graph $\bar{G}_b$ and generates the in-label $L_{in}(\cdot)$ and out-label $L_{out}(\cdot)$ set for each vertex v.

Example 5. *Figure 4 shows three snapshots during in-label construction for hub $v_7{}^i$. These snapshots that correspond to Figure 3, except that v_3, v_5, v_6, v_8 and v_9 are omitted for ease. The initial stages are shown in Figure 4(a). Except for $v_7{}^i$'s counting, all tentative distance and counting are set to ∞ and 0, respectively. After reaching $v_{10}{}^o$ with distance 7 and counting 1, $v_1{}^i$ and $v_2{}^i$ will be processed the next round in Figure 4(b). Due to the fact that $v_1{}^i \prec v_7{}^i$, the BFS at v_1 is trimmed, and the algorithm proceeds to $v_2{}^i$. All the in-label entries with hub $v_7{}^i$ are canonical prior to $v_4{}^i$. In Figure 4(c), we can see that when we reach $v_4{}^i$, $sd(v_7{}^i, v_4{}^i)$ is 10 via hub $v_1{}^i$ which is the current tentative distance. Thus, $(v_7{}^i, 10, 1)$ is inserted into $L_{in}^{nc}(v_4{}^i)$ as a non-canonical label entry.*

As with Example 5, Figure 5 illustrates the generation of out-labels with hub $v_7{}^i$.

D. Query Evaluation

Counting the shortest cycles through v in the original graph G_0 corresponds to counting the shortest paths from v^o to v^i in the transformed bipartite graph G_b. SCCnt(v) can be assessed using the index constructed above as SPCnt(v^o, v^i). Note that the distance d in the result represents the shortest distance from v^o to v^i in G_b. Due to the presence of an edge $e(v^i, v^o)$ and the doubling of all distances in G_b, the shortest cycle length in G_0 is $(d + 1) / 2$.

Example 6. *Consider SCCnt(v_7) in comparison to Example 4. Table III shows the in-labels for v_7^i and out-labels for v_7^o. The shortest distance from v_7^o to v_7^i through hub v_7^i is 4 + 7 = 11 and the associated path counting is $2 \cdot 1 = 2$. The shortest path counting with the same length via hub v_7^i is $1 \cdot 1 = 1$. Thus, SCCnt(v_7) is 2 + 1 = 3. The shortest cycle length is (11 + 1) / 2 = 6. The evaluation of SCCnt(v) is evaluated only in terms of $L_{in}(v_7^i)$ and $L_{out}(v_7^o)$.*

TABLE III
IN-LABELS OF v_7^i AND OUT-LABELS OF v_7^o

$L_{in}(v_7^i)$	$(v_1^i, 4, 2)$ $(v_7^i, 0, 1)$
$L_{out}(v_7^o)$	$(v_1^i, 7, 1)$ $(v_7^i, 11, 1)$ $(v_7^o, 0, 1)$

E. Index Reduction

Each couple pair's sequential order ensures that there is no rank gap between them. Nonetheless, v^i is capable of serving as a hub for v^o's out-labels. It relies on whether there exists $sp(v^o, v^i)$ with v^i as the highest rank vertex along the path. As a result, we can keep a duplicate of each pair of couple vertices' in-labels (out-labels). When the complete index must be recovered, we just need to modify the distance element and the v^i-hub out-label entry for v^o if necessary.

F. Complexity Analysis

Theorem IV.1. *For a graph with treewidth ω, average out-degree s_f and average in-degree s_b, the index construction time of CSC is $O(n\omega^2 \log^2 n + (s_f + s_b) \cdot n\omega \log n)$, and its index size is $O(n\omega \log n)$.*

Proof. Assume that G_b has a centroid decomposition (X, T). T is a tree in which each node t corresponds to a subset of $V_{in} \cup V_{out}$ termed a bag X_t and $X = \{X_t \mid t \in T\}$. Due to the fact that we use the couple-vertex skipping technique, each pair of couple vertices can be considered as a single vertex. We assume that the number of vertices is n which is the same as that of G_0. Vertices in the root bag are ranked highest, followed by vertices in the root's successors. The BFS will never visit the vertices beyond the current vertex (hub) depth due to the labeling constraint and the ordering of label creation. A bag contains at most ω vertices, and the depth of T is $\log n$. Thus, each vertex has $2\omega \log n$ label entries, for a total of $O(n\omega \log n)$. The time complexity of evaluating a query is $O(\omega \log n)$.

Let s_f represent the average number of successors and s_b represent the average number of ancestors. Each time a label entry is produced, we traverse s_f (s_b) edges in a forward (backward) direction, respectively. In $O(\omega \log n)$ time, a query can be evaluated. Thus, the total time required to build an index is $O((s_f + \omega \log n) \cdot n\omega \log n + (s_b + \omega \log n) \cdot n\omega \log n)$ which is equal to $O(n\omega^2 \log^2 n + (s_f + s_b) \cdot n\omega \log n)$. $\square$
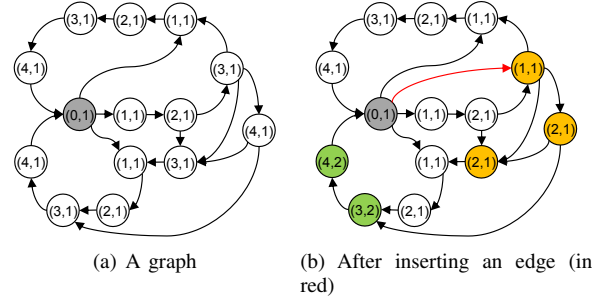

(a) A graph   (b) After inserting an edge (in red)

Fig. 6. An example for the incremental update.

V. MAINTENANCE OVER DYNAMIC GRAPHS

This section proposes an efficient algorithm for updating the bipartite hub labeling in the presence of edge insertions and deletions. Note that vertices updates can be accomplished via a series of edges updates. Thus, we concentrate on edge updates.

A. Edge Insertion

The index can be updated by simply reconstructing the whole index. To prevent performing actions on labels that are not impacted, we specify the affected hubs. The affected hubs could be out-of-date or inserted as new label entries. Our update algorithm's central idea is to perform pruned BFSs beginning from these affected hubs. The following are related lemmas and theorems:

Lemma V.1. *The shortest distance between any pair of vertices does not grow with the addition of a new edge.*

Lemma V.2. *If the shortest distance from v to w changes as a result of the insertion of edge $e(a, b)$, then all the new shortest paths from v to w travel via $e(a, b)$.*

According to Lemma V.1 and Lemma V.2 [30], some label entries should be updated or generated if the insertion of (a, b) results in new shortest paths. To discover new shortest paths, suppose we perform BFS in forward direction from a certain v_k, where $sd(v_k, a) < sd(v_k, b)$, and the BFS begins from b with distance $sd(v_k, a) + 1$ and appropriate shortest path counting information. For the reverse direction search from a specific u_k, where $sd(b, u_k) < sd(a, u_k)$, the BFS should begin at a with distance $sd(b, u_k) + 1$. Due to the fact that we need to update the index with all the new shortest paths, the BFS from v_k in the forward direction should be processed as follows.

When a vertex w is encountered, let D represent the tentative distance from v_k to w and D_L denote the distance calculated using the current index. Additionally, the provisional shortest path counting is recorded. There are three possible scenarios:

$Case_1$ If $D > D_L$, the BFS is pruned because the shortest path from v_k to w does not pass through $e(a, b)$;

$Case_2$ If $D = D_L$, we accumulate the counting as new same-length shortest path is discovered, and continue BFS;

$Case_3$ If $D < D_L$, we update the distance and counting as new shortest path with shorter distance is discovered, and continue BFS.

Example 7. *Figure 6 illustrates an example, bipartite conversion is omitted for simplicity. We begin with the grey vertex and assume it has the highest rank in the graph. The first digit of each vertex represents the shortest distance from the grey vertex, while the second digit represents the number of the shortest paths. Additionally, they are the in-label entry for each vertex associated with the grey vertex's hub. When the edge (in red) is inserted in Figure 6(b), yellow vertices alter the shortest distance ($Case_3$), whereas green vertices just affect the shortest path counting ($Case_2$). White vertices are unaffected ($Case_1$ or unvisited).*

In addition to the distance pruning, the BFS also terminates if $w \prec v_k$, indicating that v_k is an eligible hub for $L_{in}(w)$, and vice versa to flip the direction of updating out-labels. Since the query evaluation is to determine the shortest distance via common hubs, out-of-date label entries that do not reflect the shortest distance will be dominated by new label entries after the update. As a consequence, their presence has no bearing on the correctness. The selection of v_k and u_k, that is the affected hubs, is as follows.

Definition V.1 (Affected Hubs). With the addition of a new edge $e(a, b)$, we define the affected hubs associated with a as $hub_A = \{h_A | (h_A, d, c) \in L_{in}(a)\}$ and the affected hubs related to b as $hub_B = \{h_B | (h_B, d, c) \in L_{out}(b)\}$.

Some $(h_A, d, c) \in L_{in}(\cdot)$ may be out-of-date if $h_A \in hub_A$. Some $(h_B, d, c) \in L_{out}(\cdot)$ may be out-of-date if $h_B \in hub_B$. Additionally, some new in-label entries (h_A, d, c) and new out-label entries (h_B, d, c) should be generated to restore the covering constraint. Consider hub_A, for each v_k in hub_A, v_k is the highest ranked vertex among all or some of the shortest paths from v_k to a. Thus, they are eligible to pass through $e(a, b)$ and update the index using rank pruning and distance pruning. Vice versa for vertices in hub_B in the reverse direction. Other vertices are trimmed or rendered inaccessible by a in G_0 (or b in the reverse graph) during the initial index building. The new edge has no effect on the label entries that use these vertices as hubs.

Thus, the search of new shortest paths should be conducted from hub_A in the forward direction and from hub_B in the reverse direction. The shortest path counting information at the beginning of the BFS should be as follows.

Theorem V.1. *If $h \in hub_A$ (or hub_B), and $(h, d, c) \in L_{in}(a)$ (or $L_{out}(b)$). The shortest path counting utilized at the start of the BFS should be c for the affected hub h.*

Proof. As shown in Figure 7, h_c is a canonical hub in $L_{in}(a)$ and a hub in $L_{out}(h_n)$, h_n is a non-canonical hub in $L_{in}(a)$. According to the constraint of the labeling schema, h_c has a higher rank than h_n. Assume that the current affected hub to be processed is h_n, we first calculate $sd(h_n, a)$ and $SPCnt(h_n, a)$ under the current index. And then use them to start the search of new shortest paths from b. For counting labeling, if $SPCnt(h_n, a)$ is used, the initial shortest path counting also counts the paths from h_n to a via h_c if they are also the shortest. The label entries with hub h_c are already updated (if hubs are processed in descending order), which indicates the shortest path counting from h_n to other vertices via h_c is up to
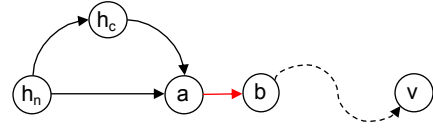

Fig. 7. Update for Canonical Hub and Non-Canonical Hub.

date. Thus, the update may overestimate the counting in non-canonical labels if using $SPCnt(h_n, a)$ before the search. □

Update algorithm INCCNT is shown in Algorithm 5. After identifying affected hubs V_k, the BFS update procedure will start from each v_k in V_k in descending order (lines 4-10). The local distance and counting of v_k are utilized to start the BFS (lines 6,9).

Algorithm 6 illustrates the process of calculating new shortest paths and updating in-labels. When a vertex w is encountered, let M_G represent the current index and $D[w]$ denote the tentative distance from v_k to w, while $D_G(v_k, w)$ specifies the distance calculated using M_G (line 7). $C[w]$ keeps track of the tentative shortest path counting. If a new shortest path is discovered ($D[w] \leq D_G(v_k, w)$), update the $L_{in}(w)$ (line 10). The BFS terminates based on the distance and rank pruning (line 12). Algorithm BACKWARDPASS is similar to Algorithm 6 except that the BFS starts at a in the reverse direction. We omit it due to the space limits.

Algorithm 7 explains how to update in-labels. If hub v_k exists in $L_{in}(w)$ and the new shortest path becomes shorter, replace the label entry with the new distance and counting (line 3). Or if the new shortest path has the same length as before, simply accumulate the counting (line 6). If hub v_k does not exist, insert a new label entry (line 8). To update out-labels, replace $L_{in}(\cdot)$ with $L_{out}(\cdot)$. M_{G_+} denotes the updated index.

Algorithm 5: INCCNT(a, b)

---

1 $hub_A \leftarrow$ hubs from $L_{in}(a)$;
2 $hub_B \leftarrow$ hubs from $L_{out}(b)$;
3 $V_k \leftarrow hub_A \cup hub_B$;
4 **for each** $v_k \in V_k$ in descending order **do**
5   **if** $v_k \in hub_A$ **and** $v_k \prec b$ **then**
6 $(v_k, d, c) \leftarrow$ label with hub v_k from $L_{in}(a)$;
7   FORWARDPASS$(v_k, b, d+1, c)$;
8   **if** $v_k \in hub_B$ **and** $v_k \prec a$ **then**
9 $(v_k, d, c) \leftarrow$ label with hub v_k from $L_{out}(b)$;
10   BACKWARDPASS$(v_k, a, d+1, c)$;

Theorem V.2. *Let M_G be the shortest cycle counting index for graph G_0 and M_{G_+} be the index updated by Algorithm 5 from M_G regarding the edge insertion to make G_+ from G_0. Then, the index M_{G_+} is a correct shortest cycle counting index for G_+.*

Proof. As shown in Figure 8, if edge $e(a, b)$ generates new shortest paths from v to w, there is at least a path from v to a and a path from b to w. Let h_v and h_w denote the hubs of the shortest paths from v to a and the shortest paths from b to w, respectively. If $h_v \prec h_w$, h_v will traverse to w in the forward direction and try to update w's in-label. And vice versa for h_w to v in the reverse direction if $h_w \prec h_v$.

Algorithm 6: FORWARDPASS(v_k, b, D, C)

1 **for each** $v \in V$ **do**
2     $D[v] \leftarrow \infty$; $C[v] \leftarrow 0$;

3 $D[b] \leftarrow D$; $C[b] \leftarrow C$;
4 Queue $Q \leftarrow \emptyset$; $Q.$enqueue(b);
5 **while** $Q \neq \emptyset$ **do**
6     $w \leftarrow Q.$dequeue();
7     $D_G(v_k, w) \leftarrow$ distance from v_k to w under M_G;
8   **if** $D[w] > D_G(v_k, w)$ **then**
9   **continue**;
10   UPDATELABEL($(v_k, D[w], C[w]), L_{in}(w)$);
11   **for each** $u \in nbr_{out}(w)$ **do**
12   **if** $D[u] > D[w] + 1$ **and** $v_k \prec u$ **then**
13           $D[u] \leftarrow D[w] + 1$; $C[u] \leftarrow C[w]$;
14           $Q.$enqueue(u);
15   **else if** $D[u] = D[w] + 1$ **then**
16           $C[u] \leftarrow C[u] + C[w]$;

17 **for each** $v \in V$ **do**
18     $D[v] \leftarrow \infty$; $C[v] \leftarrow 0$;

Algorithm 7: UPDATELABEL($(v_k, d, c), L_{in}(w)$)

1 **if** $(v_k, d', c') \in L_{in}(w)$ **then**
2   **if** $d < d'$ **then**
3   Replace (v_k, d', c') with (v_k, d, c);
4   CLEANLABEL($w, L_{in}(w)$);
5   **else if** $d = d'$ **then**
6   Replace (v_k, d', c') with $(v_k, d, c + c')$;

7 **else**
8   Insert (v_k, d, c) to $L_{in}(w)$;
9   CLEANLABEL($w, L_{in}(w)$);

Hence, after executing INCCNT(a,b), the higher-ranked one between h_v and h_w should be the hub of the corresponding shortest paths from v to w. As all the label entries with hubs like h_v and h_w will be updated. Then, for each pair of vertices, their shortest paths through $e(a, b)$ are updated into the index. Label entries related to other shortest paths are unchanged. Therefore, SPCnt(v, w) is correct under G_+, and the correctness of all pair shortest path counting guarantees the correctness of SCCnt(u) for any vertex u. $\quad\square$

Moreover, to preserve the minimality property, we need to eliminate those superfluous label entries. Redundant label entries are defined as follows:

Definition V.2 (Redundant Labels)**.** When a label entry is updated with a shorter distance or a new label entry is inserted, redundancy of the label may occur. A label entry (h, d, c) in $L_{in}(v)$ is redundant if $d > sd(h, v)$ in G_+; A label entry (h, d, c) in $L_{out}(v)$ is redundant if $d > sd(v, h)$ in G_+.

To ensure the minimality, of the label, a redundant label entry check is needed. We examine redundancy situations using the forward direction update as an example, and the
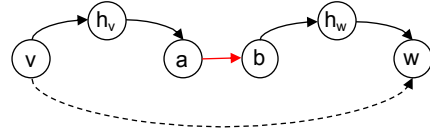

Fig. 8. Graph for Proof of Redundancy and Correctness.

Algorithm 8: CLEANLABEL($w, L_{in}(w)$)

1 **for each** $(h, d, c) \in L_{in}(w)$ **do**
2     $D_{G_+}(h, w) \leftarrow$ distance from h to w under M_{G_+};
3   **if** $d > D_{G_+}(h, w)$ **then**
4   remove (h, d, c) from $L_{in}(w)$;
5   remove w from $inv\_in(h)$;

6 **for each** $v \in inv\_out(w)$ **do**
7     $(w, d, c) \leftarrow$ the label from $L_{out}(v)$;
8     $D_{G_+}(v, w) \leftarrow$ distance from v to w under M_{G_+};
9   **if** $d > D_{G_+}(v, w)$ **then**
10   remove (w, d, c) from $L_{out}(v)$;
11   remove v from $inv\_out(w)$;

---

cleaning process is shown in Algorithm 8. As an example, in Figure 8, suppose h_v is pushed for update and encounters w. In this instance, the duplicate label entries correspond to the paths towards w (dash line in Figure 8). There are two situations of the start vertex v of such paths:

- If $v \prec w$, then (v, d, c) is a label entry in $L_{in}(w)$. Following index updates, $sd(v, w)$ must be re-evaluated. If $d > sd(v, w)$, then all the shortest paths from v to w pass via $e(a, b)$, resulting in the redundancy of (v, d, c) (lines 1-5);
- If $w \prec v$, then (w, d, c) is a label entry in $L_{out}(v)$. Similarly, we determine if $sd(v, w)$ is less than d when the index is updated. To facilitate implementation, an out-label inverted index $inv\_out(w)$ is added in this instance to locate the vertices similar to v whose out-label contains the hub w (lines 6-11). An inverted index of this kind can be constructed during the initial index creation process.

For the reverse direction cleaning, the process is similar. For the first instance, We examine the out-label and utilize an in-label inverted index $(inv\_in(\cdot))$ to locate and delete redundant in-label entries.

Theorem V.3. *If Algorithm 8 is used to update the shortest path counting index, the index is minimal. As a consequence, the lack of any label entries leads to erroneous counting query results for certain vertices.*

Proof. Algorithm 8 deletes all superfluous label entries. This indicates that for any $(h_v, d_v, c_v) \in L_{out}(v)$, $d_v = sd(v, h_v)$, h_v has the highest rank along some shortest paths from v to h_v, and the number of which is c_v. For any $(h_w, d_w, c_w) \in L_{in}(w)$, $d_w = sd(h_w, w)$, there exist c_w shortest paths from h_w to w with h_w as the highest rank vertex. We then demonstrate that the removing of any $(h_v, d_v, c_v) \in L_{out}(v)$ or $(h_w, d_w, c_w) \in L_{in}(w)$ will results in an erroneous SPCnt(v, w). Assume that h is a common hub in $L_{out}(v)$ and $L_{in}(w)$ and that it generates a portion of the shortest paths

P_h from v to w. Then, among all vertices along P_h, h has the highest rank. Assume that (h, d_h, c_h) has been removed from $L_{out}(v)$ (or $L_{in}(v)$). Another hub k exists that could induce P_h where $k \neq h$. Thus, k should be the highest ranked vertex along P_h. This implies that $k = h$ and therefore refutes the assumption. Elimination of any $(h_v, d_v, c_v) \in L_{in}(v)$ will at the very least result in $SPCnt(h_v, v)$ is the incorrect answer. And vice versa for any out-label entry. $\square$

B. Analysis

Efficiency Trade-off: The time required to remove redundant labels associated with v is dependent on the sum of $|L_{in}(v)|$ and $|inv\_out(v)|$ (or $|L_{out(v)}|$ and $|inv\_in(v)|$). Cleaning takes $O(k\omega \log n)$ time, where k is the above-mentioned size and is often considerably longer than updating with redundancy, and ω is the treewidth from Theorem IV.1. In practice, we choose to update with redundancy in order to maximize efficiency (skip lines 4 and 9 in Algorithm 7).

Time Complexity: We begin with $O(2\omega \log n)$ affected vertices. Assume that during the resumed BFS, k vertices are visited and a query is processed for each of them. Thus, the time complexity of adding a new edge is $O(k\omega^2 \log^2 n)$.

C. Edge Deletion

The deletion of an edge may result in an increase in the distance between specific pairs of vertices or a reduction in the shortest path counting between them. To ensure consistency, certain label entries must be deleted. Consider the case where $(v, d, c) \in L_{in}(w)$, but v is detached from w after the edge deletion. In this instance, (v, d, c) results in an erroneous answer of SPCnt(v, w) and SCCnt(u) for specific u when there are shortest cycles through u via v and w.

The decremental update method is comparable to the incremental technique, which could be divided into three steps. Due to space limits, we will only briefly explain them. Assume the omitted edge is $e(a, b)$. We begin by identifying two sets of impacted hubs hub_A and hub_B. Each vertex v in hub_A must fulfill the constraint $sd(v, a) + 1 = sd(v, b)$. Each vertex u in hub_B must fulfill the constraint $sd(b, u) + 1 = sd(a, u)$. Note that hub_A and hub_B may have identical vertices that are located on a cycle through (a, b). Delete out-of-date label entries in the second step as follows: If $v \in hub_A \wedge u \in hub_B$, delete (v, d, c) from $L_{in}(u)$ or delete (u, d, c) from $L_{out}(v)$ if they exist. The set of deleted label entries is a superset of out-of-date ones. The distances from or towards the vertices which are other than $hub_A \cup hub_B$ are unaffected by $e(a, b)$. The last step is to add label entries. BFS is carried out starting at each vertex v in hub_A and inserting (v, d, c) into $L_{in}(u)$ if $u \in hub_B$. Vice versa to add out-label entries from vertices in hub_B in reverse direction.

VI. Evaluation

A. Experiments Setup

Settings: Experiments are conducted on a Linux server with Intel Xeon E3-1220 CPU and 520GB memory. The algorithms are implemented in C++ and compiled by g++ at -O3 optimization level. Each label entry is encoded in a 64-bit integer. The vertex ID, distance, and counting take 23, 17, and 24 bits, respectively.

Datasets: Nine networks from SNAP[1] and Konect[2] were used for the experiments. The details are shown in Table IV. All graphs are directed and have no self-loop. For query evaluation, all vertices of each graph, or at least 50,000 vertices were used, and they were divided into five clusters according to their min-in-out degrees $\min(|nbr_{in}(v)|, |nbr_{out}(v)|)$. We first obtained the highest and lowest degree within each graph. Then divided the degree range evenly into five clusters, High, Mid-high. Mid-low, Low, and Bottom. Vertices were finally clustered based on their min-in-out degrees. For dynamic maintenance, [200,500] random edges were removed and then inserted back to each graph.

TABLE IV
THE STATISTICS OF THE GRAPHS

Graph	Notation	n	m
p2p-Gnutella04[1]	G04	10,879	39,994
p2p-Gnutella30[1]	G30	36,682	88,328
email-EuAll[1]	EME	265,214	420,045
web-NotreDame[1]	WBN	325,729	1,497,134
wiki-Talk[1]	WKT	2,394,385	5,021,410
web-BerkStan[1]	WBB	685,231	7,600,595
Hudong-Related[2]	HDR	2,452,715	18,854,882
wiki_link_War[2]	WAR	2,093,450	38,631,915
wiki_link_SR[2]	WSR	3,175,009	139,586,199

Compared Algorithms: We compared the following algorithms (i) HP-SPC (Baseline) and (ii) CSC (Proposed algorithm) for the static shortest cycle counting experiments based on the index construction time, index size, and query time for each graph. In addition, we used the naïve BFS for the comparison of the query time.

B. Experiment Results on Static Graph

1) Index Time: Figure 9(a) shows the index construction time taken by HP-SPC and CSC. The results indicate that (i) for graphs EME, WBN, and WKT, even though HP-SPC is 1.22 to 1.38 times faster than CSC in constructing index, the time differences are not significant. For instance, the index time ratio between CSC and HP-SPC for graph EME is 1.38. The time difference is 4.5s; (ii) the index construction times of CSC for other tested graphs are slightly longer than HP-SPC. The time difference does not exceed 8%. When the number of edges exceeds ten million, the time difference is reduced to within 1.5%; (iii) CSC can index any tested graphs with less than ten million edges in 24 minutes. For two graphs with tens of millions of edges, index time of HDR is 15.5 hours, while for WAR, it only requires 0.5 hour. It takes the longest to index the largest graph WSR, roughly 61 hours.

2) Index Size: Figure 9(b) illustrates the results in index size. It is expected that the two algorithms generate a similar size index, and the actual result proves it. We observe that the index size of CSC is nearly the same as that of HP-SPC. The two most significant differences in index size between the two algorithms are 4.4% and 2.7% for graph WBN and EME, respectively. At the same time, the index differences for other graphs are all less than 1%.
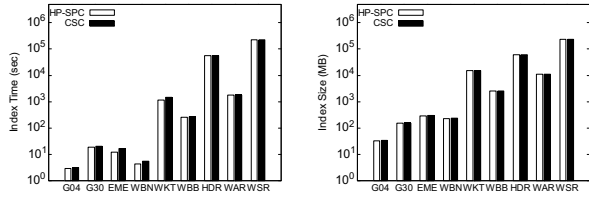
(a) Index Time   (b) Index Size
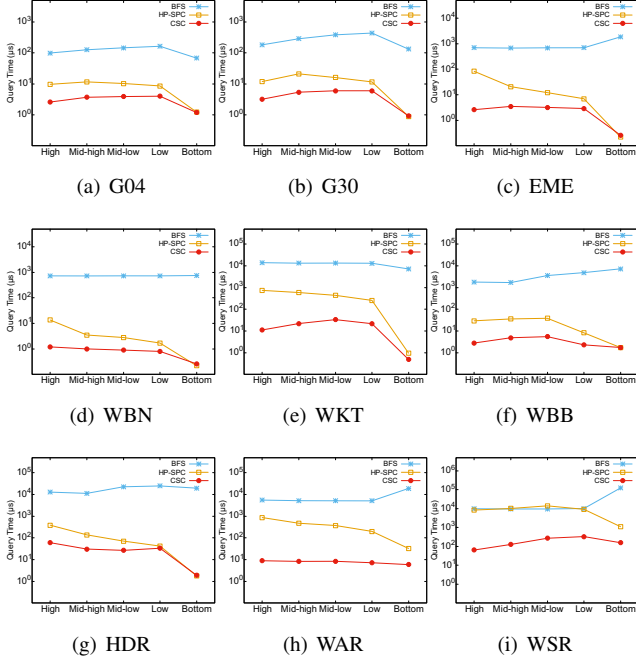
Fig. 9. Index Time (sec) and Index Size (MB).

(a) G04   (b) G30   (c) EME

(d) WBN   (e) WKT   (f) WBB

(g) HDR   (h) WAR   (i) WSR

Fig. 10. Query Times (μs).

3) Query Time: Figure 10 illustrates the average query time of each cluster for each graph taken by BFS, HP-SPC, and CSC. There is no evidence that vertex degree has an influence on query time under a naïve BFS approach, while its query time is always at a very high value. As shown in each sub-figure, the query time of CSC is more stable than that of HP-SPC and can stay at microsecond level all the time even for the largest graph. There was not a significant positive linear correlation between the query time of HP-SPC and the degree of query vertices. Because the HP-SPC method involves searching through $v's$ neighbors to answer an SCCnt(v) query, the query time is $deg_m * t_P$. Notably, $deg_m = \min(|nbr_{in}(v)|, |nbr_{out}(v)|)$ and t_P is the query time of a single SPCnt evaluation. A higher-degree query vertex has a larger deg_m. However, the variety of t_P cannot be identified in the interim as we do not know in advance the label sizes of the query vertices, which may be very varied. As a result, the query time does not increase linearly with the degree of query vertex all the time. Nevertheless, evaluating a single SPCnt query is very efficient. Thus, deg_m is a critical component affecting the final query time. If deg_m is large, HP-SPC may take more time to evaluate an SCCnt query. Nevertheless, for those higher-ranked query vertices (in High and Mid-high clusters), HP-SPC takes much more time to evaluate an SCCnt query. Meanwhile, CSC is from 3.11 to 130.1 times faster than HP-SPC among these clusters. Due to the couple-vertex

skipping approach, the label size of each vertex is almost the same under CSC as it is under HP-SPC. As a result, the query time of CSC is quite similar to the time required for a single SPCnt evaluation. CSC is no longer required to search through the neighbors. When the query vertex's degree is very large, CSC should have significantly improved query performance. Note that in graphs WKT, WAR and WSR, the average query time for High cluster with CSC is up to two orders of magnitude faster than HP-SPC. For those lower-ranked query vertices (in Mid-low and Low clusters), CSC is still 1.3 to 51.8 times faster. For those vertices with a very small degree, the two algorithms provide similar performance. In summary, these results indicate that CSC is efficient and stable for SCCnt queries with microseconds responses.

C. Experiment Results on Dynamic Graph
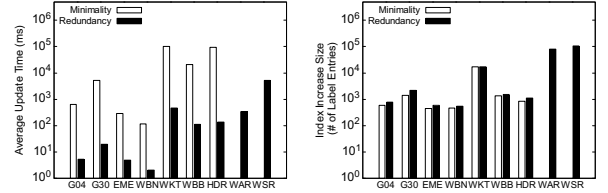
(a) Average Update Time   (b) Index Increase Size

Fig. 11. Average Update Time (ms) and Index Increase Size (Number of Label Entries) of Incremental Maintenance.

1) Update Time: Figure 11(a) shows the average incremental update time under both minimality and redundancy strategy. The update time was observed to have a similar trend as the index construction time for the graphs with less than ten million edges. The similar process of INCCNT as that of CSC can explain this. However, INCCNT is pruned earlier and more often than CSC with different conditions. The update under minimality strategy is 58 to 678 times slower than the redundancy approach. This is because of the large number of redundancy checks. Due to the time cost of minimality strategy, it is omitted for graphs WAR and WSR. When the graph size becomes larger, the update time doesn't increase too much. It benefits from the small world phenomenon [31]. In most cases, the change of an edge will not affect the shortest distance in a large range. Under redundancy strategy, although graphs WKT, WBB, HDR, WAR vary in edge size from millions to tens of millions, their update times are similar, ranging from 112ms to 469ms. For the largest graph WSR, we set the update time limit as 60 seconds (s). If an update cannot be completed within the time limit, it is terminated and the update time is set as 60s. The average update time of WSR under redundancy strategy is 5.2s. Compared with a reconstruction for each edge insertion, INCCNT only requires 2.3×10^{-5} of the reconstruction time for a single update. The results prove the effectiveness and the scalability of INCCNT.

Figure 12(a) illustrates the average decremental update time of Graph G04. For an edge $e(v, w)$, the degree of this edge is defined as the sum of in-degree of v and out-degree of w. These 500 edges are divided into five clusters according to their edge degrees. The High cluster contains those edges with the largest edge degrees. Following the addition of edge degree, a significant increase in the update time was recorded. The average update time of an edge in High cluster is about

2.6s while it is around 0.25s for the edges in Bottom cluster. The total average update time is 0.93s.
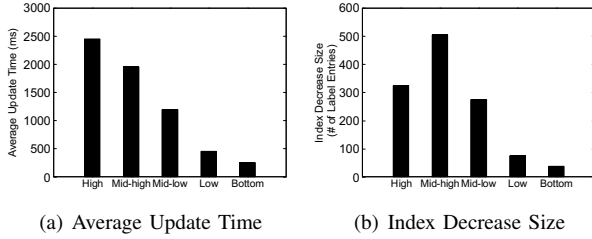

(a) Average Update Time (b) Index Decrease Size

Fig. 12. Average Update Time (ms) and Index Decrease Size (Number of Label Entries) of Decremental Maintenance (Graph G04).

2) Index Size: Figure 11(b) provides the average increase size in label entry of incremental update. The number of newly added label entries for each update is from 443 to 1447 for the first seven graphs (except for WKT) under the minimality approach. As each label entry takes up 64 bits. The increased index size for these graphs is from 3.5KB to 11.5KB, and similarly, it is 135KB for graph WKT. Among most of the graphs, the increased index size accounts for less than 0.01% of the original index. The results show that the difference in the increased label size between minimality and redundancy approaches is minor. Compared with the significant difference in their update time, the redundancy approach is considered ideal. For graphs WAR and WSR, their index increase sizes under redundancy only account for 5.7×10^{-5} and 3.4×10^{-6} of the size of their original indexes. Overall, the results illustrate that the label size under INCSCT grows slowly.

From the data in Figure 12(b), it is apparent that the deletion of high-degree edges may cause more label entries deleted. A large number of unaffected label entries are removed and recovered later.

D. Case Study

The real network dataset MAHINDAS [32] used for the case study is in the Economic Networks category. In this graph, vertices represent accounts, and edges represent transactions between them. The vertex size denotes the shortest cycle counting. The bigger a vertex, the more the shortest cycles pass through it. The vertex color denotes the shortest cycle length. The darker a vertex, the longer the shortest cycles. Figure 13 shows a subgraph centering at vertex 169, which all the shortest cycles through vertex 169 are listed. The figure matches the subgraph model in Figure 1. Vertices 281, 241, 169, 1159, and 888 are filtered out to be suspicious accounts for anomaly detection, e.g., money laundering. Based on these candidates derived from our algorithm, we could further analyse whether there is an exact case for the financial crime by enumerating such cycles or paths [11], [23]

VII. RELATED WORK

Counting Patterns: Counting the number of specific patterns is a fundamental problem in graph theory. Some counting problems are #P-complete [33]. For instance, counting the number of all the simple cycles through a given vertex or the number of simple paths between two vertices. The problem of counting the simple paths or cycles with a length constraint of l, parameterized by l, is considered to be #W[1]-complete [34]. Many works proposed various algorithms to answer such
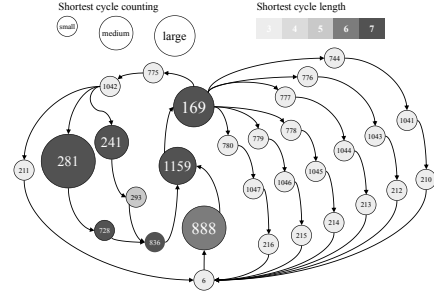

Fig. 13. A subgraph centering at vertex 169 with suspicious criminal nodes from a real economic network MAHINDAS.

problems. [35] could answer the number of shortest paths in $O(\sqrt{n})$ where n is the number of vertices, but the graph must be planar and acyclic. [36] can also answer the number of shortest cycles, but the algorithm is too slow even when the graph is not so large. Many other similar works share the shortcomings like the graph type constraint or the undesirable query time.

Enumerating Patterns: Enumerating the number of specific patterns is another method to produce the count. However, it is supposed to take much more time than direct counting [36]. For instance, when counting shortest cycles, one of the primary reasons is that the shortest cycle length is unknown in advance, and obtaining it requires the employment of a BFS-like method whose running time is already longer than the query time of hub labeling. Another reason is that enumeration requires finding all the vertices along with the cycles, which is unnecessary for the counting problem. With the help of a hot-point based index, [11] can output the newly-generated constraint cycles upon the arrival of new edges. [23] is shown to have better time performance than [11] on enumerating constraint paths and cycles.

Dynamic Maintenance for 2-hop Labeling: To adapt to the dynamic update of the network, some works [30], [37], [38] proposed dynamic algorithms to deal with edge insertion and deletion. For the edge insertion, a partial BFS for each affected hub is started from one of the inserted-edge endpoints and creates a label when finding the tentative distance is shorter than the query answer from the previous index. For the edge deletion, the state-of-the-art solution is to find the affected hubs followed by removing out-of-date labels, then recover labels. The time cost sometimes is acceptable compared with reconstructing the whole index. But it is much slower than that of incremental update.

VIII. CONCLUSION

We investigate the problem of counting the number of shortest cycles through a vertex. A 2-hop labeling based algorithm is proposed for shortest cycle counting query. The comprehensive experiments demonstrate that our algorithms could achieve up to two orders of magnitude faster than state-of-the-art. We also present an update algorithm to maintain the index for edge updates. Our comprehensive evaluations verify the effectiveness and efficiency of the algorithms.

ACKNOWLEDGMENT

REFERENCES

[1] 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)*. IEEE, 2018, pp. 1192–1203.

[2] 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.

[3] 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.

[4] 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.

[5] Y. Peng, S. Bian, R. Li, S. Wang, and J. X. Yu, "Finding top-r influential communities under aggregation function," in *ICDE*. IEEE, 2022.

[6] 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.

[7] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang, "Answering billion-scale label-constrained reachability queries within microsecond," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 812–825, 2020.

[8] 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.

[9] 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.

[10] 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.

[11] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[12] H. Weinblatt, "A new search algorithm for finding the simple cycles of a finite directed graph," *Journal of the ACM (JACM)*, vol. 19, no. 1, pp. 43–56, 1972.

[13] C. Thomassen, "Girth in graphs," *Journal of Combinatorial Theory, Series B*, vol. 35, no. 2, pp. 129–141, 1983.

[14] J. Gimbel and C. Thomassen, "Coloring graphs with fixed genus and girth," *Transactions of the American Mathematical Society*, vol. 349, no. 11, pp. 4555–4564, 1997.

[15] R. Yuster, "A shortest cycle for each vertex of a graph," *Information Processing Letters*, vol. 111, no. 21, pp. 1057 – 1061, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019011002195

[16] H. Bonneau, A. Hassid, O. Biham, R. Kühn, and E. Katzav, "Distribution of shortest cycle lengths in random networks," *Physical Review E*, vol. 96, no. 6, p. 062307, 2017.

[17] N. Vladimirov, Y. Tu, and R. D. Traub, "Shortest loops are pacemakers in random networks of electrically coupled axons," *Frontiers in Computational Neuroscience*, vol. 6, p. 17, 2012.

[18] P. M. Gleiss, P. F. Stadler, A. Wagner, and D. A. Fell, "Relevant cycles in chemical reaction networks," *Advances in complex systems*, vol. 4, no. 02n03, pp. 207–226, 2001.

[19] S. Klamt and A. von Kamp, "Computing paths and cycles in biological interaction graphs," *BMC bioinformatics*, vol. 10, no. 1, pp. 1–11, 2009.

[20] J. G. T. Zañudo, G. Yang, and R. Albert, "Structure-based control of complex networks with nonlinear dynamics," *Proceedings of the National Academy of Sciences*, vol. 114, no. 28, pp. 7234–7239, 2017.

[21] A. Barrat, M. Barthelemy, and A. Vespignani, *Dynamical processes on complex networks*. Cambridge university press, 2008.

[22] 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*. Ieee, 2007, pp. 5519–5522.

[23] 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.

[24] L. Šubelj, Š. Furlan, and M. Bajec, "An expert system for detecting automobile insurance fraud using social network analysis," *Expert Systems with Applications*, vol. 38, no. 1, pp. 1039–1052, 2011.

[25] A. Bodaghi and B. Teimourpour, "Automobile insurance fraud detection using social network analysis," in *Applications of Data Management and Analysis*. Springer, 2018, pp. 11–16.

[26] L. Hajdu and M. Krész, "Temporal network analytics for fraud detection in the banking sector," in *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*. Springer, 2020, pp. 145–157.

[27] M. Schlosser, T. Condie, and S. Kamvar, "Simulating a file-sharing p2p network," in *1st Workshop on Semantics in Grid and P2P Networks*. Stanford InfoLab, 2003.

[28] C. Ohta, Z. Ge, Y. Guo, and J. Kurose, "Index-server optimization for p2p file sharing in mobile ad hoc networks," in *IEEE Global Telecommunications Conference, 2004. GLOBECOM'04.*, vol. 2. IEEE, 2004, pp. 960–966.

[29] Y. Zhang and J. X. Yu, "Hub labeling for shortest path counting," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1813–1828.

[30] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 237–248.

[31] D. J. Watts, "Networks, dynamics, and the small-world phenomenon," *American Journal of sociology*, vol. 105, no. 2, pp. 493–527, 1999.

[32] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[33] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.

[34] J. Flum and M. Grohe, "The parameterized complexity of counting problems," *SIAM Journal on Computing*, vol. 33, no. 4, pp. 892–922, 2004.

[35] I. Bezáková and A. Searns, "On counting oracles for path problems," in *29th International Symposium on Algorithms and Computation (ISAAC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, pp. 56:1–56:12.

[36] P.-L. Giscard, N. Kriege, and R. C. Wilson, "A general purpose algorithm for counting simple cycles and simple paths of any length," *Algorithmica*, vol. 81, no. 7, pp. 2716–2737, 2019.

[37] G. D'angelo, M. D'emidio, and D. Frigioni, "Fully dynamic 2-hop cover labeling," *Journal of Experimental Algorithmics (JEA)*, vol. 24, pp. 1–36, 2019.

[38] Y. Qin, Q. Z. Sheng, N. J. Falkner, L. Yao, and S. Parkinson, "Efficient computation of distance labeling for decremental updates in large dynamic graphs," *World Wide Web*, vol. 20, no. 5, pp. 915–937, 2017.