Efficient Index Maintenance for Effective Resistance Computation on Evolving Graphs

ABSTRACT

In this paper, we study a problem of index maintenance on evolving graphs for effective resistance computation. Unlike an existing matrices-based index, we show that the index can be efficiently maintained by directly preserving samples of random walks and loop-erased walks. This approach not only enables efficient storage and rapid query response but also supports effective maintenance. We propose a novel approach to convert edge updates into *landmark* node updates. Building upon this, we present two new update algorithms for random walk and loop-erased walk samples respectively. Both algorithms update samples without requiring complete resampling, ensuring accuracy and high efficiency. A particularly challenging and innovative technique involves updating loop-erased walks. Here we develop a novel and powerful cycle decomposition technique for loop-erased walks, enabling us to update samples at the cycle level rather than the node level, significantly enhancing efficiency. Furthermore, we show that both of our methods achieve an $\widetilde{O}(1)$ time complexity per edge update in real-world graphs under a mild assumption. We conduct extensive experiments using 10 large real-world datasets to evaluate the performance of our approaches. The results show that our best algorithm can be up to two orders of magnitude faster than the baseline methods.

1 INTRODUCTION

Effective resistance [58] computation is a fundamental problem in graph data management, capable of measuring node similarity and serving as a distance metric [12, 20, 34, 35]. It finds extensive applications in node similarity query [35, 37, 45, 66], robust routing [56], geo-social network clustering [54] and graph neural networks [39, 59]. However, existing algorithms for effective resistance computation [35, 37, 45, 66] are primarily designed for static graphs, rendering them inefficient for evolving graphs. Real-world graphs are rapidly changing, necessitating swift responses to incoming queries. This dynamic nature has spurred a substantial body of research focused on the computation of graph node similarity in dynamic environments [10, 28, 49, 50, 52, 67–69]. In theoretical computer science community, dynamic effective resistance is also a vital component in solving the maximum flow problem and some other fundamental graph algorithms [13, 14, 18, 21, 24, 31, 57].

However, even on static graphs, computing effective resistance poses a significant challenge due to its reliance on intensive matrix computations. Consequently, numerous studies have concentrated on developing approximate algorithms using techniques such as sampling random walks [45, 66] and loop-erased random walks [35, 37]. Among all existing solutions, the state-of-the-art (SOTA) approach is an index-based approach with an elegant multiple landmarks technique [37]. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with Laplacian matrix $\mathbf{L}$, they first divide the node set $\mathcal{V}$ into $\mathcal{U} \cup \mathcal{V}_l$, where $\mathcal{V}_l$ is the landmark node set and $\mathcal{U}$ is the remaining node set. Based on the concept of Schur complement [9], they present new formulas to compute $r(s,t)$ in terms of elements of $\mathbf{L}^{-1}_{\mathcal{U}\mathcal{U}}$, the inverse of the sub-matrix of $\mathbf{L}$ indexed by rows and columns corresponding to $\mathcal{U}$, along with two pre-computed matrices that serve as an index. Such a matrices-based index takes $O(n \cdot |\mathcal{V}_l|)$ space, and it can be efficiently constructed by sampling $\mathcal{V}_l$-absorbed random walks or $\mathcal{V}_l$-absorbed loop-erased walks [9]. However, this index-based approach is inefficient for dynamic graphs. When the underlying graph evolves, it is necessary to re-compute the index from scratch using sampling techniques.

To address this issue, in this paper, we propose two new index methods, namely RWIndex and LEIndex respectively, which directly maintain the random walk or loop-erased random walk samples, rather than store the matrices as [37]. We show that both RWIndex and LEIndex not only enable efficient storage and rapid query response but also support effective maintenance.

Specifically, to maintain both RWIndex and LEIndex in evolving graphs, we develop a novel approach that transform the operations of handling a single edge update to the operations of handling at most two landmark node updates. We show that the operations of updating landmark nodes can be efficiently implemented by truncating or expanding the random walk and loop-erased random walk samples, making the procedure of index maintenance highly efficient. A striking feature of our approach is that it only needs to maintain a small portion of random walk or loop-erased random walk samples that are related to the two updated landmark nodes, without re-draw the entire random walk or loop-erased random walk samples from scratch. We prove that our approach to maintain both RWIndex and LEIndex can be performed in $\widetilde{O}(1)$ time per edge update in real-world graphs, under the same mild assumption as [37]. Here, $\widetilde{O}(\cdot)$ notation hides polylogarithmic (poly log n) factors.

In our index maintenance approach, a particularly challenging and innovative technique involves updating the loop-erased random walk samples. We develop a novel and powerful cycle decomposition technique that decomposes loop-erased random walks into cycles and spanning forests. These cycles are shown to form a directed acyclic graph (DAG), which proves beneficial for designing efficient update algorithms. Specifically, leveraging this property allows us to update our loop-erased random walk samples at the cycle level rather than the node level, resulting in a significant improvement in efficiency.

SIGMOD '25, June 03–05, 2022, Woodstock, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We conducted extensive experiments to evaluate the performance of our approach. The results indicate that both of our proposed index structures RWIndex and LEIndex enable more efficient updates compared to re-sampling methods. Among the solutions we developed, LEIndex stands out, achieving rapid index construction and fast query response time, while significantly reducing update costs compared to RWIndex. Notably, within the large social network LiveJournal, which comprises 4 million nodes and 35 million edges, LEIndex demonstrated its efficiency with an average update time of only 23 seconds. In contrast, the re-sampling approach required 9, 354 seconds, highlighting a at least two orders of magnitude speed-up achieved by our methods. We also conduct a case study to illustrate the utility of effective resistance for the task of link prediction. In summary, the contributions of this paper are summarized as follows:

New theoretical results. For both the RWIndex and LEIndex indexes maintenance problems, we develop a new approach to convert the operations of edge updates to operations of landmark node updates. We propose a novel cycle decomposition technique for updating the loop-erased random walk samples; and we prove that all the cycles form a directed acyclic graph (DAG), which ensures that we can update the samples at the cycle level rather than the node level, thus significantly boosting the sample updating performance. These new theoretical contributions are anticipated to be of independent interest.

Efficient index update algorithms. We propose two efficient index update algorithms for both RWIndex and LEIndex. Specifically, we design an efficient algorithm for cycle decomposition of the loop-erased walk. We show that it can be implemented efficiently with a slight modification of the classic Wilson algorithm [64] for sampling spanning forests. Additionally, we develop algorithms to handle the addition and removal of landmark nodes, ensuring that both RWIndex and LEIndex can be updated within $\widetilde{O}(1)$ time per edge in real-world graphs, under a mild assumption.

Extensive experiments. Extensive experiments on 10 large real-world graphs demonstrate that our index update methods achieve performance improvements of up to two orders of magnitude compared to the baseline methods. The results also show that our maintenance algorithm for LEIndex is extremely efficient and match the $\widetilde{O}(1)$ update time complexity in large real-world graphs. For reproducibility purposes, the source code of this paper is available at https://github.com/mhliao0516/LEindex.

2 PRELIMINARIES

Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n vertices and m edges. The Laplacian matrix is $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the degree matrix (a diagonal matrix where each diagonal element is the degree of a node) and $\mathbf{A}$ is the adjacency matrix. $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ is the probability transition matrix. Let $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ be the eigenvalues of $\mathbf{L}$, with the corresponding eigenvectors $\mathbf{u}_1, \cdots, \mathbf{u}_n$, the pseudo-inverse of the Laplacian matrix is defined as $\mathbf{L}^{\dagger} = \sum_{i=2}^{n} \lambda_i \mathbf{u}_i \mathbf{u}_i^T$. Given two nodes s, t, the effective resistance is defined as $r(s,t) = (\mathbf{e}_s - \mathbf{e}_t)^T \mathbf{L}^{\dagger} (\mathbf{e}_s - \mathbf{e}_t)$, where $\mathbf{e}_s$ is a one-hot vector that the s-th element is 1 and other elements are 0. It is well known that the effective resistance is closely related to the commute time of random walk on graphs [58]. That is, $r(s,t)$ is proportional to the expected number of steps that a random walk starting from s, hitting t, and then coming back to s. Since computing $\mathbf{L}^{\dagger}$ is challenging, calculating the effective resistance based on its definition incurs significant costs. To overcome this challenge, existing methods [35, 37] often represent $\mathbf{L}^{\dagger}$ using its submatrices. These submatrices relate to diverse combinatorial objects such as random walks and spanning forests, motivating the development of efficient combinatorial sampling algorithms to approximate the effective resistance.

An evolving graph is a graph subject to updates over time. Let $\mathcal{G}^t$ denote the graph at time t ($t > 0$). Without loss of generality, we primarily consider the case of evolving graphs with only one edge update (edge insertion or deletion). That is, the updated graph $\mathcal{G}^{t+1}$ differs from $\mathcal{G}^t$ by a single edge. Note that adding a node can be effectively transformed into the addition of multiple edges; and similarly, deleting a node can be reduced to the deletion of several edges. Thus, all the proposed techniques can also be extended to handle node updates. As the graph evolves, we adopt a similar assumption to that in [37]. Specifically, we assume the evolving graph is rapidly mixing ($\frac{1}{1-\lambda} = \widetilde{O}(1)$, where λ is the spectral radius of $\mathbf{P}$) and has a small diameter ($\Delta_{\mathcal{G}} = \widetilde{O}(1)$, where $\Delta_{\mathcal{G}}$ is the diameter of $\mathcal{G}$). In this paper, we study the problem of maintaining the index proposed in [37] to support single-pair effective resistance query on evolving graphs, which is the SOTA approach for effective resistance computation. In the following subsection, we systematically introduce the index-based approach proposed in [37].

2.1 Multiple landmark-based index approach

New formulas. Given a *landmark* node subset $\mathcal{V}_l$, we can partition the node set $\mathcal{V}$ into $\mathcal{V} = \mathcal{U} \cup \mathcal{V}_l$. Simultaneously, matrix $\mathbf{L}$ can be decomposed into blocks as $\begin{bmatrix} \mathbf{L}_{\mathcal{U}\mathcal{U}} & \mathbf{L}_{\mathcal{U}\mathcal{V}_l} \\ \mathbf{L}_{\mathcal{V}_l\mathcal{U}} & \mathbf{L}_{\mathcal{V}_l\mathcal{V}_l} \end{bmatrix}$. Then, the Schur complement $\mathbf{L}/\mathcal{V}_l$ is defined as $\mathbf{L}/\mathcal{V}_l = \mathbf{L}_{\mathcal{V}_l\mathcal{V}_l} - \mathbf{L}_{\mathcal{V}_l\mathcal{U}}\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$. To improve the efficiency of the single-landmark appraoch [35], [37] presents the following multiple-landmark effective resistance formula.

Theorem 2.1. *[37] Let $\mathbf{p}_u$ be the u-th row of the matrix $\mathbf{P}_r$ ($\mathbf{P}_f$) for $u \in \mathcal{U}$, where $\mathbf{p}_u(v)$ is the probability that a random walk from u hits $v \in \mathcal{V}_l$ (the probability that in a random spanning forest with root set $\mathcal{V}_l$, u is rooted at v). Let $\mathbf{e}_u$ be a one-hot vector such that the element indexed by u is 1 and other elements are 0. Then,*

(1) For $u_1, u_2 \in \mathcal{U}$, we have

$$r(u_1,u_2) = (\mathbf{e}_{u_1} - \mathbf{e}_{u_2})^T (\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}) (\mathbf{e}_{u_1} - \mathbf{e}_{u_2}) + (\mathbf{p}_{u_1} - \mathbf{p}_{u_2})^T (\mathbf{L}/\mathcal{V}_l)^{\dagger} (\mathbf{p}_{u_1} - \mathbf{p}_{u_2}); \tag{1}$$

(2) For $u \in \mathcal{U}, v \in \mathcal{V}_l$, we have

$$r(u,v) = \mathbf{e}_u^T \mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1} \mathbf{e}_u + (\mathbf{p}_u - \mathbf{e}_v)^T (\mathbf{L}/\mathcal{V}_l)^{\dagger} (\mathbf{p}_u - \mathbf{e}_v); \tag{2}$$

(3) For $v_1, v_2 \in \mathcal{V}_l$, we have

$$r(v_1,v_2) = (\mathbf{e}_{v_1} - \mathbf{e}_{v_2})^T (\mathbf{L}/\mathcal{V}_l)^{\dagger} (\mathbf{e}_{v_1} - \mathbf{e}_{v_2}). \tag{3}$$

Index building. Based on Theorem 2.1, [37] first employs random walk sampling and loop-erased random walk sampling to approximate the two matrices $-\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^{\dagger}$. The time complexities for sampling random walks and spanning forests are $\omega \cdot \vec{1}^T(\mathbf{I}-\mathbf{P}_{\mathcal{UU}})^{-1}\vec{1}$ and $\omega \cdot \mathrm{Tr}((\mathbf{I}-\mathbf{P}_{\mathcal{UU}})^{-1})$, respectively, where ω is the sample size. To achieve an ϵ-absolute error (i.e., $|\hat{r}(s,t)-r(s,t)| \le \epsilon$), they show that, assuming the graph is rapidly mixing and has a small diameter—common characteristics of real-world graphs—the sample size required for each node in the index is only $\widetilde{O}(1)$. They select the landmark set $\mathcal{V}_l$ as a small, easy-to-hit node set (e.g., $|\mathcal{V}_l| = 10$ with $\mathcal{V}_l$ being the highest degree nodes). Then, $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ is exactly the matrix $\mathbf{P}_r$ ($\mathbf{P}_f$) defined in Theorem 2.1, which can be estimated by examining the terminate nodes in random walks from each node in $\mathcal{U}$ or the root nodes of each node in $\mathcal{U}$ in the spanning forest. The time complexity of estimating $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ is $\omega \cdot |\mathcal{U}|$, which can be bounded by $\widetilde{O}(n)$. The v_1, v_2-th element of $\mathbf{L}/\mathcal{V}_l$ is the number of random walks from the neighbors of v_1 that hit $\mathcal{V}_l$ by v_2 (the number of neighbors of v_1 that are rooted at v_2 in a random spanning forest). Thus, $\mathbf{L}/\mathcal{V}_l$ can be estimated by examining the terminate nodes or root nodes. $(\mathbf{L}/\mathcal{V}_l)^\dagger$ is then computed directly. Since $|\mathcal{V}_l|$ is a small constant, the time complexity of estimating $\mathbf{L}/\mathcal{V}_l$ ($\omega \sum_{v\in\mathcal{V}_l} d_v$) and computing the pseudo-inverse ($O(|\mathcal{V}_l|^3)$) is negligible compared to sampling random walks and spanning forests, which can be bounded by $\widetilde{O}(n)$ under the rapid mixing and small diameter assumptions. These two approximate matrices are then stored as an index. Consequently, the index can be built within near-linear time (i.e., $\widetilde{O}(n)$) and the space complexity is $O(n \cdot |\mathcal{V}_l|) = O(n)$ (since $|\mathcal{V}_l|$ is a small constant).

Query processing. Based on such an index, in the query processing stage, it is only need to approximate several elements of $\mathbf{L}_{\mathcal{UU}}^{-1}$. [37] proposes $\mathcal{V}_l$-absorbed walk sampling and $\mathcal{V}_l$-absorbed push algorithms for this purpose. They can also be combined as a best BiPush algorithm to reduce the variance of random walk sampling. The query processing algorithm can achieve $\widetilde{O}(1)$ time under the rapid mixing and small diameter assumption.

What's new compared to [37]? In this paper, we focus on the dynamic maintenance of the indices proposed in [37]. While the index-based approach in [37] is efficient for static graphs, maintaining only the two approximated matrices, $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^\dagger$, becomes insufficient for handling dynamic updates. As the graph evolves, these matrices need to be recomputed using random walk or loop-erased walk sampling, which incurs significant computational costs. To address this challenge, we propose two novel approaches to maintain indices that compute effective resistance on evolving graphs: one for maintaining random walk samples and another for maintaining loop-erased random walk samples. Instead of storing the numerical values of the matrices, we modify the index structure by directly storing the random samples. In Sections 3.1 and 4.1, we leverage the results from [37] to determine the sample size and query complexity, demonstrating that storing random samples does not increase space costs and maintains similar query complexity to the original approach. More importantly, the results on index maintenance presented in Sections 3.2, 4.2 and 4.3 including the idea of handling edge update to landmark nodes update and the novel cycle decomposition technique for loop-erased walk trajectories, are original contributions which are not explored in previous studies.

3 RANDOM-WALK INDEX MAINTENANCE

In this section, we first introduce the random walk index, which is slightly different from the index proposed in [37], and then we propose a novel and efficient index maintenance algorithm.

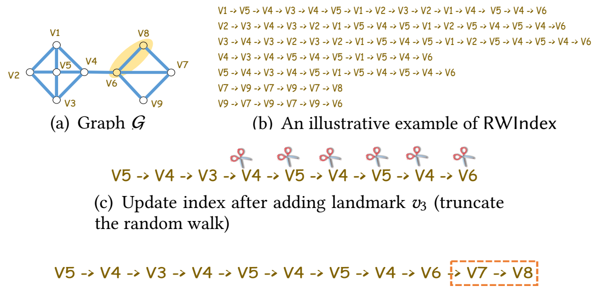
3.1 A random walk index: RWIndex

Notice that to approximate $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^\dagger$, the random walk sampling approach described in [37] samples $\mathcal{V}_l$-absorbed random walks from each node in $\mathcal{U}$. Instead of merely maintaining these matrices as an index, for each node in $\mathcal{U}$, we sample ω $\mathcal{V}_l$-absorbed random walks (ω is roughly $\widetilde{O}(1)$ [37]), and record all these random walk trajectories as an index. This approach is referred to as RWIndex.

Fig. 1(b) illustrates a simple example of RWIndex. Given a graph $\mathcal{G}$ shown in Fig. 1(a), suppose that v_6 and v_8 are the landmark nodes. RWIndex includes 7 random walks that start from all remaining nodes until they hit (or visit) v_6 or v_8. For a random walk starting from u, the expected length of the $\mathcal{V}_l$-absorbed random walk is the hitting time from u to $\mathcal{V}_l$, denoted as $h(u,\mathcal{V}_l)$. As indicated in [37], the expected space complexity for each sample is $\sum_{u\in\mathcal{U}} h(u,\mathcal{V}_l) = \vec{1}^T(\mathbf{I}-\mathbf{P}_{\mathcal{UU}})^{-1}\vec{1}$. Since ω is $\widetilde{O}(1)$ and $\vec{1}^T(\mathbf{I}-\mathbf{P}_{\mathcal{UU}})^{-1}\vec{1}$ is $\widetilde{O}(n)$ in real-life graphs, the space overhead of RWIndex is $\widetilde{O}(n)$. Compared to the original matrix-based index [37] with a space complexity of $O(n \cdot |\mathcal{V}_l|) = O(n)$ (since $|\mathcal{V}_l|$ is a small constant), RWIndex incurs no significant additional space costs.

Note that with RWIndex, there is no need to explicitly compute the matrices $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^\dagger$ for query processing. When processing an $r(s,t)$ query, efficient responses can be provided directly based on stored random walk samples. For the first matrix $-\mathbf{L}_{\mathcal{UU}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$, by Theorem 2.1, we only need to compute $\mathbf{p}_s$ and $\mathbf{p}_t$ if they belong to $\mathcal{U}$. This operation is highly efficient, involving a quick check for each sample to determine the nodes in $\mathcal{V}_l$ at which s and t terminate, with a cost of $\widetilde{O}(1)$. For the second matrix $(\mathbf{L}/\mathcal{V}_l)^\dagger$, we need to compute it from these random walk samples maintained in RWIndex following the methods proposed in [37]. Specifically, for the v_1, v_2-th element of $(\mathbf{L}/\mathcal{V}_l)^\dagger$, we need to examine the number of random walks from the neighbors of v_1 that hit $\mathcal{V}_l$ by v_2 and vice versa. This process has a time complexity of $d_{v_1}+d_{v_2}$. Thus, the total time complexity is $\omega \cdot 2\sum_{v\in\mathcal{V}_l} d_v$. Computing the pseudo-inverse of the $|\mathcal{V}_l| \times |\mathcal{V}_l|$-matrix takes $O(|\mathcal{V}_l|^3)$ time. These time costs are negligible (as $|\mathcal{V}_l|$ is small, e.g., $|\mathcal{V}_l| = 10$ [37]) compared to the index-building process. It then remains to compute elements of $\mathbf{L}_{\mathcal{UU}}^{-1}$. To achieve this, we adopt the $\mathcal{V}_l$-absorbed walk technique as described in [37]. Additionally, we can employ the $\mathcal{V}_l$-absorbed push method to reduce the variance, which refers to as the BiPush algorithm in [37]. The time complexity of this part is $\widetilde{O}(1)$ in real-life graphs, as shown in [37].

For maintaining the random walk samples in evolving graphs, a straightforward approach is to resample all random walks. However, this is obviously not efficient. Below, we develop a novel and efficient technique to maintain these random walk trajectories after an edge is updated, without needing to resample random walks from scratch.


V1
V2
V3
V4
V5
V6
V7
V8
V9
V1 -> V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V2 -> V3 -> V2 -> V1 -> V4 -> V5 -> V4 -> V6
V2 -> V3 -> V4 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 ->V6
V3 -> V4 -> V3 -> V2 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 -> V6
V4 -> V3 -> V4 -> V5 -> V4 -> V5 -> V1 -> V5 -> V4 -> V6
V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V5 -> V4 -> V5 -> V4 -> V6
V7 -> V9 -> V7 -> V9 -> V7 -> V8
V9 -> V7 -> V9 -> V7 -> V9 -> V6
V5 -> V4 -> V3 -> V4 -> V5 -> V4 -> V5 -> V4 -> V6
V5 -> V4 -> V3 -> V4 -> V5 -> V4 -> V5 -> V4 -> V6 -> V7 -> V8

(a) Graph $\mathcal{G}$

(b) An illustrative example of RWIndex

(c) Update index after adding landmark v_3 (truncate the random walk)

(d) Update index after removing landmark v_6 (restart the random walk)

Figure 1: Illustration of RWIndex **and landmark node update.**

3.2 Maintaining RWIndex

Transforming an edge update to landmark nodes update. The challenge of updating $\mathcal{V}_l$-absorbed random walk trajectories lies in ensuring that the random walks remain a *valid sample* drawn from the updated graph. Due to graph updates, previously generated random walk trajectories may no longer be valid for the updated graph. We circumvent this issue by transforming the edge update into updating of landmark nodes.

Specifically, when there is an edge (u, v) update (no matter insertion or deletion), we set the two endpoints u and v as *new landmark nodes* if they are not the landmark nodes in the original graph $\mathcal{G}^t$. Note that adding a new landmark node involves moving a node from the set $\mathcal{U}$ to $\mathcal{V}_l$, keeping the overall node set unchanged. As the landmark node set $\mathcal{V}_l$ expands, the $\mathcal{V}_l$-absorbed random walks become shorter, without altering their trajectories before they reach the updated landmark node set. Thus, in this step, we only need to truncate certain random walk samples once they traverse these two *new landmark nodes*. After that, we remove the two *new landmark nodes* u and v from $\mathcal{V}_l$ for the updated graph $\mathcal{G}^{t+1}$ (i.e., the set of landmark nodes remains the same). Then, on the updated graph $\mathcal{G}^{t+1}$, we draw several new $\mathcal{V}_l$-absorbed random walk samples starting from u and v. These new random walk samples are then concatenated with the previous random walk samples that terminated at u and v respectively.

Note that if both u and v are landmark nodes in $\mathcal{G}^t$, no updates are needed for the random walk samples. This is because in this case, all random walk trajectories avoid traversing the edge (u, v) (since u and v are landmarks), thus the update of (u, v) does not affect the random walk samples. If only the node u (similar for v) is a landmark, we simply add v as a *new landmark node*. The processing procedure remains the same as when introducing two new landmark nodes. Compared to the method of resampling all random walks from every node in $\mathcal{U}$, our approach is much more efficient, as it draws several random walks only from *two landmark nodes*. Below, we detail the updating procedure of adding and deleting a landmark node.

Adding a landmark node v**.** We begin by examining the addition of a landmark node v from $\mathcal{U}$ to $\mathcal{V}_l$. The updated landmark node set (remaining node set) is denoted as $\mathcal{V}_l^{t+1}$ ($\mathcal{U}^{t+1}$) , while the previous set is $\mathcal{V}_l^t$ ($\mathcal{U}^t$). When a landmark node v is added, the updated sets are $\mathcal{V}_l^{t+1} = \mathcal{V}_l^t \cup \{v\}$ and $\mathcal{U}^{t+1} = \mathcal{U}^t \setminus \{v\}$. The addition of vertex v into the landmark node set is expected to result in earlier termination of some random walk samples. Each random walk passing through v will terminate immediately upon reaching this node. Therefore, the algorithm is quite simple, as depicted in Algorithm 1. Let $\mathcal{R}^t$ be the RWIndex of $\mathcal{G}^t$. The algorithm truncates all random walks in $\mathcal{R}^t$ that pass through v the first time they encounter v. The updated random walks are then output as $\mathcal{R}^{t+1}$.

Algorithm 1: RWIndex-add-landmark

```
Input: Graph $\mathcal{G}^t$, previous random walks $\mathcal{R}^t$, new landmark node $v$
Output: updated random walks $\mathcal{R}^{t+1}$
for each random walk in $\mathcal{R}^t$ that passes through $v$ do
    Shorten the random walk at the first time it hits $v$;
return $\mathcal{R}^t$;
```

Next, we analyze the correctness and time complexity of the landmark addition operation. Intuitively, after moving a node from $\mathcal{U}$ to $\mathcal{V}_l$, we must update the stored random walks to ensure they are sampled from the updated graph structure. As described in [37], the distribution of the $\mathcal{V}_l$-absorbed random walks is depicted by the matrix $\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}$. Specifically, let $\tau_{\mathcal{V}_l}[s, u]$ denote the expected number of steps for a $\mathcal{V}_l$-absorbed random walk from s that passes u. We can derive that the s, u-th element of $\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}$ is $\mathbf{e}_s^T \mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1} \mathbf{e}_u = \frac{\tau_{\mathcal{V}_l}[s,t]}{d_u}$ [37]. Consequently, we first express $\mathbf{L}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}^{-1}$ in terms of $\mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1}$. Then, we derive the following lemma that establishes the relationship between the distribution of $\mathcal{V}_l^{t+1}$-absorbed random walks and the distribution of $\mathcal{V}_l^t$-absorbed random walks.

Lemma 3.1. *Let* $Pr[s \rightsquigarrow \mathcal{V}_l; v]$ *denote the probability that a* $\mathcal{V}_l$*-absorbed walk starts from* s *passes* v *before hitting* $\mathcal{V}_l$*, we have:* $\tau_{\mathcal{V}_l^{t+1}}[s, u] = \tau_{\mathcal{V}_l^t}[s, u] - \tau_{\mathcal{V}_l^t}[v, u] Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$.

Proof. According to the block matrix inverse formula, suppose that $S = D - CA^{-1}B$, we have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}BS^{-1}CA^{-1} & -A^{-1}BS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{bmatrix}. \tag{4}$$

By substituting A with $\mathbf{L}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}^{-1}$ in this formula, $\mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1}$ can be expressed as:

$$\mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} = \begin{bmatrix} \mathbf{L}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}^{-1} + \cdots & \cdots \\ \cdots & \mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v \end{bmatrix}.$$

Here, we retain the v-th row and column of $\mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1}$ in their original form and express the remaining $|\mathcal{U}^{t+1}|$ rows and columns using the v-th row and column. According to Eq. (3.4), we have $S^{-1} = \mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v$. Then, $-CA^{-1}$ and $-A^{-1}B$ is the v-th row and column of $\mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1}$, divided by S^{-1}. Thus, we can derive that:

$$\mathbf{e}_s^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_u = \mathbf{e}_s^T \mathbf{L}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}^{-1} \mathbf{e}_u + \frac{\mathbf{e}_s^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v}{\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v} \mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_u.$$

Additionally, $Pr[s \rightsquigarrow \mathcal{V}_l^t; v] = \frac{\tau_{\mathcal{V}_l^t}[s,v]}{\tau_{\mathcal{V}_l^t}[v,v]} = \frac{\mathbf{e}_s^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v}{\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t\mathcal{U}^t}^{-1} \mathbf{e}_v}$. Furthermore, we have $\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1} \mathbf{e}_u = \frac{\tau_{\mathcal{V}_l^t}[v,u]}{d_u}$. The first equality holds because the expected number of passes through node v in a $\mathcal{V}_l^t$-absorbed random walk starting from s is equal to the expected number of times the random walk returning to v, multiplied by the probability that the random walk will pass v. The lemma is established by combining these three equations. □

Based on Lemma 3.1, we are able to verify the correctness and analyze the time complexity of the landmark addition operation.

Lemma 3.2. *Let $\mathcal{R}^{t+1}$ be the set of updated $\mathcal{V}_l^{t+1}$-absorbed random walks obtained by shortening each random walk in the original $\mathcal{V}_l^t$-absorbed random walks $\mathcal{R}^t$ according to Algorithm 1. Then, the distribution of the updated random walks in $\mathcal{R}^{t+1}$ is the same as that of the $\mathcal{V}_l^{t+1}$-absorbed random walks sampled from $\mathcal{G}^{t+1}$.*

Proof. After moving a landmark node v from $\mathcal{U}$ to $\mathcal{V}_l$, only the random walks that pass through v are affected. The portions of random walks that do not encounter v are sampled independently from the updated graph. If a random walk meets v, we shorten the random walk at that point. Lemma 3.1 shows that the distribution of a $\mathcal{V}_l^{t+1}$-absorbed walk from s in the updated graph is related to the probability of the walk passing through v. If it does not pass through v, the distribution remains the same as the $\mathcal{V}_l^t$-absorbed walk from s in the original graph. If it passes through v, the distribution is the original distribution minus the distribution of a $\mathcal{V}_l^t$-absorbed random walk starting from v. Due to the memoryless property of random walks [33], the distribution of the segment of the walk removed upon its first encounter with v is identical to that of a $\mathcal{V}_l^t$-absorbed walk starting from v. Thus, the lemma is established. □

Lemma 3.3. *The expected time complexity of adding a landmark node v is $h(v, \mathcal{V}_l^t) \sum_{s \in \mathcal{U}^t} Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$ for each sample, where $h(v, \mathcal{V}_l^t)$ is the hitting time from v to $\mathcal{V}_l^t$. In real-life graphs, this is approximately $\widetilde{O}(1)$ by our assumption.*

Proof. When adding a new landmark node v and considering a random walk sample $\mathcal{R}^t$, Algorithm 1 first checks all random walks passing through v. For a random walk starting from $s \in \mathcal{U}^t$, the expected random walk length is $\sum_{u \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[s, u]$. After v is moved from $\mathcal{U}^t$ to $\mathcal{V}_l^t$, the expected random walk length becomes $\sum_{u \in \mathcal{U}^{t+1}} \tau_{\mathcal{V}_l^{t+1}}[s, u]$. According to Lemma 3.1, the change in the expected random walk length is:

$$\sum_{u \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v, u] Pr[s \rightsquigarrow \mathcal{V}_l^t; v] = h(v, \mathcal{V}_l^t) Pr[s \rightsquigarrow \mathcal{V}_l^t; v].$$

Given that each set of random walks $\mathcal{R}^t$ contains $|\mathcal{U}^t|$ random walks in total, the overall difference is the sum over all starting nodes $s \in \mathcal{U}^t$, which is $h(v, \mathcal{V}_l^t) \sum_{s \in \mathcal{U}^t} Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$. Consequently, the expected time complexity of Algorithm 1 is bounded by the total difference in the random walk lengths, as each element of the trajectories must be erased.

In real-life graphs, it has been shown in [37] that $h(v, \mathcal{V}_l)$ is $\widetilde{O}(1)$ if the underlying graph is fast mixing and the landmark set $\mathcal{V}_l$ is selected properly. Furthermore, $\sum_{s \in \mathcal{U}^t} Pr[s \rightsquigarrow \mathcal{V}_l^t; v] = \frac{\sum_{s \in \mathcal{U}^t} \mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} \mathbf{e}_s}{\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} \mathbf{e}_v} \leq \frac{\sum_{s \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v,s]}{\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} \mathbf{e}_v} = \frac{h(v, \mathcal{V}_l^t)}{\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} \mathbf{e}_v} = \widetilde{O}(1)$. The last equality holds because $\mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} \mathbf{e}_v = O(\frac{1}{d_v}) = \widetilde{O}(1)$ in real-life graphs [61]. Thus, the time complexity simplifies to $\widetilde{O}(1)$. This completes the proof. □

Algorithm 2: RWIndex-remove-landmark

```
Input: Graph $\mathcal{G}^t$, a set of random walks $\mathcal{R}^t$, past landmark node
       set $\mathcal{V}_l^t$, a landmark node $v \in \mathcal{V}_l^t$
Output: updated random walks $\mathcal{R}^{t+1}$
for each random walk in $\mathcal{R}^t$ that terminates at node $v$ do
    Restart the random walk from $v$, until it hits $\mathcal{V}_l^t \setminus \{v\}$ ($\mathcal{V}_l^{t+1}$);
return $\mathcal{R}^t$;
```

Algorithm 3: RWIndex-add-edge

```
Input: Graph $\mathcal{G}^t$, past landmark node set $\mathcal{V}_l^t$, new edge $e = (u, v)$,
       a set of random walks $\mathcal{R}^t$
Output: updated random walks $\mathcal{R}^{t+1}$
if $u \notin \mathcal{V}_l^t$ then
    $\mathcal{R}^t \leftarrow$ RWIndex-add-landmark ($\mathcal{G}^t$, $\mathcal{R}^t$, $u$);
if $v \notin \mathcal{V}_l^t$ then
    $\mathcal{R}^t \leftarrow$ RWIndex-add-landmark ($\mathcal{G}^t$, $\mathcal{R}^t$, $v$);
Let $\mathcal{G}^{t+1}$ denote the updated graph;
if $u \notin \mathcal{V}_l^t$ then
    $\mathcal{R}^t \leftarrow$ RWIndex-remove-landmark ($\mathcal{G}^{t+1}$, $\mathcal{R}^t$, $\mathcal{V}_l^t$, $u$);
if $v \notin \mathcal{V}_l^t$ then
    $\mathcal{R}^t \leftarrow$RWIndex-remove-landmark ($\mathcal{G}^{t+1}$, $\mathcal{R}^t$, $\mathcal{V}_l^t$, $v$);
return $\mathcal{R}^t$;
```

Removing a landmark node v. In the scenario where a landmark node v is moved from $\mathcal{V}_l$ to $\mathcal{U}$, the updated sets are $\mathcal{V}_l^{t+1} = \mathcal{V}_l^t \setminus \{v\}$ and $\mathcal{U}^{t+1} = \mathcal{U}^t \cup \{v\}$. When a landmark node v is shifted from $\mathcal{V}_l$ to $\mathcal{U}$, the random walks in $\mathcal{R}^t$ are expected to be longer, as it becomes harder to reach $\mathcal{V}_l^{t+1}$ compared to $\mathcal{V}_l^t$. Observe that the random walks which do not terminate at v will remain unaffected, as they are still sampled independently and uniformly in the updated graph. Thus, we only need to modify the random walks that terminate at v. Algorithm 2 illustrates the pseudo-code for removing a landmark node. Specifically, for each random walk that terminates at node v, the random walk restarts from v and continues until it hits $\mathcal{V}_l^{t+1}$. We update these random walk samples by concatenating the previous truncated random walk trajectory and the newly-sampled random walk trajectory. To analyze the correctness and time complexity of this algorithm, we first present a lemma similar to Lemma 3.1.

Lemma 3.4. *Let $Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$ denote the probability that a $\mathcal{V}_l^t$-absorbed walk starting from s hits $\mathcal{V}_l^t$ by node v, we have: $\tau_{\mathcal{V}_l^{t+1}}[s, u] = \tau_{\mathcal{V}_l^t}[s, u] + \tau_{\mathcal{V}_l^{t+1}}[v, u] Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$.*

Proof. According to the block matrix inverse formula, suppose that $S = D - CA^{-1}B$, we have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}BS^{-1}CA^{-1} & -A^{-1}BS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{bmatrix},$$

By substituting A with $\mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1}$ in this formula, $\mathbf{L}_{\mathcal{U}^{t+1} \mathcal{U}^{t+1}}^{-1}$ can be expressed as:

$$\mathbf{L}_{\mathcal{U}^{t+1} \mathcal{U}^{t+1}}^{-1} = \begin{bmatrix} \mathbf{L}_{\mathcal{U}^t \mathcal{U}^t}^{-1} + \cdots & \cdots \\ \cdots & \mathbf{e}_v^T \mathbf{L}_{\mathcal{U}^{t+1} \mathcal{U}^{t+1}}^{-1} \mathbf{e}_v \end{bmatrix},$$

Here, we retain the v-th row and column of $\mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}$ in their original form and express the remaining $|\mathcal{U}^t|$ rows and columns using the v-th row and column. According to Eq. (3.4), we have $S^{-1} = \mathbf{e}_v^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_v$. Then, $-CA^{-1}$ and $-A^{-1}B$ is the v-th row and column of $\mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}}$, divided by S^{-1}. Thus, we can derive that:

$$\mathbf{e}_s^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_u = \mathbf{e}_s^T \mathbf{L}^{-1}_{\mathcal{U}^{t}\mathcal{U}^{t}} \mathbf{e}_u + \frac{\mathbf{e}_s^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_v}{\mathbf{e}_v^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_v} \mathbf{e}_v^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_u.$$

Moreover, $Pr[s \rightsquigarrow \mathcal{V}_l^t(v)] = Pr[s \rightsquigarrow \mathcal{V}_l^{t+1}; v] = \frac{\mathbf{e}_s^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_v}{\mathbf{e}_v^T \mathbf{L}^{-1}_{\mathcal{U}^{t+1}\mathcal{U}^{t+1}} \mathbf{e}_v}$ and also, $\mathbf{e}_s^T \mathbf{L}^{-1}_{\mathcal{U}^{t}\mathcal{U}^{t}} \mathbf{e}_u = \frac{\tau_{\mathcal{V}_l^t}[s,u]}{d_u}$. The first equality holds because the probability a $\mathcal{V}_l^t$-absorbed walk from s hits $\mathcal{V}_l^t$ by v is equal to the probability a $\mathcal{V}_l^{t+1}$-absorbed walk from s passes through v before hitting $\mathcal{V}_l^{t+1}$. Similar to the proof of Lemma 3.1, the lemma can be proved by combining these three equations. □

By Lemma 3.4, we can prove the correctness and analyze the time complexity of our algorithm.

Lemma 3.5. *Let $\mathcal{R}^{t+1}$ be the set of updated $\mathcal{V}_l^{t+1}$-absorbed random walks obtained by extending the $\mathcal{V}_l^t$-absorbed random walks $\mathcal{R}^t$ according to Algorithm 2. Then, the distribution of the updated random walks $\mathcal{R}^{t+1}$ is the same as that of the $\mathcal{V}_l^{t+1}$-absorbed random walks sampled from $\mathcal{G}^{t+1}$.*

Proof. For the random walks in $\mathcal{R}^t$ that do not terminate at v, moving the landmark node v from $\mathcal{V}_l$ to $\mathcal{U}$ will not influence them. According to Lemma 3.4, for the random walks that terminate at v, the new distribution is the old distribution plus the distribution of a $\mathcal{V}_l^{t+1}$-absorbed walk starting from v in the new graph. For the restart segment of the random walk in Algorithm 2, it is sampled uniformly and independently in the updated graph. Due to the memoryless property of random walks [33], the distribution of the updated random walks $\mathcal{R}^{t+1}$ is the same as that of $\mathcal{R}^t$, which establishes the lemma. □

Lemma 3.6. *The expected time complexity of removing a landmark node v is $h(v, \mathcal{V}_l^{t+1}) \sum_{s\in\mathcal{U}^{t+1}} Pr[s \rightsquigarrow \mathcal{V}_l^{t+1}(v)]$ for each sample. In real-life graphs, this is approximately $\widetilde{O}(1)$ by our assumption.*

Proof. When removing a landmark node v and considering a random walk sample $\mathcal{R}^t$, Algorithm 2 restarts all random walks terminating at v. For a random walk starting from $s \in \mathcal{U}^t$, similar to the proof of Lemma 3.1, according to Lemma 3.4, the change in the expected random walk length is:

$$\sum_{u\in\mathcal{U}^{t+1}} \tau_{\mathcal{V}_l^{t+1}}[v,u] Pr[s \rightsquigarrow \mathcal{V}_l^t(v)] = h(v, \mathcal{V}_l^{t+1}) Pr[s \rightsquigarrow \mathcal{V}_l^t(v)].$$

Given that each set of random walks $\mathcal{R}^{t+1}$ contains $|\mathcal{U}^{t+1}|$ random walks in total, the overall difference is the sum over all starting nodes $s \in \mathcal{U}^{t+1}$, which is $h(v, \mathcal{V}_l^{t+1}) \sum_{s\in\mathcal{U}^{t+1}} Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$. Consequently, the expected time complexity of Algorithm 2 can be bounded by the total difference in the random walk lengths, as we need to resample all such random walks and record their trajectories.

In real-life graphs, since removing a landmark node is the inverse operation of adding a landmark node, we can also bound the time complexity by $\widetilde{O}(1)$, as shown in Lemma 3.3. Thus, the time complexity simplifies to $\widetilde{O}(1)$. This establishes the lemma. □

It is important to note that by Lemma 3.3 and Lemma 3.6, both the landmark addition and removal operations can be performed in $\widetilde{O}(1)$ time, under the same assumption as in [37].

Adding an edge e. Armed with the two aforementioned landmark updating operations, when an edge $e = (u, v)$ is added to the current graph, we can convert this operation to at most two operations of landmark addition or removal. The algorithm is depicted in Algorithm 3. It first checks that whether u and v belong to the landmark node set $\mathcal{V}_l^t$. If they do not, we add them to the landmark node set by our landmark node addition operation described in Algorithm 1 (Lines 1-4). After that, the random walks that pass through u and v will be shortened the first time they hit u or v. Then, we remove the added landmark node from the landmark node set (Lines 6-9). This requires resampling some random walk samples from the newly-added landmark. According to Lemma 3.2 and Lemma 3.5, the random walks after these operations still follow the distribution of random walks in the updated graph. By Lemma 3.3 and Lemma 3.6, the expected time complexity is at most $h(u, \mathcal{V}_l) + h(v, \mathcal{V}_l)$ (if both u and v do not belong to the current $\mathcal{V}_l$ set), and is $\widetilde{O}(1)$ in real-life graphs.

Deleting an edge e. Interestingly, the operations of an edge deletion are the same as those of an edge insertion (the only difference is that the updated graph $\mathcal{G}^{t+1}$ in Line 5 of Algorithm 3 is different). Specifically, when an edge $e = (u, v)$ is deleted from the current graph, we conduct same operations by first adding u and v into the landmark node set and then removing them accordingly. The pseudo-code is exactly the same as that of Algorithm 3, and the time complexity is also the same. Below, we provide an example to illustrate the updating operations of RWIndex.

Example 1. *As shown in Fig. 1, Fig. 1(a) depicts an example graph $\mathcal{G}$ with 9 nodes and 14 edges, where v_6 and v_8 are the landmark nodes. The* RWIndex *is then built as shown in Fig. 1(b). The index contains 7 random walks starting from each node $u \in \mathcal{U}$ until they hit $\mathcal{V}_l$. Many important quantities can be estimated from such samples. When a landmark node v_3 is added, the index is updated as shown in Fig. 1(c). For simplicity, we use the random walk starting from v_5 as an example. As shown in Fig. 1(c), it terminates when it hits v_3, and the length changes from 9 to 3. When a landmark node v_6 is removed, the index is updated as shown in Fig. 1(d). The random walk starting from v_5 continues after passing v_6 and finally hits v_8, which is the only remaining landmark node.*

Comparison to the result in [18]. We note that in a theoretical paper [18], there is a similar landmark node addition operation for a dynamic vertex sparsifier. However, our approach significantly differs from theirs and is more practical. First, the algorithm in [18] does not support a landmark node removal operation. This limitation can cause the landmark node set to grow larger over time with continuous updates and queries. When a large set of nodes is updated, they re-compute the entire index at once. Although they prove that the amortized time complexity of each update is

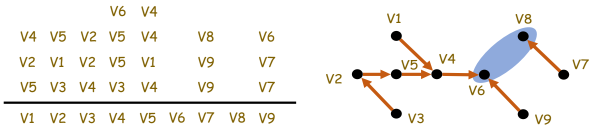
$\widetilde{O}(1)$, to ensure this bound, they set $|\mathcal{V}_l| = O(m)$ and use a dynamic graph sparsifier [1] to maintain the large Schur complement graph. This part is very challenging to implement efficiently as it is primarily designed for theoretical purposes. Our method supports the operation of landmark node removal, allowing control over the landmark node set size and ensuring that it remains a small group of easy-to-hit nodes. Additionally, they do not provide the exact distribution of the shortened random walk trajectories, whereas we precisely analyze the distribution of those updated random walks.

4 LOOP-ERASED RANDOM WALK INDEX MAINTENANCE

As shown in [37], the loop-erased random walk-based spanning forests sampling approach significantly outperforms random walk sampling for index construction in terms of both time and space efficiency. Instead of maintaining the two matrices $-\mathbf{L}^{-1}_{\mathcal{U}\mathcal{U}}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ as an index, an alternative approach is to use the sampled spanning forests as an index. However, unlike the random walk sampling technique, where our proposed algorithm in Section 3 allows efficient maintenance of random walk samples, maintaining spanning forests is much more challenging due to their intricate distributions and complex structural characteristics. In this section, we circumvent this challenge by maintaining the loop-erased random walks, instead of directly maintaining the spanning forests. Based on this idea, we propose a new *stack* index, called LEIndex, which compactly represents all loop-erased random walks and, consequently, can also represent spanning forests.

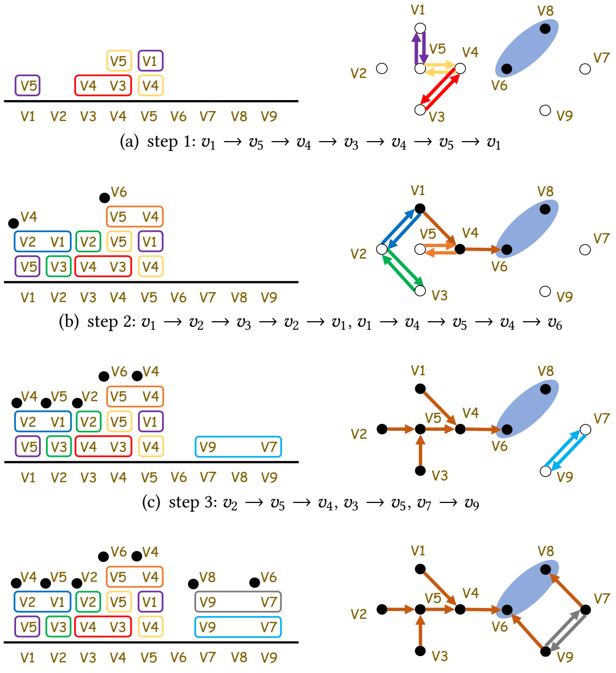
4.1 The proposed LEIndex index

The loop-erased random walk is a classic method, originally proposed by Wilson [64], for sampling spanning forests in graphs, which is widely used in graph analysis [6, 15, 26, 35–37, 46–48, 53]. We utilize the *stack representation* of loop-erased walks, as introduced in [64], which uses stacks to record the trajectories of loop-erased walks. The stack representation of random walks is a classical concept in probability theory [17]. These stacks do not follow the traditional LIFO (Last-In-First-Out) property, as elements can be accessed from both the top and bottom. Infinite stacks are generated from each node by independently sampling neighbors. Wilson observed in [64] that popping cycles from the bottom of these stacks results in finite stacks, independent of the traversal order. By applying the Wilson algorithm, we can therefore obtain a finite stack representation of a loop-erased walk. Fig. 2(a) provides an illustrative example of the *stack representation* of loop-erased walks, sampled from the graph shown in Fig. 1(a) with the root set $\{v_6, v_8\}$. Each node in $\mathcal{U}$ corresponds to a stack (e.g. $\mathcal{S}[v_1] = \{v_5, v_2, v_4\}$). From the bottom of the stacks, the i-th element of the stack for node u records the subsequent node of u in the loop-erased walk trajectory for the i-th step the loop-erased walk passes node u. Fig. 3 illustrates the process of building a *stack representation* of loop-erased walks. Specifically, a loop-erased walk maintains a spanning forest $\mathcal{F}$ during the random walk process, starting with $\mathcal{F}$ initialized as the root set $\mathcal{V}_l$ [37, 64]. Following a fixed ordering of nodes in $\mathcal{U}$ (In Fig. 3, it is the increasing node ordering from v_1 to v_9), it begins from the first node and simulates a random walk until it reaches $\mathcal{V}_l$. For each node u the walk passes, it adds the subsequent neighbor of u into $\mathcal{S}[u]$, as shown in Fig. 3. The random walk trajectory, with loops erased, is then added to $\mathcal{F}$. This process continues from the next node not in $\mathcal{F}$, stopping the random walk until it hits $\mathcal{F}$. The process terminates once all nodes in $\mathcal{U}$ have been added into $\mathcal{F}$. Wilson's original proof [64] demonstrates that this process samples a spanning forest with root $\mathcal{V}_l$ uniformly.



(a) An example of LEIndex (b) The corresponding spanning forest

Figure 2: An illustrative example of LEIndex. (a) For each node u, we store a stack $\mathcal{S}[u]$ (e.g. $\mathcal{S}[v_1] = \{v_5, v_2, v_4\}$); (b) A spanning forest is represented by the edges on top of each stacks.



(a) step 1: $v_1 \to v_5 \to v_4 \to v_3 \to v_4 \to v_5 \to v_1$

(b) step 2: $v_1 \to v_2 \to v_3 \to v_2 \to v_1, v_1 \to v_4 \to v_5 \to v_4 \to v_6$

(c) step 3: $v_2 \to v_5 \to v_4, v_3 \to v_5, v_7 \to v_9$

(d) step 4: $v_7 \to v_9, v_7 \to v_8, v_9 \to v_6$

Figure 3: The process of building a *stack representation* of loop-erased walk using the Wilson algorithm.

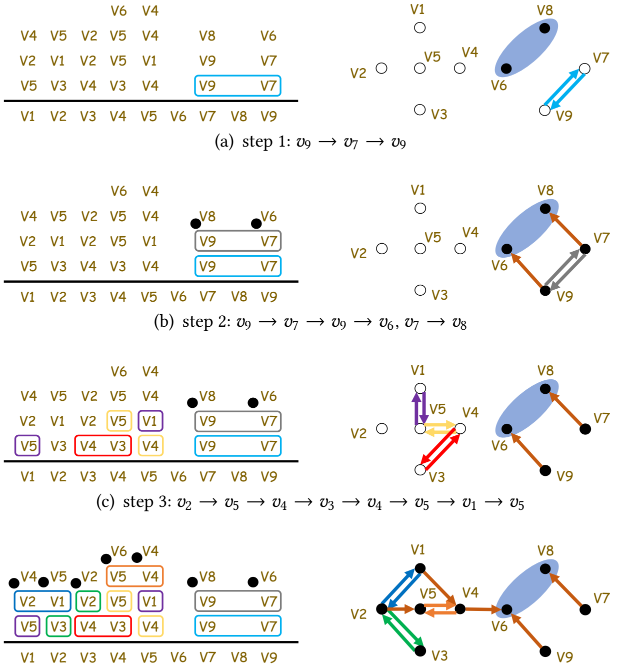
The *stack representation* of loop-erased walks possesses several advantageous properties: (i) Independence. Each stack is independent of the others, and each element in the stack of node u is merely a uniformly-sampled neighbor of node u. Each element within a stack is also independent of the others. (ii) Order invariance. The ordering of visiting the nodes is irrelevant. Formally, we can traverse the stacks following a fixed ordering. Fig. 4 provides an illustrative example of traversing the stacks in increasing order by node id (from v_1 to v_9). Initially, all elements in the stacks are marked as unvisited. Then, following the fixed ordering, similar to the sampling process of the Wilson algorithm, we maintain a spanning forest $\mathcal{F}$ (the nodes in $\mathcal{F}$ are marked black and the edges in $\mathcal{F}$ are marked red), initialized as the root set $\mathcal{V}_l$. Instead of sampling neighbors, for each node u encountered, we read the first unvisited neighbor from the bottom of u's stack to determine the next node to jump to until it hits $\mathcal{F}$. This process continues following the fixed ordering until all elements of the stacks are visited. Wilson showed [64] that the probability of sampling the stacks remains the same regardless of the fixed ordering. For the first node in the ordering, since all elements are unvisited, the distribution of the trajectory from that node until it hits $\mathcal{F}$ is identical to the distribution of a $\mathcal{V}_l$-absorbed walk trajectory from that node. Consequently, a loop-erased walk trajectory can encode the information of $|\mathcal{U}|$ $\mathcal{V}_l$-absorbed random walks, we can obtain $|\mathcal{U}|$ random walk trajectories from such set of stacks. For instance, the random walks in Fig. 1(b) can be derived from the *stack representation* of loop-erased walk illustrated in Fig. 2(a). If we traverse the stacks from v_9, we obtain the trajectory $v_9, v_7, v_9, v_7, v_9, v_6$; From v_4, we obtain the trajectory v_4, v_3, v_4, v_5, v_4, v_5, v_1, v_5, v_4, v_6, which exactly match the random walks from v_9 and v_4 shown in Fig. 1(b) respectively. Since $\mathcal{F} = \mathcal{V}_l$ initially, the distribution of such a trajectory from u in an *stack representation* is identical to the distribution of the $\mathcal{V}_l$-absorbed walk from u. (iii) Containing a uniform spanning forest. On the top of each stack, each node points to another node, and the graph consisting of these directed edges is acyclic, forming a spanning forest, as illustrated in Fig. 2(b).

We store ω *stack representation* of loop-erased walks as illustrated in Fig. 2(a) for the estimation of $-\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^{\dagger}$, based on the spanning forest estimators presented in [37]. We refer to this approach as LEIndex. The space complexity of LEIndex is $\omega \cdot \mathrm{Tr}((\mathbf{I}-\mathbf{P}_{\mathcal{U}\mathcal{U}})^{-1})$, which is strictly smaller than RWIndex [37]. The *stack representation* is space-efficient because the random walks share several common steps. On real-life graphs, $\mathrm{Tr}((\mathbf{I}-\mathbf{P}_{\mathcal{U}\mathcal{U}})^{-1})$ is $\widetilde{O}(n)$, and ω is $\widetilde{O}(1)$, which implies that such an index structure will not introduce too much additional space cost. For the query process, similar to RWIndex, we also do not need to compute $-\mathbf{L}_{\mathcal{U}\mathcal{U}}^{-1}\mathbf{L}_{\mathcal{U}\mathcal{V}_l}$ and $(\mathbf{L}/\mathcal{V}_l)^{\dagger}$ explicitly. Instead, we can efficiently answer a query using the spanning forest samples by employing the method proposed in [37].

4.2 Cycle decomposition of LEIndex

While LEIndex offers many advantages, it is challenging to maintain the *stack representation* without re-sampling the entire structure. Within these stacks, each element in one stack of a node may influence the random walk trajectory of another node, as it has a probability of being passed through by the other node in its trajectory of traversing the stacks. Note that when the landmark node set is modified, only a small part of elements will be influenced. The most challenging aspect is precisely identifying the segments of stack elements that require modification. For example, if we simply remove the elements of the stack corresponding to node u, the remaining stacks can no longer form a loop-erased walk trajectory. Similarly, if we remove the trajectory from u to the landmark node set $\mathcal{V}_l$, it will also violate the structure of the loop-erased walk trajectory. As a result, it cannot be efficiently maintained at a *node level* or a *stack level*. Interestingly, we discover that there is a cycle decomposition of the *stack representation* of the loop-erased walks through modifications to the Wilson algorithm [64], allowing for efficient maintenance of loop-erased walk trajectories at a *cycle level*.

Cycle decomposition of LEIndex. Observe the *stack representation* of loop-erased walks illustrated in Fig. 2(a). Except for the top



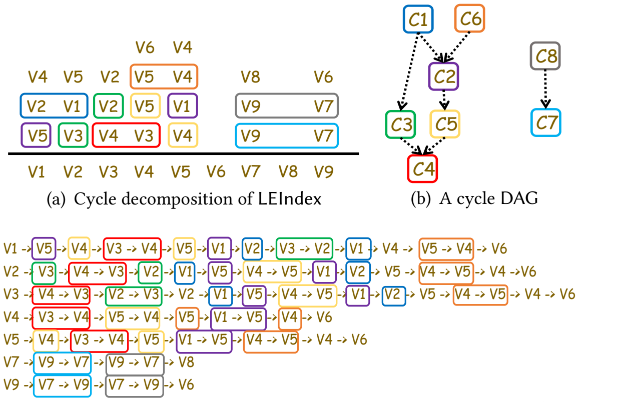
(a) step 1: $v_9 \to v_7 \to v_9$

(b) step 2: $v_9 \to v_7 \to v_9 \to v_6$, $v_7 \to v_8$

(c) step 3: $v_2 \to v_5 \to v_4 \to v_3 \to v_4 \to v_5 \to v_1 \to v_5$

(d) step 4: $v_5 \to v_4 \to v_5 \to v_4 \to v_6$, $v_3 \to v_2 \to v_3 \to v_2 \to v_1 \to v_2 \to v_5$, $v_1 \to v_4$

Figure 4: By traversing the stacks in decreasing order by node id, we can recover a loop-erased walk.


V6 V4
V4 V5 V2 V5 V4 V8 V6
V2 V1 V2 V5 V1 V9 V7
V5 V3 V4 V3 V4 V9 V7
V1 V2 V3 V4 V5 V6 V7 V8 V9
C1
C6
C2
C8
C3
C5
C7
C4
V1 -> V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V2 -> V3 -> V2 -> V1 -> V4 -> V5 -> V4 -> V6
V2 -> V3 -> V4 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 ->V6
V3 -> V4 -> V3 -> V2 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 -> V6
V4 -> V3 -> V4 -> V5 -> V4 -> V5 -> V1 -> V5 -> V4 -> V6
V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V5 -> V4 -> V5 -> V4 -> V6
V7 -> V9 -> V7 -> V9 -> V7 -> V8
V9 -> V7 -> V9 -> V7 -> V9 -> V6

(a) Cycle decomposition of LEIndex

(b) A cycle DAG

(c) Cycle decomposition of the random walks derived from the LEIndex

Figure 5: Cycle decomposition of LEIndex**. (a)** LEIndex **can be decomposed into cycles; (b) All cycles form a** DAG**; (c) The random walks derived by** LEIndex **can also be decomposed into cycles.**

elements of the stacks, which form a spanning forest, all other elements appear due to passing loops. When a loop occurs in the random walk process, the top of the stacks of the nodes through which the loop passes will be covered by the subsequent random walk process, *implicitly* storing a cycle within the stacks. Note that different orders of traversing the stacks may lead to different cycle entrances, but all the random walks derived from the loop-erased walks share these cycles. Consequently, the loop-erased walk trajectory can be decomposed into a set of cycles and a spanning forest stored at the top of the stacks. Specifically, for each element in the *stack representation* of the loop-erased walk, we formally store three additional values $(u, C, \mathsf{ID}[u, C])$: (i) u denotes that the element belongs to the stack of node u; (ii) C denotes the cycle C that contains that element; (iii) $\mathsf{ID}[u, C]$ denotes that the element is the $\mathsf{ID}[u, C]$-th element of the stack of u. Here, a cycle refers to a simple cycle in which no node appears twice. An illustrative example is provided in Fig. 5(a). In this example, the *stack representation* of a loop-erased walk contains eight cycles. For instance, for the first element of the stack of v_1, which points to v_5 in the loop-erased walk trajectory, we store $(v_1, C_2, 1)$ as additional information of the cycles. The cycle decomposition can also be applied to the random walks derived from the *stack representation* of the loop-erased walks, establishing a one-to-one correspondence between the cycles in the *stack representation* and those in the random walk trajectories. This correspondence is illustrated in Fig. 5(a) and Fig. 5(c), where the cycles are depicted in the same color in both the stacks and the random walk trajectories.

Moreover, we find that there is a *topological ordering* among the cycles in the stacks. This means that regardless of the node traversal order, a cycle on the top of the stacks can only be completely visited after the cycle on the bottom has been fully traversed. In a loop-erased walk $\mathcal{S}$, let the cycle set be denoted by $\mathcal{C} = \{C_1, \cdots, C_{|\mathcal{C}|}\}$. A directed acyclic graph (DAG) is a directed graph with no cycles, and it can define several topological orderings over a set. We find that the cycles naturally form a DAG, which is useful for designing dynamic algorithms.

Lemma 4.1. *Let $\mathcal{G}_C$ denote the graph where the node set is $\mathcal{C}$, for any two cycles C_i, C_j, there is an edge from C_i to C_j if and only if there exists a node that satisfies $u \in C_i$ and $u \in C_j$, and in the stack of node u, $\mathsf{ID}[C_i, u] = \mathsf{ID}[C_j, u] + 1$. Then, $\mathcal{G}_C$ is a* DAG *which records topological orderings over the cycle set $\mathcal{C}$.*

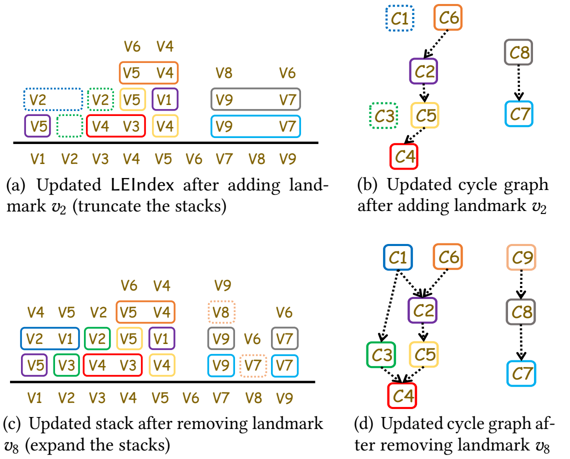
Proof. For two cycles $C_i, C_j \in \mathcal{C}$, we define that cycle C_j is *at the bottom of* C_i if there is a path from C_i to C_j in $\mathcal{G}_C$. Conversely, we define that cycle C_j is *on the top of* C_i if there is a path from C_j to C_i in $\mathcal{G}_C$. To prove that the cycles form a DAG, we need to show that no loop exists in the graph $\mathcal{G}_C$, meaning that a cycle C_i cannot be both *on the top of* and *at the bottom of* another cycle C_j simultaneously. In the process of traversing the *stack representation* of the loop-erased walk, an element of a stack can be visited only after all the elements beneath it have been visited. When such a situation occurs, given two nodes $u, v \in C_i$ and $u, v \in C_j$, suppose that for node u, $\mathsf{ID}[C_i, u] > \mathsf{ID}[C_j, u]$, and for node v, $\mathsf{ID}[C_i, v] < \mathsf{ID}[C_j, v]$. When u is visited at $\mathsf{ID}[C_j, u]$, it follows from cycle C_j to visit the element $(v, C_j, \mathsf{ID}[v, C_j])$. Since $\mathsf{ID}[C_i, v] < \mathsf{ID}[C_j, v]$, the element $(v, C_i, \mathsf{ID}[v, C_i])$ must be visited first. However, as the element $(v, C_i, \mathsf{ID}[v, C_i])$ belongs to cycle C_i, it will follow C_i back to node u. This implies that the elements $(u, C_j, \mathsf{ID}[u, C_j])$ and $(v, C_i, \mathsf{ID}[v, C_i])$ already form a cycle, which violates the definition of a cycle, as no node should appear twice. Thus, this situation is impossible, which will lead to a contradiction. This completes the proof. □

For two cycles $C_i, C_j \in \mathcal{C}$, we define a cycle C_j is *at the bottom of* C_i, if there is a path from C_i to C_j in $\mathcal{G}_C$. Conversely, a cycle C_j is *on the top of* C_i if there is a path from C_j to C_i in $\mathcal{G}_C$. The DAG built from the example *stack representation* of loop-erased walk in Fig. 5(a) is shown in Fig. 5(b). For instance, there is a path from C_6 to C_4, which means that C_4 must be completely visited before C_6 is fully traversed. Given such a topological ordering, when a cycle changes as the graph evolves, only the cycles *on the top of* that cycle can be influenced, while the cycles at the bottom of it remain unchanged. The remaining problem is how to construct such a DAG. Interestingly, we show that it can be obtained simultaneously with the sampling of the loop-erased walk through a slight modification of the Wilson algorithm.

Algorithm 4: Cycle decomposition of LEIndex

```
   Input: Graph $\mathcal{G}$, landmark node set $\mathcal{V}_l$
   Output: A set of stacks $\mathcal{S}[u]$ for $u \in \mathcal{U}$, a cycle graph $\mathcal{G}_C$, a spanning forest $\mathbf{F}$
1  $\mathcal{S}[u] \leftarrow \emptyset$ for $u \in \mathcal{U}$, $\mathcal{G}_C \leftarrow \emptyset$;
2  $\mathsf{Seen}[u] \leftarrow$ False, $\mathsf{Next}[u] \leftarrow \emptyset$ for $u \in \mathcal{V}$;
3  $\mathsf{InTree}[u] \leftarrow$ False for $u \in \mathcal{U}$, $\mathsf{InTree}[u] \leftarrow$ True for $u \in \mathcal{V}_l$;
4  for each node $u \in \mathcal{U}$ do
5      cur $\leftarrow u$;
6      while !InTree[cur] do
7          if Seen[cur] then
8              finder $\leftarrow$ cur, $C \leftarrow \emptyset$;
9              do
10                 $C$.pushback(finder, $\mathcal{S}$[finder].$size$());
11                 $\mathcal{S}$[finder].pushback($C$);
12                 Seen[finder] $\leftarrow$ False;
13                 finder $\leftarrow$ Next[finder];
14             while finder! = cur;
15             add cycle $C$ to $\mathcal{C}$;
16         Seen[cur] $\leftarrow$ True;
17         Next[cur] $\leftarrow$ $RandomNeighbor$(cur);
18         cur $\leftarrow$ Next[cur];
19     cur $\leftarrow u$;
20     while !InTree[cur] do
21         InTree[cur] $\leftarrow$ True, cur $\leftarrow$ Next[cur];
22 return $\mathcal{S}[u]$ for $u \in \mathcal{U}$, $\mathcal{G}_C$, $\mathsf{Next}[u]$ for $u \in \mathcal{V}$;
```

The pseudo-code of the cycle decomposition algorithm is outlined in Algorithm 4. At a high level, our algorithm conducts similar operations to the Wilson algorithm, while we detect cycles during the sampling process. Once a cycle is detected, it is recorded in the stacks by storing the cycle ID. The cycle DAG will be stored implicitly in the stacks without introducing too much additional cost. Specifically, similar to the Wilson algorithm used in [64], we use a vector $\mathsf{Next}[u]$ for $u \in \mathcal{V}$ to record the next node on top of the stack of u. During the sampling process, it will only record the top element of the stacks. Ultimately, it will store $n - |\mathcal{V}_l|$ edges, forming a spanning forest $\mathcal{F}$ with root set $\mathcal{V}_l$. We also use a vector $\mathsf{InTree}[u]$ for $u \in \mathcal{V}$ to record whether the node has been added to $\mathcal{F}$. Initially, $\mathsf{InTree}[u]$ is False for $u \in \mathcal{U}$ and True for $u \in \mathcal{V}_l$ (Line 3). Then, it follows the Wilson algorithm to simulate random walks from nodes that are not added to $\mathcal{F}$, until they hit $\mathcal{F}$ (Line 6-18). After the random walk stops, the loops are erased (Line 20-21). Specifically, we use an additional vector Seen to detect cycles. Initially, $\mathsf{Seen}[u]$ is set to False for $u \in \mathcal{V}$. For every node v encountered during the walk, $\mathsf{Seen}[v]$ is marked as True (Line 16). Thus, if we meet a node v with $\mathsf{Seen}[v] =$ True, a cycle is detected. We use an auxiliary variable finder to trace the cycles, and mark all nodes in that cycle with $\mathsf{Seen}[v] =$ False for the subsequent cycle detection (Lines 8-14). Then, the cycle C is added to the cycle set $\mathcal{C}$ (Line 15). For each cycle C, we store each node in the order they are visited, and we also record $\mathsf{ID}[u, C]$ which indicates that the cycle is recorded as the $\mathsf{ID}[u, C]$-th element of the stack of u (Line 10). In the stacks, instead of storing the next neighbor, we only need



(a) Updated LEIndex after adding landmark v_2 (truncate the stacks)

(b) Updated cycle graph after adding landmark v_2

(c) Updated stack after removing landmark v_8 (expand the stacks)

(d) Updated cycle graph after removing landmark v_8

Figure 6: LEIndex **maintenance after landmark node update.**

Algorithm 5: LEIndex-add-landmark

```
Input: Graph $\mathcal{G}$, landmark node set $\mathcal{V}_l$, new landmark node $v$, a loop-erased walk
       trajectory $\mathcal{S}^t$, a spanning forest $\mathcal{F}^t$ stored in a vector Next, cycle set $C^t$
Output: Updated loop-erased walk trajectory $\mathcal{S}^{t+1}$, updated spanning forest $\mathcal{F}^{t+1}$,
        updated cycle set $C^{t+1}$
1  StackSize$[u]$ ← $\mathcal{S}^t[u]$.size() for $u \in \mathcal{U}^t$;
2  $C$ ← the first cycle in $\mathcal{S}^t[u]$;
3  $Q \leftarrow \emptyset$, $Q$.push($C$);
4  visitedCycle$[C]$ ← False for $C \in C^t$;
5  while $Q$ is not empty do
6      $C \leftarrow Q$.pop();
7      visitedCycle$[C]$ ← True, $C^t \leftarrow C^t \setminus \{C\}$;
8      for each node $u \in C$ do
9          if ID$[u, C]$ < StackSize$[u]$ then
10             Next$[u]$ ← next node of $u$ in cycle $C$;
11             StackSize$[u]$ ← ID$[u, C]$;
12         $C_{next} \leftarrow \mathcal{S}^t[u][\text{ID}[u, C] + 1]$;
13         if !visitedCycle$[C_{next}]$ then
14             $Q$.push($C_{next}$);
15 for $u \in \mathcal{U}^{t+1}$ do
16     $\mathcal{S}^t$.erase(StackSize$[u]$, $\mathcal{S}^t[u]$.size());
17 return $\mathcal{S}^t$, Next, $C^t$;
```

to store the cycle ID since the neighbor is recorded in the cycles (Line 11). Notice that the DAG $\mathcal{G}_C$ is implicitly stored in the stacks $\mathcal{S}$, as shown in Fig. 5(a) and Fig. 5(b). For each cycle C in C, we can identify the out-neighbors of C by tracing all nodes u in the cycle, where its neighbor cycles are stored in $\mathcal{S}[u][\text{ID}[u, C] + 1]$. The process of building the cycle graph is illustrated in Fig. 3 where cycles are depicted by different color. Since these additional operations do not introduce significant additional cost, the time complexity of the modified algorithm remains the same as the original Wilson algorithm, i.e., $O(\text{Tr}((\mathbf{I} - \mathbf{P}_{\mathcal{U}\mathcal{U}})^{-1}))$ [37]. For real-life graphs, it takes $\widetilde{O}(n)$ time to build our LEIndex index.

4.3 Maintaining LEIndex

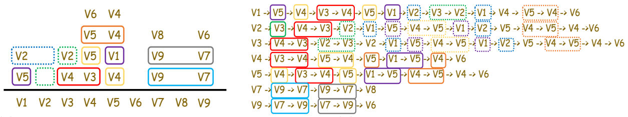
Transforming an edge update to landmark nodes update. Similar to RWIndex, updating the *stack representation* requires ensuring that the loop-erased walk remain a *valid sample* drawn from the updated graph. Due to graph updates, previously generated *stack representation* may no longer be valid for the updated graph. We also circumvent this issue by transforming the edge update into updating of landmark nodes.

Specifically, when there is an edge (u, v) update (no matter insertion or deletion), we set the two endpoints u and v as *new landmark nodes* if they are not the landmark nodes in the original graph $\mathcal{G}^t$. As the landmark node set $\mathcal{V}_l$ expands, the *stack representation* become smaller, but only a small part of the stacks need to be modified. We remove the cycles that are on top of the first cycles in the stacks of the *new landmark nodes.* After that, we remove the two *new landmark nodes* u and v from $\mathcal{V}_l$ for the updated graph $\mathcal{G}^{t+1}$. Then, on the updated graph $\mathcal{G}^{t+1}$, we continue the loop-erased walk process from u and v, until it obtains a spanning forest on top of the stacks. The new elements in the stacks are then concatenated with the previous stacks. Below, we detail the updating procedure of adding and deleting a landmark node.

Adding a landmark node v**.** After moving a landmark node v from $\mathcal{U}$ to $\mathcal{V}_l$, we need to ensure that the updated $\mathcal{S}^{t+1}$ is sampled according to the updated graph $\mathcal{G}^{t+1}$. As discussed before, instead of updating the stacks at the stack level or node level, our method benefits from updating the stacks at the cycle level. We aim to complete the update by exploring only a small, necessary part of the stack $\mathcal{S}^t$. Since v is moved from $\mathcal{U}$ to $\mathcal{V}_l$, the stack of v should be removed. This will influence the cycles stored in the stack of v. In the DAG $\mathcal{G}_C$, if we remove cycle C, all cycles *on the top of* C must also be removed, as they can be visited only after C is fully traversed. The top element of stack u will be replaced by the element corresponding to the last cycle removed from stack $\mathcal{S}[u]$.

The pseudo-code is outlined in Algorithm 5. At a high level, our algorithm conducts BFS traversal over $\mathcal{G}_C$ from the first cycle stored in $\mathcal{S}[v]$ and remove all the cycles visited. Considering the data structure we used in Algorithm 4 to implicitly store $\mathcal{G}_C$ in the *stack representation* $\mathcal{S}^t$, compared to maintaining random walks where we need to find the precise position each random walk passes through a specific node v, here we only need to traverse the stacks from v. Our goal is to remove all the cycles *on the top of* the first cycle in $\mathcal{S}[u]$. Thus, we perform a BFS traversal over the cycle graph $\mathcal{G}_C$. Starting from the first cycle C_0 in $\mathcal{S}^t[u]$ (Line 2), the algorithm uses a queue Q and a boolean vector visitedCycle to implement the BFS process (Line 3-4). It traverses all the cycles *on the top of* C_0 in $\mathcal{G}_C$ (Line 5-14). For each stack $\mathcal{S}[u]$, in addition to keeping cycles, we maintain an extra element as the Next$[u]$. Thus, we use a vector StackSize to record the current top of each stack. For each node u in a visited cycle, if ID$[u, C]$ is smaller than StackSize$[u]$, we update both Next$[u]$ and StackSize$[u]$ to ensure that the element corresponding to the last removed cycle in $\mathcal{S}[u]$ is retained for Next$[u]$ (Lines 9-11). After that, we add the neighbors of C into the queue if they have not been visited (Lines 12-14). We continue this process until the queue is empty. When the process terminates, the Next vector records a spanning forest, and we erase all the removed elements from $\mathcal{S}^t$ to obtain $\mathcal{S}^{t+1}$ (Line 15-16).

We analyze the correctness and time complexity of the landmark addition operation in the following lemmas. Specifically, we prove the correctness by establishing a one-to-one correspondence between the updated loop-erased walks and the updated random walks.


V6 V4
V5 V4
V8 V6
V2
V2
V5
V1
V9 V7
V5
V4 V3
V4
V9 V7
V1 V2 V3 V4 V5 V6 V7 V8 V9
V1 -> V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V2 -> V3 -> V2 -> V1 -> V4 -> V5 -> V4 -> V6
V2 -> V3 -> V4 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 ->V6
V3 -> V4 -> V3 -> V2 -> V3 -> V2 -> V1 -> V5 -> V4 -> V5 -> V1 -> V2 -> V5 -> V4 -> V5 -> V4 -> V6
V4 -> V3 -> V4 -> V5 -> V4 -> V5 -> V1 -> V5 -> V4 -> V6
V5 -> V4 -> V3 -> V4 -> V5 -> V1 -> V5 -> V4 -> V5 -> V4 -> V6
V7 -> V9 -> V7 -> V9 -> V7 -> V8
V9 -> V7 -> V9 -> V7 -> V9 -> V6

(a) Updated *stack representation* (b) Updated random walks

Figure 7: Updated stacks and updated random walks.

Lemma 4.2. *$\mathcal{S}^{t+1}$ is still uniform in the updated graph $\mathcal{G}^{t+1}$ after adding a landmark node v.*

Proof. Recall that given a loop-erased walk trajectory, there is a set of $|\mathcal{U}|$ $\mathcal{V}_l$-absorbed random walks that can be derived from it. The derived random walks have the same distribution as an independent set of $|\mathcal{U}|$ $\mathcal{V}_l$-absorbed random walks from each node $u \in \mathcal{U}$. Based on Lemma 3.2, these random walks can be updated by shortening them exactly when they hit v. We prove the correctness of Algorithm 5 by establishing a one-to-one correspondence between the updated loop-erased walks and the updated random walks. As illustrated in Fig. 5(c), the derived random walks can also be decomposed into cycles and paths in the spanning forest. Since all cycles *on the top of* the cycle containing node v must be removed, all cycles after v in the random walks derived from the updated *stack representation* of the loop-erased walk will also be removed. For the top of the stacks, we reuse the elements in the last cycle removed in those stacks, these elements will represent an edge in the updated spanning forest. For instance, in the updated *stack representation* is illustrated in Fig. 7, the random walks derived from the updated *stack representation* is the same as that of truncating the random walks illustrated in Fig. 1(b) after adding a landmark node v_2. Thus, the random walks derived from the updated *stack representations* are exactly truncated at v, which is the same as the updated random walks. The lemma is established. □

Lemma 4.3. *Let $Pr[s \rightsquigarrow \mathcal{V}_l; v]$ denote the probability that a $\mathcal{V}_l$-absorbed walk starts from s passes v. The time complexity of adding a landmark node v is $\sum_{s \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v, s] Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$. In real-life graphs, this can be simplified to $\widetilde{O}(1)$ by our assumption.*

Proof. In the process of Algorithm 5, the time complexity is upper bounded by the need to visit all the elements that are removed from the stacks. According to Lemma 3.1, since the expected size of the stack $\mathcal{S}^t[u]$ is $\tau_{\mathcal{V}_l^t}[u, u]$ [37], and the expected size of the stack $\mathcal{S}^{t+1}[u]$ is $\tau_{\mathcal{V}_l^{t+1}}[u, u]$. The average number of removed elements when v is added to the landmark node set is given by $\sum_{u \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v, u] Pr[u \rightsquigarrow \mathcal{V}_l^t; v]$. The summation is over all nodes $u \in \mathcal{U}^t$. Thus, the time complexity is strictly less than $\sum_{u \in \mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v, u] = h(v, \mathcal{V}_l^t)$, which can be further bounded by $\widetilde{O}(1)$ in real-life graphs [37]. Thus, the lemma is established. □

Removing a landmark node v. Moving a landmark node v from $\mathcal{V}_l$ to $\mathcal{U}$ is the inverse operation of adding a landmark node. In contrast to removing parts of loop-erased walks, we must resample certain sections. We aim to perform the update locally, avoiding the resampling of the entire loop-erased walk. Given the *stack representation* of the loop-erased walk walk $\mathcal{S}^t$, additional elements need to be added to the stacks. Since the order of visiting loops in the Wilson algorithm is irrelevant, we adjust the order to ensure that C^t has been visited first. Consequently, only the cycles in $C^{t+1} \setminus C^t$ needs to be further sampled. We use a stack Q to process the nodes, which correctly implements this ordering.

Algorithm 6: LEIndex-remove-landmark

```
Input: Graph $\mathcal{G}$, landmark node set $\mathcal{V}_l$, a landmark node $v$, a loop-erased walk trajectory
        $\mathcal{S}^t$, a spanning forest $\mathcal{F}^t$ stored in a vector Next, a cycle set $C^t$
Output: Updated loop-erased walk trajectory $\mathcal{S}^{t+1}$, updated spanning forest $\mathcal{F}^{t+1}$,
        updated cycle set $C^{t+1}$
1  Seen$[u]$ ← False, InTree$[u]$ ← True for $u \in \mathcal{V}$;
2  $Q \leftarrow \emptyset$;
3  InTree$[v]$ ← False, $Q$.push($v$);
4  while $Q$ is not empty do
5      $u \leftarrow Q.pop()$, cur ← $u$;
6      while cur $\notin \mathcal{V}_l \setminus \{v\}$ do
7          if Seen[cur] then
8              finder ← cur, $C \leftarrow \emptyset$;
9              do
10                 $C$.pushback(finder, $\mathcal{S}$[finder].$size()$);
11                 $\mathcal{S}$[finder].pushback($C$);
12                 Seen[finder] ← False;
13                 if InTree[finder] then
14                     InTree[finder] ← False, $Q$.push(finder);
15                 finder ← Next[finder];
16             while finder! = cur;
17             add cycle $C$ to $C^t$;
18         Seen[cur] ← True;
19         if !InTree[cur] then
20             Next[cur] ← $RandomNeighbor$(cur);
21         cur ← Next[cur];
22     cur ← $u$;
23     while cur $\notin \mathcal{V}_l \setminus \{v\}$ do
24         InTree[cur] ← True, cur ← Next[cur];
25 return $\mathcal{S}^t$, Next, $C^t$;
```

The pseudo-code of our algorithm is presented in Algorithm 6. At a high level, our algorithm continues the Wilson algorithm from node v. For the first time a node is visited, we reuse the neighbors stored on the top of the stacks. Otherwise, we sample random neighbors as the Wilson algorithm does. Specifically, given the current *stack representation* of loop-erased walk $\mathcal{S}^t$, we employ a queue Q to maintain the nodes that are marked as outside the spanning forest $\mathcal{F}^t$ and require further sampling. We start by adding node v into Q (Line 3). Then, similar to Algorithm 4, we continue the same process to simulate loop-erased walks until it hits $\mathcal{F}$ (Line 6-24). During this random walk process, if InTree$[u]$ is True for a node u encountered, the random walk will directly use the element on the top of the stack of node u. Originally, this element represents an edge in the sampled spanning forest; now it is reused to construct possible cycles (Lines 19-20). When a cycle is constructed, it indicates that the original edges in the spanning forest form a cycle. Thus, the InTree value of all the nodes in that cycle should be further marked False, requiring more steps of random walk until it hits $\mathcal{F}$ again. Therefore, besides recording the cycle, we set InTree$[u]$ = False for all nodes u in that cycle and add them into Q (Lines 13-14). If the current node u satisfies InTree$[u]$ = False, we sample a new neighbor of u instead of using the Next vector (Line 19-20). The algorithm terminates when the queue Q is empty, indicating that all nodes are added into the new spanning forest $\mathcal{F}^{t+1}$, and the loop-erased walk process is complete. The updated index $\mathcal{S}^{t+1}$, the spanning forest, and the cycle set C^{t+1} are returned (Line 25).

The following lemmas analyze the correctness and time complexity of the landmark deletion operation.

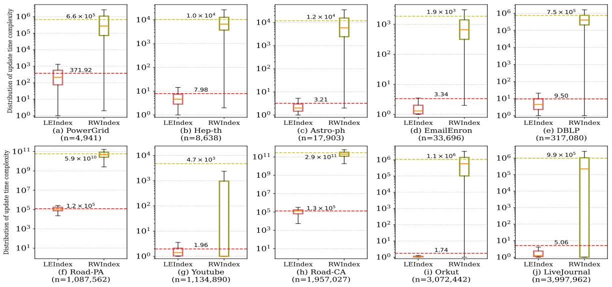
Lemma 4.4. *$\mathcal{S}^{t+1}$ is still uniform in the updated graph $\mathcal{G}^{t+1}$ after removing a landmark node v.*

Proof. During the process of Algorithm 6, once the queue Q is empty, all nodes are marked as InTree = True. Consequently, there is a spanning forest on the top of the stacks, with several loops formed beneath it, mirroring the finished state of the loop-erased walk process. Since v is now also a node in $\mathcal{U}^{t+1}$, $\mathcal{S}^{t+1}$ represents a valid *stack representation* of the loop-erased walk according to the graph structure $\mathcal{G}^{t+1}$. Additionally, all the elements in the stacks are sampled uniformly and independently as a random neighbor of the corresponding node. Therefore, $\mathcal{S}^{t+1}$ is uniformly distributed over all loop-erased walks on graph $\mathcal{G}^{t+1}$. □

Lemma 4.5. *Let $Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$ denote the probability that a $\mathcal{V}_l^t$-absorbed walk from s hits $\mathcal{V}_l^t$ by v. The expected time complexity of removing a landmark node v is $\sum_{s\in\mathcal{U}^{t+1}} \tau_{\mathcal{V}_l^{t+1}}[v,s]Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$. In real-life graphs, this can be simplified to $\widetilde{O}(1)$ by our assumption.*

Proof. Similar to Algorithm 5, the expected time complexity of Algorithm 6 is also bounded by the difference of the expected size of $\mathcal{S}^{t+1}$ and $\mathcal{S}^t$. According to Lemma 3.4, since the expected size of the stack $\mathcal{S}^t[u]$ is $\tau_{\mathcal{V}_l^t}[u,u]$, and the expected size of the stack $\mathcal{S}^{t+1}[u]$ is $\tau_{\mathcal{V}_l^{t+1}}[u,u]$. The expected number of deleted elements when v is removed from the landmark node set is given by $\sum_{s\in\mathcal{U}^{t+1}} \tau_{\mathcal{V}_l^{t+1}}[v,s]Pr[s \rightsquigarrow \mathcal{V}_l^t(v)]$. The summation is over all nodes $u \in \mathcal{U}^{t+1}$. Since the probability is strictly less than 1, the time complexity is strictly less than $\sum_{s\in\mathcal{U}^{t+1}} \tau_{\mathcal{V}_l^{t+1}}[v,s] = h(v,\mathcal{V}_l^{t+1})$, which can be further bounded by $\widetilde{O}(1)$ according to [35]. □

Theoretical characterization of the proposed algorithms. According to [37], under the raipd mixing and small diameter assumptions, both the proposed RWIndex and LEIndex have a $\widetilde{O}(n)$-size index and can be built in $\widetilde{O}(n)$ time. Based on the index, they can support a query in $\widetilde{O}(1)$ time. According to our analysis, the indices can handle an edge update in $\widetilde{O}(1)$ time under the same assumptions. In all situations, LEIndex has strictly smaller index space, index building time, query time and update complexity compared to RWIndex. To gain insight into the magnitude of the update time complexity involved (whether it is $\widetilde{O}(1)$), we also conducted experiments to compute the values of the expected time complexity as described in Lemma 3.3 and Lemma 4.3. Given that the addition and removal of landmarks are dual operations, the time complexity of moving v from $\mathcal{U}$ to $\mathcal{V}_l$ is equal to that of moving v from $\mathcal{V}_l$ to $\mathcal{U}$. We select the 10 highest degree nodes as $\mathcal{V}_l$ and compute the values of the reported quantities for updating 1000 uniformly sampled landmarks. We employed box plots to visualize the distribution of time overheads across 10 datasets described in Table 1. The results are illustrated in Fig. 8. On social networks (excluding PowerGrid, Road-PA and Road-CA, which are road networks), the values of LEIndex are less than 10, aligning with the expected $\widetilde{O}(1)$ time complexity. The values of RWIndex are often not very small, but they are still significantly smaller than the node size n. The potential reason is that the theoretical $\widetilde{O}(1)$ time complexity of RWIndex may incorporate a significant hidden *poly-log* factor within the $\widetilde{O}$ notation. On road networks, the values can even exceed n due to the absence of rapid mixing and small-world properties, which contradict our assumptions. We also note that the values of LEIndex are consistently much lower than those of RWIndex, suggesting superior updating performance. Moreover, even on road networks, LEIndex still shows a value much smaller than the node size n, further demonstrates the high efficiency of our LEIndex maintenance algorithm.



(a) PowerGrid (n=4,941)
(b) Hep-th (n=8,638)
(c) Astro-ph (n=17,903)
(d) EmailEnron (n=33,696)
(e) DBLP (n=317,080)
(f) Road-PA (n=1,087,562)
(g) Youtube (n=1,134,890)
(h) Road-CA (n=1,957,027)
(i) Orkut (n=3,072,442)
(j) LiveJournal (n=3,997,962)

Figure 8: Distribution of the expected update time complexity for RWIndex and LEIndex (the quantities $h(v,\mathcal{V}_l^t)\sum_{s\in\mathcal{U}^t} Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$ and $\sum_{s\in\mathcal{U}^t} \tau_{\mathcal{V}_l^t}[v,s]Pr[s \rightsquigarrow \mathcal{V}_l^t; v]$ reported in Lemma 3.3 and Lemma 4.3 respectively).

Adding or deleting an edge e. When adding or deleting a landmark node or an edge $e = (u,v)$, similar to Algorithm 3, we first add u and v to the landmark node set by removing a part of loop-erased walks. Subsequently, to ensure the size of the landmark node set remains at a reasonable level, we remove the added landmark node by resampling a small portion of the loop-erased walks. The expected time complexity for this operation remains $\widetilde{O}(1)$ in real-life graphs.

Example 2. *Fig. 6 provides an illustrative example of the dynamic update of the LEIndex as shown in Fig. 2. Suppose that v_2 is added as a landmark, as depicted in Fig. 6(a). When the landmark node v_2 is added, we first identify C_3 as the first cycle in the stack of v_2. We then perform a BFS traversal starting from C_2 over the cycle graph $\mathcal{G}_C$, as shown in Fig. 5(b), and remove the cycles that are on top of C_3, which is C_1 in this example. Finally, we designate the nodes in the last removed cycles as the new top of the stacks. For v_1, we make $v_2 \in C_1$ the top of the stack of v_1, and similarly, we make $v_2 \in C_3$ the top of the stack of v_3. The updated cycle graph $\mathcal{G}_C$ is illustrated in Fig. 6(b). When the landmark node v_8 is removed, we restart the loop-erased walk process from a $\mathcal{V}_l$-absorbed walk originating from v_8, reusing the random neighbors in the stacks. After constructing a new cycle C_9, it influences the stacks of both v_7 and v_8. This is illustrated in Fig. 6(c) and Fig. 6(d).*

Novelty of this paper compared to previous studies. In this paper, we focus on the problem of dynamically maintaining the indices proposed in [37], which represent the SOTA index-based method for effective resistance computation. The method in [37] builds on [35] by introducing multiple landmark nodes. Compared to related work [35, 37, 64], the key contributions of our paper are as follows: (i) To handle edge updates in the indices from [37], we propose a novel approach that transforms edge update operations into landmark node update operations. This idea is original and has not been explored in previous work, including [37]; (ii) Maintaining the loop-erased walk-based index is particularly challenging due to the complexity of the loop-erased walk trajectory. To address this, we propose a novel cycle decomposition technique that expresses the stack representation of loop-erased walks in terms of cycles and forests. We show that these cycles form a directed acyclic graph (DAG), enabling us to manage edge updates at the cycle level. Such intriguing and important properties of loop-erased walk trajectories were not identified in [37, 64]; (iii) For both the random walk-based and loop-erased walk-based indices, we present efficient algorithms capable of handling both edge insertions and deletions. We provide a rigorous theoretical analysis of the correctness and time complexity of these algorithms. Our approach maintains random walk and loop-erased walk samples by exploring only a small portion of the indices near the updated edge. In contrast, the random walk and loop-erased walk sampling techniques in [37] require recomputing the entire index to accommodate edge updates.

Generalizability of the proposed approaches. It is worth noting that our loop-erased walk maintenance technique is of independent interest and has the potential to accelerate various graph computation tasks that rely on loop-erased walks, such as personalized PageRank computation [36], electrical closeness centrality approximation [5, 49], and graph signal processing [46, 47]. For example, as demonstrated in [36], loop-erased walks can also be used to construct a personalized PageRank index. Since our technique inherently enables loop-erased walk samples to handle edge updates, the LEIndex framework can be applied to maintain personalized PageRank indices as well. Compared to the current SOTA method, FIRM [28], which is based on random walks, the advantages of loop-erased walk index maintenance over random walk-based index maintenance are similar to those highlighted in this paper. Initial experimental results for personalized PageRank maintenance will be presented in Section 5.2. Furthermore, for more expressive graph models, such as attributed graphs [71] and labeled property graphs [60], the definition of effective resistance is not yet well-established. However, while personalized PageRank and effective resistance are distinct concepts, both of them are grounded in random walks and graph matrices. Since personalized PageRank algorithms are supported by widely-used graph databases such as Neo4J [63] and graph processing systems like GraphX [23], our algorithms can likely be extended to these graph models in a similar manner. For instance, [38] discusses the implementation of PageRank algorithms in practical property graphs, and given that our algorithms share similar operations, such as iteratively querying neighbors, they can also be implemented in these systems. Similarly, the attributed random walk model discussed in [65] could be adapted to define effective resistances and design efficient algorithms for attributed graphs. These are potential directions for future work to extend the algorithms proposed in this paper.

5 EXPERIMENTS

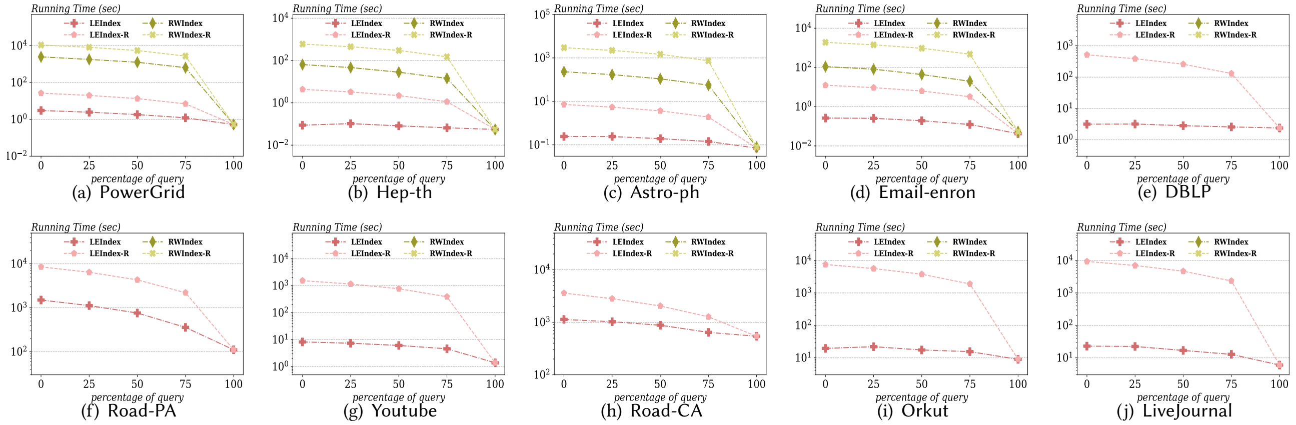
5.1 Experimental setup

Table 1: Datasets ($\bar{d}$: average degree; $\tau_{rw} = \vec{1}^T(\mathbf{I} - \mathbf{P}_{\mathcal{UU}})^{-1}\vec{1}$: time complexity of the building of RWIndex; $\tau_{le} = \mathrm{Tr}((\mathbf{I} - \mathbf{P}_{\mathcal{UU}})^{-1})$: time complexity of the building of LEIndex)

Dataset	n	m	$\bar{d}$	τ_{rw}	τ_{le}
PowerGrid	4,941	6,594	2.67	7.8×10^6	2.4×10^4
Hep-th	8,638	24,806	5.74	1.2×10^6	1.6×10^4
Astro-ph	17,903	196,972	22	2.1×10^6	2.3×10^4
Email-enron	33,696	180,811	10.73	1.6×10^6	4.5×10^4
DBLP	317,080	1,049,866	6.62	3.6×10^8	5.9×10^5
Road-PA	1,087,562	1,541,514	2.83	7.9×10^{11}	9.9×10^6
Youtube	1,134,890	2,987,624	5.27	1.1×10^8	1.8×10^6
Road-CA	1,957,027	2,760,388	2.82	3.4×10^{12}	2.3×10^7
Orkut	3,072,441	117,185,083	76.28	3.2×10^9	3.1×10^6
LiveJournal	3,997,962	34,681,189	17.35	4.3×10^9	5.1×10^6

Datasets. We utilize 10 real-life datasets that are widely adopted in previous studies on effective resistance computation [35, 37, 45, 66]. The detailed statistics of these datasets are shown in Table 1. PowerGrid, Road-PA and Road-CA are road networks, which are the hard cases reported in [37], while other datasets are social networks that are easier to handle. All these datasets can be obtained from public sources [32]. To measure the performance of different methods under a dynamic setting, we generate a query-update workload that consists of both queries and updates following [28]. For each dataset, we first randomly shuffle the order of edges and use the first 90% of edges to build the initial graph. Then, an update involves either (i) adding an edge uniformly at random from the remaining 10% of edges; (ii) deleting an edge uniformly at random from the current graph. We follow previous studies [28] in selecting a 90% proportion of edges in our experiments. Additionally, we conduct experiments using an initial graph constructed with the first 75% of the edges. We vary the proportion of updates to compare the performance of the algorithms under different dynamic scenarios. A workload with x% updates means that x% of the operations are updates, which have an equal probability of being either an edge addition or deletion, and $(1 - x)$% of the operations are queries, where the query node pairs are uniformly generated from the current graph structure. During the evolution process, the underlying graph structure is adjusted to ensure connectivity by skipping any graph snapshots that are not connected. We set the number of operations in a workload to 100.

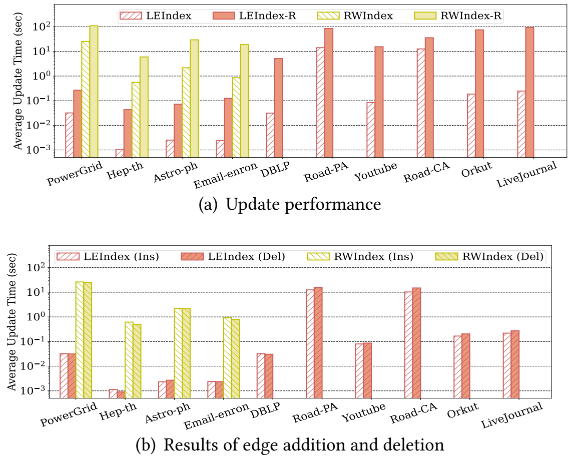
Methods. For the problem of dynamic index maintenance for effective resistance computation, we implement two proposed index maintenance methods, denoted by RWIndex and LEIndex, as described in Section 3 and Section 4 respectively. We also implement two static versions of these methods, which recompute the index from scratch using the state-of-the-art (SOTA) *static* methods proposed in [37] when an update occurs. These are denoted as RWIndex-R and LEIndex-R respectively. For all these methods, the sample size is set to $\frac{\log n}{\epsilon^2}$, and we set $\epsilon = 0.5$ by default, following [37]. Based on our assumption, the graph structure will retain the properties of real-life graphs during the evolution process. Thus, we maintain the same sample size throughout the dynamic process. For the query process, we also set the query sample size to $\frac{\log n}{\epsilon^2}$, in line with previous studies [37]. For RWIndex and LEIndex, we combine $\mathcal{V}_l$-absorbed push ($r_{max} = 10^{-3}$) with $\mathcal{V}_l$-absorbed walk sampling (BiPush proposed in [37]) to enhance query performance. To compute the ground-truth of the effective resistance value in each graph snapshot, we use the SOTA index-free approach BiPush as proposed



(a) PowerGrid (b) Hep-th (c) Astro-ph (d) Email-enron (e) DBLP

(f) Road-PA (g) Youtube (h) Road-CA (i) Orkut (j) LiveJournal

Figure 9: General performance of different methods with various query-update workloads



(a) Update performance

(b) Results of edge addition and deletion

Figure 10: Update performance of different methods

in [35]. We compare the quantity of the query result $\hat{r}(s,t)$ with the exact result $r(s,t)$ and use the relative error $\frac{|\hat{r}(s,t)-r(s,t)|}{r(s,t)}$ to measure the query quality. Following previous studies, we select the landmark node set using the Degree+ heuristic method Degree+ proposed in [37]. We set the landmark node set size $|\mathcal{V}_l| = 10$ for social networks and $|\mathcal{V}_l| = 100$ for road networks, in accordance with [37].

Experimental environment. We implement all the methods in C++. The experiments are conducted on a server with a 2.2 GHz Intel Xeon CPU and 128 GB of memory. We run all the experiments 10 times and report the average results.

5.2 Experimental results

Exp-1: Results with different query-update workloads. We then evaluate the general performance of the proposed methods under different query-update workloads. We vary x from 0 to 100 and compare the running time of RWIndex and LEIndex. We also include the resampling algorithms that recompute the index every time, denoted as RWIndex-R and LEIndex-R. The results are shown in Fig. 9. On the large datasets (e)-(j), RWIndex is out of memory as it requires storing the entire random walk trajectories. Thus, all the results of RWIndex in the six large datasets are omitted. When the percentage of query increases, the running time of all methods decrease. This indicates that the update time is larger than the query time. It can be seen that RWIndex and LEIndex are significantly faster than their resampling counterparts on all datasets. Notably, on LiveJournal, LEIndex-R takes 9,354 seconds, while LEIndex only takes 23 seconds, which is around 400 times faster. In general, LEIndex is always much faster than RWIndex. For example, when $x = 50$ on Astro-ph, RWIndex takes 40 seconds to process the workload, while LEIndex takes only 10^{-1} seconds. Although RWIndex theoretically exhibits $\widetilde{O}(1)$ time complexity, its practical performance is hindered by a large hidden *poly-log* factor within the $\widetilde{O}$ notion (also indicated in Fig. 8). However, the performance of LEIndex confirms our analysis that it has an $\widetilde{O}(1)$ complexity for updating, which only processes a small part of the graph. These results demonstrate the high efficiency of the proposed LEIndex method.

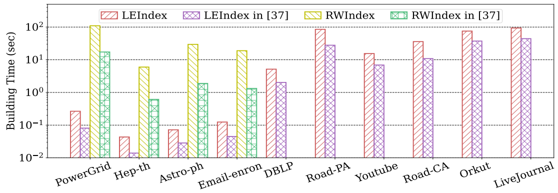
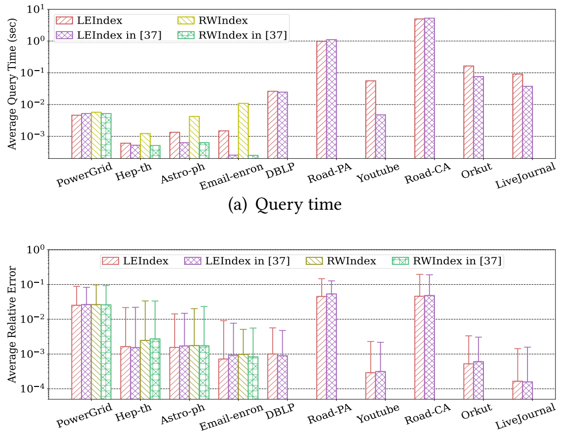


Figure 11: Index building time of different methods

Exp-2: Update performance. In this experiment, we only evaluate the index updating performance of different methods. Specifically, we employ 100 updates and report the average update time. The results are shown in Fig. 10(a). It can be seen that the update of LEIndex is much faster than both LEIndex-R and RWIndex. For example, on a large graph Orkut with 3 million nodes and 117 million edges, LEIndex takes only 10^{-1} seconds for each update, demonstrating its high efficiency. We also compare the update time for adding and deleting edges. It can be seen in Fig. 10(b) that the update times for edge addition and deletion are similar for both RWIndex and LEIndex, as we convert the operations of edge updates to landmark node updates. These results confirms our theoretical analysis in Sections 3 and 4.



(a) Query time

(b) Accuracy result

Figure 12: Query performance of different methods ($x = 50$)

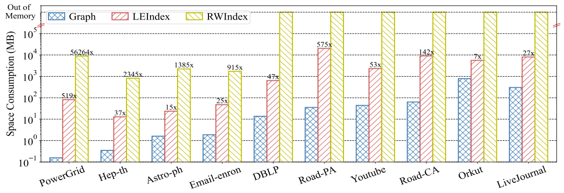
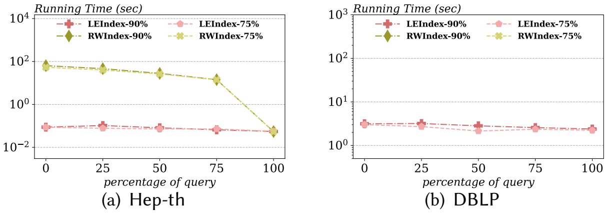


Figure 13: Memory consumption of different methods



(a) Hep-th

(b) DBLP

Figure 14: General performance of different algorithms with smaller initial graph size

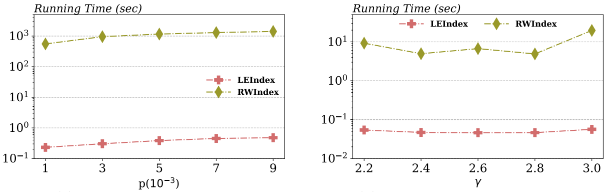
Exp-3: Index building time. Notice that the index structure RWIndex and LEIndex in this paper is very similar to that of [37], with the only difference being that we directly store the samples instead of the numerical values. As a result, the modified index structure can handle dynamic updates while the index structure in [37] can not. In this experiment, we compare the index building time of RWIndex and LEIndex, as well as their static versions in [37]. The results are reported in Fig. 11. As can be seen, the index building time of our index structures are slightly longer than those in [37], since our methods require additional operations such as retracing cycles and stroing auxiliary data structures to handle dynamic updates. On the other hand, the index building time of LEIndex is several orders of magnitude times lower than that of RWIndex. On the largest dataset LiveJournal, it takes only 10^2 seconds to build LEIndex, while RWIndex is out of memory on this dataset. These results further demonstrate the high efficiency of the proposed LEIndex technique.

Exp-4: Query performance. We evaluate the query performance of our approaches by performing 100 queries with an update rate 50%. We also include the query performance results based on the static index structures in [37]. Both the query time and accuracy results are compared, as illustrated in Fig. 12. As shown in Fig. 12(a), the average query times for RWIndex and LEIndex are comparable. They are both slightly higher than the static versions in [37], since we need to compute the required matrices at the query phase. On most datasets, the time required for this computation is negligible compared to the total query time. Furthermore, the results of accuracy, depicted using a box plot in Fig. 12(b), show that all methods yield similar accuracy results. The box represents the average relative error, and the lines indicate the maximum relative error. Both error results of LEIndex are comparable to that of RWIndex, aligning with previous studies [37]. Our findings indicate that LEIndex can achieve significant update time improvements while maintaining both the query accuracy and index building time compared to the SOTA method throughout the graph evolving process.

Exp-5: Memory consumption. Fig. 13 illustrates the memory consumption of all methods. We compare the memory usage of different indices relative to the graph size. Above each box, the space overhead of the indices compared to the graph size is indicated. As shown, RWIndex requires storing entire random walk trajectories, leading to substantial memory consumption. For instance, on the Hep-th dataset, the memory usage is 2345× the graph size, and on larger graphs, RWIndex exceeds available memory. In contrast, LEIndex only stores a compact stack structure that efficiently maintains all loop-erased walks, making it far more space-efficient than RWIndex. Across all datasets, the maximum memory consumption of LEIndex is approximately 10 GB, which is manageable on a standard PC. For most social networks, the space overhead is typically dozens of times larger than the size of the graph. These results indicate that our index is space efficient.

Exp-6: Results with smaller initial graph size. In previous experiments, we followed the approach in [28], dividing the datasets into two parts and randomly selecting the first 90% of edges to construct the initial graph by default. In this experiment, we also evaluate the performance of the algorithms with a smaller initial graph size. Specifically, we compare the overall performance of the proposed RWIndex and LEIndex algorithms when using 75% of the initial edges against 90%. Results for a small graph (Hep-th) and a large graph (DBLP) are shown in Fig. 14, with consistent findings across other datasets. As shown, the performance of our algorithms does not significantly change when starting with a smaller initial graph size. This is because the time complexities of the algorithms are not directly related to the graph size but rather to the graph spectral properties (see Lemma 3.3 and Lemma 4.3). As a result, overall performance may improve or decline slightly. However, LEIndex consistently demonstrates superior performance, even with a smaller initial graph size.

Exp-7: Results on synthetic graphs. In this experiment, we study the performance of our algorithms on synthetic datasets with varying parameters. We select the commonly used Erdős–Rényi model [11] and power law graph model [2] to generate synthetic graphs. For the Erdős–Rényi model, there are two parameters n and p, which control the graph size and the probability of an edge existence. We fix n as 10^5 and vary p from 1×10^{-3} to 9×10^{-3}. For the power-law graph model, there are also two parameters n and γ, which control the graph size and the power-law exponential factor.



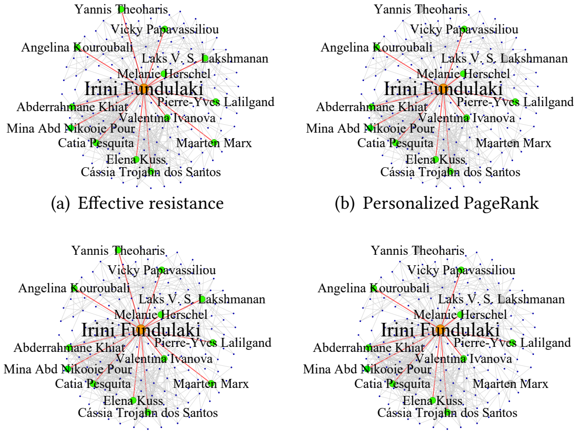
(a) Erdős–Rényi model (b) power law graph model

Figure 15: Results on synthetic graphs with varying parameters

We fix n as 10^5 and vary γ from 2 to 3. We set the update rate as 50% by default. The results of the general performance of RWIndex and LEIndex can be found in Fig. 15. It can be seen that when p becomes larger and γ becomes smaller (which indicates that the graphs exhibit faster mixing), the running time of our algorithms also become larger. This is consistent with our time complexity analysis. Among all generated graphs, LEIndex exhibits a distinguished performance compared to RWIndex.

Exp-8: Case studies. In this experiment, we perform a case study to evaluate the effectiveness of using effective resistance as a similarity measure. We utilize the DBLP graph from the database community, consisting of 62,150 nodes and 228,772 edges, which can be obtained from [32]. The DBLP graph is a co-authorship network where each node represents an author, and each edge represents a co-authorship. We compare effective resistance with other widely used similarity measures, including personalized PageRank [22], Jaccard similarity [43] and SimRank [29] for the task of link prediction. Specifically, we randomly remove 10% of the edges from the original graph and compute single-source similarities for the endpoints of the removed edges. For each node, we select the non-neighboring nodes with the top 100 highest similarity scores as the predicted co-authors, and measure accuracy by calculating the proportion of removed edges that are correctly predicted. The results show accuracy rates of 82.4%, 85.7%, 64.9% and 33.5% for effective resistance, personalized PageRank, Jaccard similarity and SimRank, respectively. Thus, effective resistance demonstrates strong predictive accuracy among these measures. We further illustrate the results with a specific case: the prediction of co-authors for author "Irini Fundulaki". As can be seen in Fig. 16, orange edges connecting to green nodes represent successfully predicted co-authorships, while grey nodes indicate co-authors that were not predicted. As shown, effective resistance predicts the most co-authors compared to other similarity measures. This highlights its potential for link prediction, as also discussed in [8, 16, 44, 70].

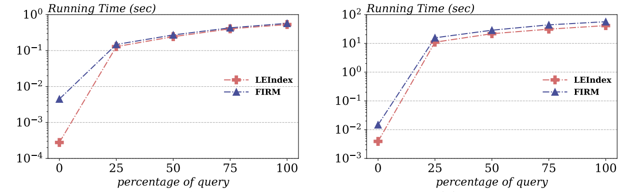
Exp-9: Results of personalized PageRank index maintenance. As we have discussed in Section 4, besides effective resistance index maintenance, our LEIndex technique also has the potential to improve the existing personalized PageRank index maintenance algorithms [28]. In this experiment, we report several initial experimental results. Since the loop-erased walk-based index was experimentally proved to be better than the random walk-based index in terms of index building time, query speed and query accuracy in [36], we focus on studying the index maintenance efficiency for personalized PageRank. We compare our algorithm LEIndex with the SOTA method FIRM, whose original implementation is open-source [28]. We set $\alpha = 0.1$ by default and compare the running time of LEIndex and FIRM under different workloads. We show the results on Hep-th and DBLP. The results on other datasets are consistent. As shown in Fig. 17, our LEIndex technique can achieve a significant speedup compared to FIRM [28]. Different from effective resistance, the running time of personalized PageRank maintenance increases when the update rate increases. This is because personalized PageRank index update is much faster than query processing. For example, on DBLP, LEIndex is an order of magnitude faster than FIRM when performing only updates. This validates our claim that the loop-erased walk-based index can be utilized to improve the personalized PageRank index maintenance.



(a) Effective resistance (b) Personalized PageRank

(c) Jaccard similarity (d) SimRank

Figure 16: Case studies on DBLP



(a) Hep-th (b) DBLP

Figure 17: Results of personalized PageRank index maintenance by our LEIndex technique

6 RELATED WORK

Effective resistance computation. Effective resistance computation is a fundamental problem in graph analysis. In the theoretical community, numerous algorithms have been proposed for computing effective resistance on both static and dynamic graphs [13, 19, 24, 25, 27, 51, 57]. Most of these theoretical solutions primarily aim to improve the worst-case complexity of the problem, but they often perform worse than recently-proposed practical algorithms [35, 37, 45, 66]. For instance, [45] proposes several local algorithms for answering single-pair effective resistance queries, which can compute effective resistance by exploiting only a part of the graph. [66] further enhance the efficiency of effective resistance computation by reducing the estimator's variance. Liao et al. [35] introduce a landmark-based local algorithm for efficiently processing single-pair and single-source effective resistance queries, and they recently generalize this approach using multiple landmark techniques to develop an index-based solution [37]. According to [37], this index-based approach represents the state-of-the-art method for computing effective resistance. However, their method cannot handle dynamic graphs. We are the first to propose an index maintenance method for processing effective resistance queries on evolving graphs.

Dynamic algorithms for PPR computation. Although index maintenance for effective resistance computation has not been extensively studied, there is a rich body of research on similar problems, such as personalized PageRank (PPR) computation [7, 40, 52, 67, 68]. Personalized PageRank is a well-explored topic in graph data management [3, 4, 30, 36, 41, 42, 55, 62]. There are several dynamic algorithms proposed for computing the PPR in the literature. For example, [28] introduces an index-based method for personalized PageRank computation on evolving graphs, including efficient algorithms for maintaining a set of α-random walks. Other techniques, like dynamic push [7, 67], focus primarily on maintaining a single-source PPR vector. Notably, existing solutions are limited to single-source PPR computation and do not address our specific challenge of efficiently answering single-pair effective resistance queries on dynamic graphs. To the best of our knowledge, our work is the first practical index maintenance method for computing effective resistance on evolving graphs.

7 CONCLUSION

In this work, we investigate the index maintenance problem for effective resistance computation on evolving graphs. Unlike existing matrices-based index, we propose two new index structure, namely RWIndex and LEIndex, which maintain the samples of random walks and loop-erased random walks respectively. To efficiently maintain RWIndex and LEIndex, we propose a novel approach that transforms edge updates using a newly-developed landmark node update technique. For maintaining LEIndex, we present a novel and powerful cycle decomposition technique that allows us to maintain the index at the cycle level rather than the node level, significantly enhancing efficiency. We also prove that both of our maintenance algorithms for RWIndex and LEIndex achieve $\widetilde{O}(1)$ time complexity per edge update. Extensive experiments on real-world graphs demonstrate the high efficiency of our solutions.

REFERENCES

[1] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. 2016. On Fully Dynamic Graph Sparsifiers. In *FOCS*. 335–344.

[2] William Aiello, Fan Chung, and Linyuan Lu. 2001. A random graph model for power law graphs. *Experimental mathematics* 10, 1 (2001), 53–66.

[3] Reid Andersen, Christian Borgs, Jennifer T. Chayes, John E. Hopcroft, Vahab S. Mirrokni, and Shang-Hua Teng. 2007. Local Computation of PageRank Contributions. In *WAW*. 150–165.

[4] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.

[5] Eugenio Angriman, Maria Predari, Alexander van der Grinten, and Henning Meyerhenke. 2020. Approximation of the Diagonal of a Laplacian's Pseudoinverse for Complex Network Analysis. In *ESA*, Vol. 173. 6:1–6:24.

[6] Luca Avena and Alexandre Gaudillière. 2018. Two applications of random spanning forests. *Journal of Theoretical Probability* 31, 4 (2018), 1975–2004.

[7] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast Incremental and Personalized PageRank. *Proc. VLDB Endow.* 4, 3 (2010), 173–184.

[8] Bryn Balls-Barker and Benjamin Webb. 2020. Link prediction in networks using effective transitions. *Linear Algebra Appl.* 599 (2020), 79–104.

[9] Ravindra B Bapat. 2010. *Graphs and matrices*. Vol. 27.

[10] Esteban Bautista and Matthieu Latapy. 2022. A local updating algorithm for personalized PageRank via Chebyshev polynomials. *Soc. Netw. Anal. Min.* 12, 1 (2022), 31.

[11] Béla Bollobás and Béla Bollobás. 1998. *Random graphs*.

[12] Enrico Bozzo and Massimo Franceschet. 2013. Resistance distance, closeness, and betweenness. *Soc. Networks* 35, 3 (2013), 460–469.

[13] Dongrun Cai, Xue Chen, and Pan Peng. 2023. Effective Resistances in Non-Expander Graphs. In *ESA*, Vol. 274. 29:1–29:18.

[14] Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. 2024. Almost-Linear Time Algorithms for Incremental Graphs: Cycle Detection, SCCs, s-t Shortest Path, and Minimum-Cost Flow. In *STOC*. 1165–1173.

[15] Fan Chung and Ji Zeng. 2023. Forest formulas of discrete Green's functions. *Journal of Graph Theory* 102, 3 (2023), 556–577.

[16] Manuel Curado. 2020. Return random walks for link prediction. *Information Sciences* 510 (2020), 99–107.

[17] Persi Diaconis and William Fulton. 1991. A growth model, a game, an algebra, Lagrange inversion, and characteristic classes. *Rend. Sem. Mat. Univ. Pol. Torino* 49, 1 (1991), 95–119.

[18] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. 2019. Fully dynamic spectral vertex sparsifiers and applications. In *STOC*. 914–925.

[19] Rajat Vadiraj Dwaraknath, Ishani Karmarkar, and Aaron Sidford. 2023. Towards Optimal Effective Resistance Estimation. In *NIPS*.

[20] Katherine Fitch and Naomi Ehrich Leonard. 2013. Information centrality and optimal leader selection in noisy networks. In *CDC*. 7510–7515.

[21] Yu Gao, Yang P. Liu, and Richard Peng. 2021. Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao. In *FOCS*. 516–527.

[22] David F. Gleich. 2015. PageRank Beyond the Web. *SIAM Rev.* 57, 3 (2015), 321–363.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[24] Gramoz Goranci, Monika Henzinger, and Pan Peng. 2018. Dynamic Effective Resistances and Approximate Schur Complement on Separable Graphs. In *ESA*, Vol. 112. 40:1–40:15.

[25] Craig Gotsman and Kai Hormann. 2023. Efficient Point-to-Point Resistance Distance Queries in Large Graphs. *J. Graph Algorithms Appl.* 27, 1 (2023), 35–44.

[26] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. 2016. Efficient Algorithms for Spanning Tree Centrality. In *IJCAI*. 3733–3739.

[27] Monika Henzinger, Billy Jin, Richard Peng, and David P. Williamson. 2023. A Combinatorial Cut-Toggling Algorithm for Solving Laplacian Linear Systems. *Algorithmica* 85, 12 (2023), 3680–3716.

[28] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proc. ACM Manag. Data* 1, 1 (2023), 25:1–25:26.

[29] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD*. 538–543.

[30] Jinhong Jung, Namyong Park, Lee Sael, and U Kang. 2017. BePI: Fast and Memory-Efficient Method for Billion-Scale Random Walk with Restart. In *SIGMOD*. 789–804.

[31] Rasmus Kyng and Sushant Sachdeva. 2016. Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple. In *FOCS*. 573–582.

[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[33] David A Levin and Yuval Peres. 2017. *Markov chains and mixing times*. Vol. 107.

[34] Huan Li, Richard Peng, Liren Shan, Yuhao Yi, and Zhongzhi Zhang. 2019. Current Flow Group Closeness Centrality for Complex Networks. In *WWW*. 961–971.

[35] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, Hongchao Qin, and Guoren Wang. 2023. Efficient Resistance Distance Computation: The Power of Landmark-based Approaches. *Proc. ACM Manag. Data* 1, 1 (2023), 68:1–68:27.

[36] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, and Guoren Wang. 2022. Efficient Personalized PageRank Computation: A Spanning Forests Sampling Based Approach. In *SIGMOD*. 2048–2061.

[37] Meihao Liao, Junjie Zhou, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, and Guoren Wang. 2024. Efficient and Provable Effective Resistance Computation on Large Graphs: An Index-based Approach. *Proc. ACM Manag. Data* 2, 3 (2024), 133.

[38] Seung-Hwan Lim, Sangkeun Lee, Gautam Ganesh, Tyler C. Brown, and Sreenivas R. Sukumar. 2015. Graph Processing Platforms at Scale: Practices and Experiences. In *ISPAS*. 42–51.

[39] Yang Liu, Chuan Zhou, Shirui Pan, Jia Wu, Zhao Li, Hongyang Chen, and Peng Zhang. 2023. CurvDrop: A Ricci Curvature Based Approach to Prevent Graph Neural Networks from Over-Smoothing and Over-Squashing. In *WWW*. 221–230.

[40] Peter Lofgren. 2014. On the complexity of the Monte Carlo method for incremental PageRank. *Inf. Process. Lett.* 114, 3 (2014), 104–106.

[41] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized PageRank Estimation and Search: A Bidirectional Approach. In *WSDM*. 163–172.

[42] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing Personalized PageRank Quickly by Exploiting Graph Structures. *VLDB* 7, 12 (2014), 1023–1034.

[43] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[44] Benjamin Pachev and Benjamin Webb. 2018. Fast link prediction for large networks using spectral embedding. *Journal of Complex Networks* 6, 1 (2018), 79–94.

[45] Pan Peng, Daniel Lopatta, Yuichi Yoshida, and Gramoz Goranci. 2021. Local Algorithms for Estimating Effective Resistance. In *KDD*. 1329–1338.

[46] Yusuf Yigit Pilavci, Pierre-Olivier Amblard, Simon Barthelmé, and Nicolas Tremblay. 2020. Smoothing Graph Signals via Random Spanning Forests. In *ICASSP*. 5630–5634.

[47] Yusuf Yigit Pilavci, Pierre-Olivier Amblard, Simon Barthelmé, and Nicolas Tremblay. 2021. Graph Tikhonov Regularization and Interpolation Via Random Spanning Forests. *IEEE Trans. Signal Inf. Process. over Networks* (2021), 359–374.

[48] Jim Pitman and Wenpin Tang. 2018. Tree formulas, mean first passage times and Kemeny's constant of a Markov chain. *Bernoulli* 24, 3 (2018), 1942 – 1972.

[49] Maria Predari, Lukas Berner, Robert Kooij, and Henning Meyerhenke. 2023. Greedy optimization of resistance-based graph robustness with global and local edge insertions. *Soc. Netw. Anal. Min.* 13, 1 (2023), 130.

[50] Gyan Ranjan, Zhi-Li Zhang, and Daniel Boley. 2014. Incremental Computation of Pseudo-Inverse of Laplacian. In *Combinatorial Optimization and Applications (Lecture Notes in Computer Science, Vol. 8881)*. 729–749.

[51] Sushant Sachdeva and Yibin Zhao. 2023. A Simple and Efficient Parallel Laplacian Solver. In *SPAA*. 315–325.

[52] Scott Sallinen, Juntong Luo, and Matei Ripeanu. 2023. Real-Time PageRank on Dynamic Graphs. In *HPDC*. 239–251.

[53] Aaron Schild. 2018. An almost-linear time algorithm for uniform random spanning tree generation. In *STOC*. 214–227.

[54] Jieming Shi, Nikos Mamoulis, Dingming Wu, and David W. Cheung. 2014. Density-based place clustering in geo-social networks. In *SIGMOD*. 99–110.

[55] Kijung Shin, Jinhong Jung, Lee Sael, and U Kang. 2015. BEAR: Block Elimination Approach for Random Walk with Restart on Large Graphs. In *SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). 1571–1585.

[56] Ali Kemal Sinop, Lisa Fawcett, Sreenivas Gollapudi, and Kostas Kollias. 2021. Robust Routing Using Electrical Flows. In *SIGSPATIAL*. 282–292.

[57] Daniel A. Spielman and Nikhil Srivastava. 2008. Graph sparsification by effective resistances. In *STOC*. 563–568.

[58] Prasad Tetali. 1991. Random walks and the effective resistance of networks. *Journal of Theoretical Probability* 4, 1 (1991), 101–109.

[59] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. 2022. Understanding over-squashing and bottlenecks on graphs via curvature. In *ICLR*.

[60] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*. 7.

[61] Ulrike von Luxburg, Agnes Radl, and Matthias Hein. 2014. Hitting and commute times in large random neighborhood graphs. *J. Mach. Learn. Res.* 15, 1 (2014), 1751–1798.

[62] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *KDD*. 505–514.

[63] Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH*. 217–218.

[64] David Bruce Wilson. 1996. Generating Random Spanning Trees More Quickly than the Cover Time. In *STOC*. 296–303.

[65] Renchi Yang, Jieming Shi, Yin Yang, Keke Huang, Shiqi Zhang, and Xiaokui Xiao. 2021. Effective and Scalable Clustering on Massive Attributed Graphs. In *WWW*. 3675–3687.

[66] Renchi Yang and Jing Tang. 2023. Efficient Estimation of Pairwise Effective Resistance. *Proc. ACM Manag. Data* 1, 1 (2023), 16:1–16:27.

[67] Minji Yoon, Woojeong Jin, and U Kang. 2018. Fast and Accurate Random Walk with Restart on Dynamic Graphs with Guarantees. In *WWW*. 409–418.

[68] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *KDD*. 1315–1324.

[69] Yanping Zheng, Hanzhi Wang, Zhewei Wei, Jiajun Liu, and Sibo Wang. 2022. Instant Graph Neural Networks for Dynamic Graphs. In *KDD*. 2605–2615.

[70] Kai Zhou, Tomasz P. Michalak, Marcin Waniek, Talal Rahwan, and Yevgeniy Vorobeychik. 2019. Attacking Similarity-Based Link Prediction in Social Networks. In *AAMAS*. 305–313.

[71] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. 2009. Graph Clustering Based on Structural/Attribute Similarities. *VLDB* 2, 1 (2009), 718–729.