Scalable and Provable Kemeny Constant Computation on Static and Dynamic Graphs: A 2-Forest Sampling Approach

Cheng Li, Meihao Liao, Rong-Hua Li, Guoren Wang
Beijing Institute of Technology, China
lichengbit@bit.edu.cn,mhliao@bit.edu.cn,lironghuabit@126.com,wanggrbit@gmail.com

ABSTRACT

Kemeny constant, defined as the expected hitting time of random walks from a source node to a randomly chosen target node, is a fundamental metric in graph data management with numerous real-world applications. However, exactly computing the Kemeny constant on large graphs is highly challenging, as it requires inverting large graph matrices. Existing solutions primarily focus on approximate methods, such as random walk sampling, which still require large sample sizes and lack strong theoretical guarantees. To overcome these limitations, in this paper, we propose a novel approach for approximating the Kemeny constant via 2-forest sampling. We first present an unbiased estimator of the Kemeny constant in terms of spanning trees, by introducing a *path mapping* technique that establishes a direct correspondence between spanning trees and specific sets of 2-forests. Unlike random walk-based estimators, 2-forest-based estimators yield bounded random variables, resulting in a bounded required sample size. Next, we design efficient algorithms to sample and traverse spanning trees, leveraging advanced data structures such as the Binary Indexed Tree (BIT) for efficiency optimization. Our theoretical analysis shows that our method computes the Kemeny constant with relative error ϵ in $O\left(\frac{\Delta^2\bar{d}^2}{\epsilon^2}(\tau + n\min(\log n, \Delta))\right)$ time, where τ is the time to sample a spanning tree, $\bar{d}$ is the average degree, and Δ is the diameter of the graph. This complexity is near-linear in practical scenarios. Furthermore, most existing methods are designed for static graphs and lack mechanisms for dynamic updates. To address this, we propose two sample maintenance strategies that efficiently update samples partially, while preserving estimation accuracy in dynamic graphs. Extensive experiments on 10 large real-world datasets show that our method outperforms state-of-the-art (SOTA) approaches in both efficiency and accuracy, for both static and dynamic graphs.

PVLDB Reference Format:
Cheng Li, Meihao Liao, Rong-Hua Li, Guoren Wang . Scalable and Provable Kemeny Constant Computation on Static and Dynamic Graphs: A 2-Forest Sampling Approach. PVLDB, 14(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Kemeny constant (KC), introduced by Kemeny and Snell [22], is a fundamental metric closely related to hitting time, which is a key quantity associated with random walks. Formally, KC is defined as the expected steps of a random walk that starts from a fixed initial node and stops until reaching a randomly chosen target node according to the stationary distribution. A notable property of KC is that it remains invariant regardless the choice of the starting node [22]. Intuitively, a smaller KC value suggests better overall connectivity of the graph, as nodes are easily reachable from one another. As a unique global network invariant, the Kemeny constant has been applied extensively in various domains of network analysis, including serving as an indicator of network robustness [37], analyzing the disease transmission [55], and quantifying global performance costs in road networks [13]. In recent years, due to its intrinsic connection to hitting times, KC has also been widely used in graph data management, such as network centrality representation [3, 46], graph clustering [10, 34], and recommendation systems [24, 56], among others.

It is well known that the computation of KC can be reformulated as an eigenvalue problem involving the Laplacian or transition matrix of a graph [14, 20, 22, 24]. However, classical methods for computing eigenvalues require a time complexity of $O(n^3)$, making them impractical for large-scale graphs. To overcome this limitation, various sampling-based approaches have been developed to estimate KC with improved scalability at the cost of some accuracy. Random walk-based methods, such as DynamicMC [25] and RefinedMC [50], approximate KC by simulating truncated random walks. Although they avoid the unacceptable computational cost of eigenvalues, these methods still require a large number of samples to achieve acceptable accuracy. More recent methods, including LEwalk [29] and ForestMC [50], leverage the connection between KC and loop-erased random walks (LERWs) to improve sampling efficiency. While these methods may offer some improvements over standard random walks, they still lack strong theoretical guarantees, because the number of steps of a LERWs can be unbounded in worst case, making it difficult to bound sampling size for all kind of graphs. Consequently, LERW-based methods may fail to provide reliable estimates with great efficiency on real-life large graphs.

Recent studies increasingly focus on exploring the connection between spanning trees or spanning forests with the Laplacian matrix, to address classic random walk-related problems, such as the computation of PageRank [27, 30] and effective resistance [28, 31]. Since the scale of a tree or forest can be bounded by the graph diameter, which is more stable on various graphs compared to the length of random walks. This advantage offers the potential for tighter theoretical guarantees about the sample size. Motivated by this insight, we propose a novel formula of KC expressed by the volume of 2-forests. Specifically, a 2-forest is a spanning forest

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

consisting of exactly two disjoint trees (see Fig. 1), and the volume of a 2-forest is related to the sum of the degrees, in the underlying graph, of all nodes in one of its two trees (see Theorem 3.1).

Building on the proposed 2-forest formula of KC, we develop a theoretical framework and an efficient algorithm to estimate KC via sampling 2-forests. A key technique of our approach is *path mapping* that maps a spanning tree to a set of 2-forests, enabling us to construct an unbiased estimator for KC in terms of spanning trees. As a result, we can estimate KC by sampling spanning trees and then deriving 2-forests from them, effectively bypassing the challenges associated with directly sampling 2-forests. To calculate the volume of 2-forests obtained from each spanning tree, we design a depth-first search (DFS) algorithm and optimize it using Binary Indexed Trees (BIT) [15]. Compared to the naive spanning tree traversal method used in similar tasks [28, 29], our approach reduces the time complexity from $O(n\Delta)$ to $O(n \cdot \min(\Delta, \log n))$, where Δ is the diameter. This significantly improves performance on large-scale graphs. Finally, we provide theoretical guarantees for both the correctness and computational complexity of the proposed algorithm. The results show that our method computes the Kemeny constant with relative error ϵ in $O\left(\frac{\Delta^2 \bar{d}^2}{\epsilon^2}(\tau + n\min(\log n, \Delta))\right)$ time, where τ is the time to sample a spanning tree, $\bar{d}$ is the average degree, and Δ is the diameter of the graph. This complexity is near-linear in practical scenarios.

Moreover, most existing methods for estimating KC are designed for static graphs, while real-world networks are often dynamically changing. To address this gap, we develop two sample maintenance strategies that support efficient updates without requiring full re-computation on dynamic graphs. Benefit from the introduction of spanning trees, we can selectively adjust an amount of spanning tree samples according to the change of the entire sample space caused by updates, thereby indirectly preserving the correctness of the induced 2-forests. Specifically, when an edge is inserted, the basic maintenance method replaces a portion of samples by trees that include the new edge. When an edge is deleted, all spanning trees containing the deleted edge are substituted with newly sampled spanning trees from the updated graph. To further enhance efficiency, we introduce an improved sample maintenance method incorporating *link-cut* and *cut-link* operations. These operations enable the transformation of spanning trees that lack a specific edge into ones that contain it, significantly reducing update overhead. Although the resulting spanning trees may not follow a uniform distribution, we derive a correction mechanism that computes the deviation from uniformity and adjusts the weight of the samples accordingly. This ensures accurate KC estimation while achieving substantial computational efficiency on dynamic graphs. Both two methods are faster than resampling from scratch. The correctness and time complexity of two methods are discussed.

We conduct extensive experiments to evaluate the performance of the proposed methods. The results show that, for static graphs, our new algorithm outperforms SOTA methods, with significant improvements on some graphs. For evolving graphs, the two proposed maintenance strategies achieve significant speed-ups compared to static algorithm recalculations. Specifically, the basic maintenance algorithm provides up to an improvement of more than one order of magnitude, while the improved maintenance algorithm achieves a larger speedup, but with a slight sacrifice in accuracy. In summary, the main contributions of this paper are as follows.

Novel approximate algorithms. We propose a novel forest formula of Kemeny constant and design an unbiased estimator by introducing a technique called *path mapping*, which establishes a direct connection between spanning trees and 2-forests. Based on this, we propose a sampling-based algorithm for approximating Kemeny constant and optimize the data structure with Binary Index Trees, which enables nearly linear time complexity and makes it easy to extend to evolving graphs. A detailed theoretical analysis of the proposed algorithm is also provided.

New sample maintenance methods. We propose two novel methods, BasicSM and ImprovedSM, to maintain the correctness of spanning tree samples as the graph evolves, substantially reducing computational cost compared to rerunning static algorithms after each update. BasicSM adjusts the spanning tree samples by pre-computing changes in the sample space, while ImprovedSM further enhances efficiency by reusing prior computations. We provide theoretical analysis showing that, even in the worst case, our algorithms are faster than re-sampling methods, with only a slight sacrifice in accuracy.

Extensive experiments. We conduct comprehensive experiments on 10 real-world graphs to evaluate our algorithms. On static graphs, our method achieves up to an order of magnitude speedup over state-of-the-art algorithms while maintaining the same level of estimation accuracy. For example, on the road network roadNet-PA, our algorithm reaches a relative error of 0.03 in just 26 seconds, whereas two SOTA methods ForestMC and SpanTree require 297 seconds and 676 seconds, respectively, to achieve similar accuracy. On evolving graphs, we further demonstrate the efficiency of our sample maintenance strategies. For the large social networks Orkut, which contains 3 million nodes and 11 million edges, ImprovedSM completes each deletion and insertion update in an average of 1.6 seconds and 4.3 seconds, respectively. In contrast, the fast re-sampling algorithm requires 109 seconds, demonstrating an improvement of approximately one order of magnitude speedup. For reproducibility, the source code for our implementation is available at [https://github.com/ChengLi1010/DynKemeny.git.

2 PRELIMINARIES

2.1 Notations and Concepts

Let $G = (V, E)$ be a simple, connected, undirected graph with $n = |V|$ nodes and $m = |E|$ edges. The adjacency matrix A of G is a $n \times n$ matrix whose (i, j) entry is 1 if and only if edge $(i, j) \in E$ and 0, otherwise. The degree matrix D of G is a diagonal matrix where each entry $D_{ii} = d_i = \sum_{j=1}^{n} A_{ij}$ represents the degree of node i.

A random walk on graph G is a stochastic process. At each step, the walker moves to a randomly chosen neighbor of the current node. The transition probability is described by the matrix $P = (p_{ij}) = D^{-1}A$, where p_{ij} is the probability of moving from node i to node j. It is well known that the stationary distribution π of a random walk on undirected graphs is proportional to node degrees, i.e., $\pi_i = d_i/2m$.

The hitting time $H(i, j)$ is a fundamental measure in the study of random walks [12, 33], defined as the expected number of steps required to visit the node j for the first time, starting from a node

i. For an arbitrarily fixed node i, the expected hitting time from i to any other node j, where j is chosen according to the stationary distribution, is a constant regardless of the choice of i. This constant is known as Kemeny constant [22], denoted as $\kappa(G)$, and is given by $\kappa(G) = \sum_j \pi_j H(i,j)$.

The Kemeny constant can also be expressed in terms of the pseudo-inverse of the normalized Laplacian matrix. The Laplacian matrix L and the normalized Laplacian matrix $\mathcal{L}$ is defined as $L = D - A$ and $\mathcal{L} = D^{-1/2}LD^{-1/2} = I - D^{-1/2}AD^{-1/2}$, respectively. Let $0 = \sigma_1 \leq \cdots \leq \sigma_n$ be the eigenvalues of $\mathcal{L}$, with the corresponding eigenvectors $u_1, \cdots, u_n$. The pseudo-inverse of $\mathcal{L}$ is defined as $\mathcal{L}^\dagger = \sum_{i=2}^{n} \frac{1}{\sigma_i} u_i u_i^\top$. According to [33], the Kemeny constant can be represented as the trace of the pseudo-inverse of the normalized Laplacian matrix, i.e.,

$$\kappa(G) = \mathrm{Tr}(\mathcal{L}^\dagger) = \sum_{i=2}^{n} \frac{1}{\sigma_i}. \tag{1}$$

From Eq. (1), we see that the Kemeny constant can be computed exactly by finding the eigenvectors of the normalized Laplacian matrix, an $O(n^3)$ operation. While theoretically sound, this approach becomes prohibitively expensive for large-scale graphs. Moreover, real-world networks often evolve dynamically, making the challenge of efficiently tracking the Kemeny constant over time even more acute. To address these limitations, we aim to develop scalable algorithms to approximate the Kemeny constant on both static and evolving graphs. Below, we provide a concise overview of the SOTA methods for approximating KC on large graphs.

2.2 Existing SOTA Methods and Their Defects

In this section, we provide an overview of algorithms for computing KC, which can be broadly categorized into three classes: matrix-related methods, truncated random walk-based methods, and loop-erased random walk-based methods. For each category, we briefly discuss their underlying principles and limitations.

Matrix related method. As shown in Eq. (1), KC can be expressed by eigenvalues of the normalized Laplacian matrix $\mathcal{L}$. A naive solution is solving eigenvalues in $O(n^3)$ time, which is impractical for large-scale graphs. Xu et al. [51] proposed ApproxyKemeny based on Hutchinson's Monte Carlo [21]. It approximates the trace of $\mathcal{L}^\dagger$, and thus KC as: $\kappa(G) = \mathrm{Tr}(\mathcal{L}^\dagger) \approx \frac{1}{M}\sum_{i=1}^{M} x_i^\top \mathcal{L}^\dagger x_i$, where x_i are Rademacher random vectors, with each entry independently taking a value of 1 or -1 with equal probability. To compute the quadratic forms of $\mathcal{L}^\dagger$, ApproxyKemeny transforms the problem into solving Laplacian linear systems: $x_i^\top \mathcal{L}^\dagger x_i = \|BL^\dagger y_i\|^2$, where $y_i = D^{1/2}(I - \frac{1}{2m}D^{1/2}\mathbf{1}\mathbf{1}^\top D^{1/2})$. By using a Laplacian Solver to compute $L^\dagger y_i$, KC can be approximated. However, the efficiency and accuracy of ApproxyKemeny are limited by the performance of the specific Laplacian solver used.

Truncated Random Walk Based Method. DynamicMC [25] and RefinedMC [50] utilize truncated random walks to approximate KC. This kind of approach takes advantage of the relationship between KC and the transition matrix P, that is,

$$\kappa(G) = n - 1 + \sum_{k=1}^{\infty}[\mathrm{Tr}(P^k) - 1]. \tag{2}$$

The i-th diagonal entry of P^k represents the transition probability from i to itself after k steps, which becomes negligible when k is large. Therefore, by choosing a sufficiently large truncation length k and estimating the diagonal entry of P^k using truncated random walks initiated from each node, these methods can achieve a small approximation error. To improve performance, DynamicMC employs GPU acceleration to parallelize walk simulations, while RefinedMC improves sampling efficiency by optimizing sample size, truncation length, and the number of starting nodes. However, because of the lack of strong theoretical guarantees, the improvement effect of these optimizations is limited. As a result, these methods has been proven less efficient compared to LERW-based methods as shown in [50].

Loop-Erased Random Walk (LERW) Based Method. Recently, the relationship between $\mathcal{L}^{-1}$ with $(I - P_v)^{-1}$ has been independently explored by [50] and [29] from the perspective of resistance distance and the inverse of the Laplacian submatrix L_v^{-1}, respectively. Specifically, KC can be expressed as:

$$\kappa(G) = \mathrm{Tr}(\mathcal{L}^\dagger) = \mathrm{Tr}(I - P_v)^{-1} + \frac{(\mathcal{L}^\dagger)_{vv}}{\pi_v}. \tag{3}$$

Both methods, ForestMC proposed in [50] and LEwalk proposed in [29], employ loop-erased random walks (LERWs) to approximate $\mathrm{Tr}(I - P_v)^{-1}$, but differ in their computation of $(\mathcal{L}^\dagger)_{vv}/\pi_v$: ForestMC [50] uses truncated random walks, while LEwalk [29] utilizes v-absorbed random walks. The LERW technique itself is well-established, most notably forming the basis of Wilson's algorithm for uniform random spanning trees sampling [47].

The Wilson algorithm constructs a spanning tree through an iterative process: it begins with a single-node tree and sequentially adds LERWs trajectory starting from remaining nodes in arbitrary order. Each LERW terminates upon hitting the current tree, and its acyclic path is incorporated into the tree. Crucially, the expected total length of these LERWs is equal to $\mathrm{Tr}(I - P_v)^{-1}$, allowing an efficient approximation of the first term in Eq. (3). By leveraging LERWs, we can obtain diagonal-related information of the matrix inverse in nearly linear time, significantly improving efficiency compared to performing n independent random walk from each node. As a result, this technique has been widely adopted in recent studies on single-source problems or tasks involving matrix diagonals [5, 28, 30], offering a practical alternative to traditional random walk based methods.

However, analyzing the variance of LERW-based methods remains challenging, primarily due to the unbounded length of loop-erased random walks. In the worst case, the number of steps can grow arbitrarily large, causing such methods to fail on certain graphs. Although ForestMC provides a theoretical guarantee by bounding the absolute error with high probability as the time complexity of algorithm is $O(\epsilon^{-2}\Delta^2 d_{max}^{2\Delta} \log^3 n \cdot \mathrm{Tr}(I - P_v)^{-1})$, the term $d_{max}^{2\Delta}$ makes it impractical for most real-world graphs. There is still no good theoretical guarantee for this type of method. Moreover, all aforementioned methods are designed for static graphs and cannot handle dynamic updates efficiently. Even minor modifications to the graph require complete recomputation, significantly limiting their applicability in evolving graph scenarios.

3 KC ESTIMATION ALGORITHM

In this section, we propose a novel sampling-based algorithm for approximating the Kemeny constant. We begin by a new formulation of KC based on 2-forests. Next, we introduce a technique that can maps spanning trees to 2-forests, and we leverage this to design an unbiased estimator for KC. Based on this framework, we develop a sampling algorithm that estimates KC via sampling uniform random spanning trees (UST), and further optimize it using a Binary Index Tree and depth-first search. The correctness and complexity analysis for proposed algorithm are also discussed.

3.1 New Forest Formula of KC

First, we derive a new expression for KC using 2-forests. Given two fixed nodes u and v, we consider the set of 2-forests in which u and v belong to different trees, denoted as $\mathbb{F}_{u|v}$. Utilizing $n-1$ such sets $\mathbb{F}_{u|v}$, we present a new formula for computing KC.

Theorem 3.1 (Forest formula of KC).

$$\kappa(G) = \frac{1}{2m|\Gamma|} \sum_{u \neq r} d(u) \sum_{\substack{T_1 \cup T_2 \in \mathbb{F}_{r|u} \\ r \in T_1}} \mathrm{vol}(T_1), \tag{4}$$

where r is an arbitrary fixed node, and $\mathbb{F}_{r|u}$ denotes the set of 2-forests where r and u belong to different trees.

Proof.

$$\begin{aligned}
\kappa(G) &= \frac{1}{2m|\Gamma|} \sum_{T_1 \cup T_2 \in \mathbb{F}} \mathrm{vol}(T_1)\mathrm{vol}(T_2) \\
&= \frac{1}{2m|\Gamma|} \sum_{\substack{T_1 \cup T_2 \in \mathbb{F} \\ r \in T_1}} \mathrm{vol}(T_1) \sum_{u \in T_2} d(u) \\
&= \frac{1}{2m|\Gamma|} \sum_{u \neq r} \sum_{\substack{T_1 \cup T_2 \in \mathbb{F} \\ r \in T_1, u \in T_2}} \mathrm{vol}(T_1) d(u) \\
&= \frac{1}{2m|\Gamma|} \sum_{u \neq r} d(u) \sum_{\substack{T_1 \cup T_2 \in \mathbb{F}_{r|u} \\ r \in T_1}} \mathrm{vol}(T_1).
\end{aligned}$$

The first equality follows from Corollary 1.7 in [11], which expresses the Kemeny constant in terms of all 2-forests, where $\mathbb{F}$ denotes the set of all 2-forests in G. Note that for any given 2-forest $T_1 \cup T_2$, the designation of which tree is T_1 or T_2 does not affect anything. Therefore, we deduce the second equation by simply treating the tree containing r as T_1, and utilizing the definition of $\mathrm{vol}(T)$. □

Theorem 3.1 expresses KC in terms of $n-1$ specific subsets of $\mathbb{F}$, denoted as $\mathbb{F}_{r|u}$. To design an unbiased estimator based on this formulation, it is crucial to establish a relationship between spanning trees and $\mathbb{F}_{r|u}$, as the normalization factor in Eq. (4) involves Γ. To bridge this gap, we introduce a technique called *path mapping*, which constructs a special correspondence between spanning trees and 2-forests by fixing a path.

For the 2-forests in $\mathbb{F}_{r|u}$, an interesting discovery is that for any spanning tree, by removing any edge between u and r, we can obtain a 2-forest where u and r belong to different trees. This operation establish a direct connection between spanning trees and $\mathbb{F}_{r|u}$, offering a potential solution to the second challenge. We provide a formal definition below:

Definition 3.2 (Path Mapping). Let $\mathcal{P}$ be a simple path from node u to node r in the graph. We define *path mapping* that associates each spanning tree $\tau \in \Gamma$ with a subset of $\mathbb{F}_{r|u}$, i.e., $\mathcal{P} : \Gamma \mapsto 2^{\mathbb{F}_{r|u}}$, where 2^S denotes the power set of S. For any spanning tree τ, there exists a unique path between u and r, denoted by $\mathcal{P}^{(\tau)}_{u \to r}$. By removing all edges shared by both $\mathcal{P}$ and $\mathcal{P}^{(\tau)}_{u \to r}$, we obtain a set of 2-forests. Depending on the relative direction of the overlapping edges, we define two types of path mappings:

- *Forward Path Mapping*: if the removed edge has same direction in two path, the obtained 2-forest set is denoted as $\mathbb{F}^{P}(\tau)$, i.e. ,
$$\mathbb{F}^{\mathcal{P}}(\tau) = \left\{ \tau \setminus (e_1, e_2) \;\middle|\; (e_1, e_2) \in \mathcal{P} \wedge (e_1, e_2) \in \mathcal{P}^{(\tau)}_{u \to r} \right\},$$
- *Reverse Path Mapping*: if the removed edge has opposite direction in two path, the obtained 2-forest set is denoted as $\mathbb{F}^{P'}(\tau)$, i.e. ,
$$\mathbb{F}^{\mathcal{P}'}(\tau) = \left\{ \tau \setminus (e_1, e_2) \;\middle|\; (e_2, e_1) \in \mathcal{P} \wedge (e_1, e_2) \in \mathcal{P}^{(\tau)}_{u \to r} \right\}.$$

Path mapping facilitates the transformation of any spanning tree into several (or possibly zero) 2-forests within $\mathbb{F}_{r|u}$. However, our goal is to establish a one-to-one correspondence between the partitions of $\mathbb{F}_{r|u}$ and the spanning trees in Γ. Notably, a 2-forest can be derived from multiple spanning trees via path mapping. To resolve this redundancy, we previously introduce the concept of reverse path mapping, $\mathbb{F}^{\mathcal{P}'}(\tau)$, which helps eliminate duplicate 2-forests obtained from different spanning trees.

The following theorem demonstrates that by combining forward and reverse path mapping, we can decompose $\mathbb{F}_{r|u}$ such that each spanning tree in Γ contributes a specific subset of 2-forests (possibly involving additions and subtractions), and the union of all such contributions exactly reconstructs $\mathbb{F}_{r|u}$.

Theorem 3.3. *Let $\mathcal{P}$ be a simple path from u to r, then*

$$\mathbb{F}_{r|u} = \sum_{\tau \in \Gamma} \left(\mathbb{F}^{\mathcal{P}}(\tau) - \mathbb{F}^{\mathcal{P}'}(\tau) \right).$$

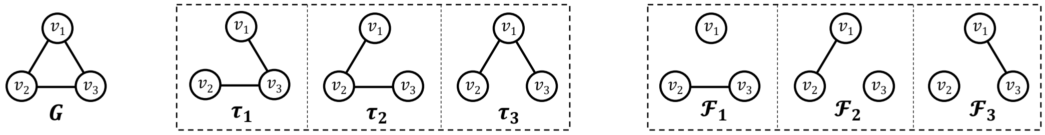
Example 1. *Fig. 2 illustrates how we map Γ to $\mathbb{F}_{u|r}$. The spanning tree set Γ and the 2-forest set $\mathbb{F}$ are shown in Fig. 1(b) and Fig. 1(c), respectively. In $\mathbb{F}$, it is evident that $\mathcal{F}_1$ and $\mathcal{F}_3$ belong to $\mathbb{F}_{v_1|v_3}$, as v_1 and v_3 are located in different trees within these 2-forests. Selecting v_1 as the root node and fixing an arbitrary path from v_2 (or v_3) to v_1, as depicted in Fig. 2(a), we can map Γ to $\mathbb{F}_{v_1|v_2}$ (or $\mathbb{F}_{v_1|v_3}$), as illustrated in Fig. 2(b).*

Specifically, in τ_1, there exist two edges along the path from v_2 to v_1 that align with the fixed path $\mathcal{P}_{v_2}$. By removing these two edges separately, we can map τ_1 to $\mathcal{F}_1$ and $\mathcal{F}_3$. For τ_2 and τ_3, no overlapping edges appear between $\mathcal{P}_{v_2}$ and the paths from v_2 to v_1 in τ_2 or τ_3, resulting in an empty mapping set. By combining these three result sets, we exactly recover $\mathbb{F}_{v_1|v_2}$. This implies that if we sample a spanning tree from Γ uniformly and map it to one or more (possibly zero) 2-forests in $\mathbb{F}_{v|r}$, then each 2-forest is obtained with equal probability.

Lemma 3.4. *Given an arbitrary simple path P from u to r, the following equation holds:*

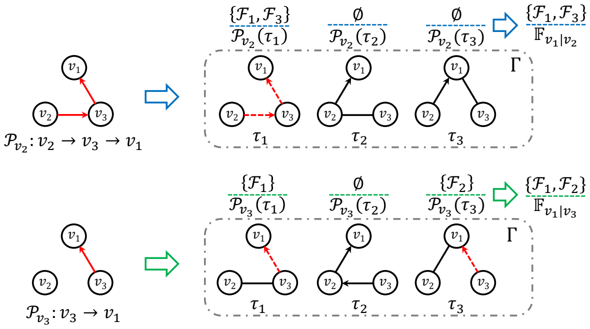
$$\bigcup_{\tau \in \Gamma} \mathbb{F}^{P}(\tau) = \mathbb{F}_{r|u}.$$

Proof. The proof idea is to prove that for any 2-forest, there must be a spanning tree that can be mapped to it by a feasible path.



(a) Graph G (b) $\Gamma = \{\tau_1, \tau_2, \tau_3\}$: set of all spanning trees in G (c) $\mathbb{F} = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\}$: set of all 2-forests in G

Figure 1: An example graph, its spanning trees and 2-forests


$\{\mathcal{F}_1, \mathcal{F}_3\}$
$\mathcal{P}_{v_2}(\tau_1)$
$\emptyset$
$\mathcal{P}_{v_2}(\tau_2)$
$\emptyset$
$\mathcal{P}_{v_2}(\tau_3)$
$\{\mathcal{F}_1, \mathcal{F}_3\}$
$\mathbb{F}_{v_1|v_2}$
Γ
v_1
v_2
v_3
τ_1
τ_2
τ_3
$\mathcal{P}_{v_2}: v_2 \to v_3 \to v_1$
$\{\mathcal{F}_1\}$
$\mathcal{P}_{v_3}(\tau_1)$
$\emptyset$
$\mathcal{P}_{v_3}(\tau_2)$
$\{\mathcal{F}_2\}$
$\mathcal{P}_{v_3}(\tau_3)$
$\{\mathcal{F}_1, \mathcal{F}_2\}$
$\mathbb{F}_{v_1|v_3}$
$\mathcal{P}_{v_3}: v_3 \to v_1$

(a) $\mathcal{P}_{v_i}$: path from v_i to v_1 (b) Mapping Γ to $\mathbb{F}_{v_1|v_2}(\mathbb{F}_{v_1|v_3})$ by using $\mathcal{P}_{v_2}$ ($\mathcal{P}_{v_3}$)

Figure 2: Illustration of Transforming Spanning Trees to 2-Forests Using Path Mapping

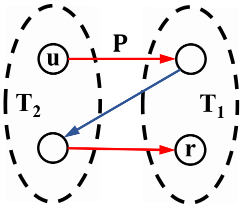


Figure 3: Proof of Lemma 3.5

For each 2-forest $T_1 \cup T_2 \in \mathbb{F}_{r|u}$, the node set V is divided into two distinct sets, T_1 and T_2, and we assume that the tree contains r is T_1. Since P starts from u and ends at r, there must be at least one edge e in P which leaves from the node in T_2 and enters to the node in T_1. Otherwise, u and r are in the same one connected component, or path P fails to connect u and r. Adding the edges accrossing two components to the 2-forest can get a spanning tree $T_1 \cup T_2 \cup e$, which means this 2-forest $T_1 \cup T_2$ is an element of $\mathbb{F}^P(T_1 \cup T_2 \cup e)$. □

Lemma 3.5. *For any 2-forest $T_1 \cup T_2 \in \mathbb{F}_{r|u}$, we have*

$$\left|\{\tau \mid T_1 \cup T_2 \in \mathbb{F}^P(\tau)\}\right| - \left|\{\tau \mid T_1 \cup T_2 \in \mathbb{F}^{P'}(\tau)\}\right| = 1.$$

Proof. for each 2-forest $T_1 \cup T_2 \in \mathbb{F}_{r|u}$, we assume r is in T_1. As proved in Lemma 3.4, if a spanning tree τ is composed of $T_1 \cup T_2$ and an edge e in P, which leaves from T_2 and enters to T_1 (as the red edges in Fig. 3). Then $T_1 \cup T_2$ appears in the set obtained by path mapping τ, i.e. , $T_1 \cup T_2 \in \mathbb{F}^P(T_1 \cup T_2 \cup (e_1, e_2))$, for each $(e_1, e_2) \in P$, satisfying $e_1 \in T_2, e_2 \in T_1$. Similarly $\mathbb{F}^{P'}(\tau')$ has $T_1 \cup T_2$ if τ' is composed of $T_1 \cup T_2$ and the edge that is pointed to T_2 from T_1 (see the blue edge in Fig. 3). Therefore

$$\begin{aligned}
&\left|\{\tau \mid T_1 \cup T_2 \in \mathbb{F}^P(\tau)\}\right| - \left|\{\tau \mid T_1 \cup T_2 \in \mathbb{F}^{P'}(\tau)\}\right| \\
=&|(e_1, e_2) \in P \mid e_1 \in T_2 \wedge e_2 \in T_1| \\
&- |(e_1, e_2) \in P \mid e_1 \in T_1 \wedge e_2 \in T_2| \\
=&1.
\end{aligned}$$

The last equality holds because P is a simple path from $u \in T_2$ to $r \in T_1$, which makes the edges from T_2 to T_1 exactly one more than the edge from T_1 to T_2 in P. □

Proof of Theorem 3.3. According to Lemma 3.4 and Lemma 3.5, we can know that $\sum_{\tau \in \Gamma} \mathbb{F}^P(\tau)$ includes all elements in $\mathbb{F}_{r|u}$, while $-\sum_{\tau \in \Gamma} \mathbb{F}^{P'}(\tau)$ eliminates the excess. Therefore, the right term of equation precisely matches $\mathbb{F}_{r|u}$. □

Although we slightly abuse the set notation of addition and subtraction to operate on the multiplicities of identical 2-forests, the final construction guarantees that each 2-forest appears exactly once, resulting in an exact reconstruction of $\mathbb{F}_{r|u}$. Combining Theorem 3.3 with Theorem 3.1, KC can be expressed in terms of spanning trees as follows.

Lemma 3.6.

$$\kappa(G) = \frac{1}{2m|\Gamma|} \sum_{\tau \in \Gamma} f(\tau), \tag{5}$$

where

$$f(\tau) = \sum_{u \neq r} d(u) \left(\sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u}(\tau)} \mathrm{vol}(T_1) - \sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}'_u}(\tau)} \mathrm{vol}(T_1) \right),$$

$r \in T_1$ is a fixed root node, $\{\mathcal{P}_u \mid u \neq r\}$ represents a set of fixed simple paths from u to r, and $\mathcal{P}'_u$ denotes the reverse path of $\mathcal{P}_u$.

Proof. By using Theorem 3.1 and Theorem 3.3, we have:

$$\begin{aligned}
\kappa(G) &= \frac{1}{2m|\Gamma|} \sum_{u \neq r} d(u) \sum_{\substack{T_1 \cup T_2 \in \mathbb{F}_{r|u} \\ r \in T_1}} \mathrm{vol}(T_1) \\
&= \frac{1}{2m|\Gamma|} \sum_{u \neq r} d(u) \sum_{\tau \in \Gamma} \left(\sum_{\substack{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}}(\tau) \\ r \in T_1}} \mathrm{vol}(T_1) - \sum_{\substack{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}'}(\tau) \\ r \in T_1}} \mathrm{vol}(T_1) \right) \\
&= \frac{1}{2m|\Gamma|} \sum_{\tau \in \Gamma} \sum_{u \neq r} d(u) \left(\sum_{\substack{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}}(\tau) \\ r \in T_1}} \mathrm{vol}(T_1) - \sum_{\substack{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}'}(\tau) \\ r \in T_1}} \mathrm{vol}(T_1) \right) \\
&= \frac{1}{2m|\Gamma|} \sum_{\tau \in \Gamma} f(\tau).
\end{aligned}$$

□

3.2 The Tree-To-Forest Algorithm

In this section, we present a sampling-based algorithm TTF for approximating KC, leveraging Eq. (5). By applying Lemma 3.6, we can construct an unbiased estimator of KC transforming uniformly sampled spanning trees into corresponding 2-forests.

Specifically, we sample ω spanning trees uniformly and fix a root node r along with a predefined set of paths $\mathcal{P}_u$ from each node u to r. Let $\hat{\tau}_i$ denote the i-th sampled spanning tree. Then, KC can be estimated as: $\tilde{\kappa} = \frac{1}{2mT}\sum_{i=1}^{\omega} f(\hat{\tau}_i)$.

Lemma 3.7. *$\tilde{\kappa}$ is an unbiased estimator of $\kappa(G)$.*

To sample a uniform random spanning tree (UST), we employ the current SOTA method, Wilson's algorithm [47], which utilizes loop-erased random walks and completes the sampling in $O(\mathrm{Tr}(I - P_v)^{-1})$ time. The algorithm starts with a tree containing only the root. Following an arbitrarily fixed order, the algorithm performs a loop-erased random walk starting from each node. Once the walk reaches a node already in the tree, the acyclic path obtained by erasing loops from the walk is then added to the tree. This process continues until all nodes are included, yielding a UST [2, 43, 47].

Given a UST τ, we maintain an auxiliary array vol_sum to store the sum of $\mathrm{vol}(T_1)$ for each node u for all 2-forests $T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u}(\tau) - \mathbb{F}^{\mathcal{P}'_u}(\tau)$. This enables efficient computation of $f(\tau)$ as $f(\tau) = \sum_{u \neq r} d(u) \cdot \texttt{vol\_sum}[u]$. However, if we enumerate each node u and map the tree to $\mathbb{F}^{\mathcal{P}}(\tau)$, we must traverse the path from each node to the root r in the tree, resulting in a worst-case time complexity of $O(n^2)$. By constraining the length of fixed reference paths to the graph diameter Δ, the complexity can be reduced to $O(n\Delta)$. Still, when the diameter is large, this approach becomes computationally expensive. To overcome this, we optimize the data structure to achieve a better complexity of $O(n \min(\log n, \Delta))$.

To accelerate path comparisons, we leverage the fact that a spanning tree uniquely defines paths from all nodes to the root r. By fixing a reference tree τ_0, we obtain a set of paths. Based on this, we further utilize structural properties of trees: for any edge e along the path from a node u to the root, the node u lies in the subtree rooted at e. Therefore, rather than comparing the entire paths directly, we reverse the perspective: we check whether nodes belong to the same edge-induced subtrees in both trees.

Lemma 3.8. *Given an arbitrary spanning tree τ and a fixed spanning tree τ_0, let r be the root of both trees. Let $\mathcal{P}_u^{(\tau_0)}$ denote the path from node u to r in τ_0. Then, the following equality holds:*

$$\sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u^{(\tau_0)}}(\tau)} \mathrm{vol}(T_1) = \sum_{\substack{(e_1,e_2)\in\tau \\ u \in S}} (2m - \mathrm{vol}(Sub(\tau, e_1))), \quad (6)$$

where $Sub(\tau, e_1)$ is the set of nodes in the subtree rooted at edge e_1 within τ, and $S = Sub(\tau, e_1) \cap Sub(\tau_0, e_1)$. Each edge (e_1, e_2) is directed towards the root r, meaning that if the edge is removed, node e_2 remains on the same side as the root r.

We utilize depth-first search (DFS) to avoid the costly process node-by-node comparison when identifying common nodes between two subtrees. A key property of DFS is that, when visiting a node u, only the nodes on the path from u to the root remain active, meaning that DFS has not yet backtracked from them. So we update the vol_sum values for all nodes in the subtree of the active path in the fixed tree, and query the corresponding value when DFS visits that node. Before DFS backtracks from a node, we remove its contribution to ensure correctness in future queries.

Algorithm 1: TTF

```
Input: A graph $G$, the sample size $\omega$, the root node $r$
Output: $\tilde{\kappa}$ as the estimation of Kemeny Constant
$\tilde{\kappa} \leftarrow 0$;
Generate $\tau_0$ using BFS with root $r$;
$\mathrm{DFS_{in}}, \mathrm{DFS_{out}} \leftarrow \tau_0$;
for $i \leftarrow 1$ to $\omega$ do
    $\tau_i \leftarrow$ wilson $(G, r)$;
    for each node $u \in V$ do
        vol[$u$] $\leftarrow$ the volume of $Sub(\tau_i, u)$;
    $\tilde{\kappa} \leftarrow \tilde{\kappa}$ + DFS $(u, \tau_i, \tau_0)$;
return $\tilde{\kappa}/(2m \cdot \omega)$;
Function DFS $(u, \tau, \tau_0)$:
    Let $f_u$ be the father node of $u$ in $\tau$;
    if $(u, f_u) \in \tau_0$ then
        vol_sum.$add$(DFS$_{in}$[$u$], DFS$_{out}$[$u$], $2m-$vol[$u$]) ;
    else if $(f_u, u) \in \tau_0$ then
        vol_sum.$add$(DFS$_{in}$[$f_u$], DFS$_{out}$[$f_u$], vol[$f_u$]$-2m$) ;
    $\tilde{\kappa} \leftarrow d(u) \cdot$ vol_sum.$query(u)$;
    for each node $i \in Child_\tau(u)$ do
        $\tilde{\kappa} \leftarrow \tilde{\kappa}+$ DFS $(i, \tau, \tau_0)$;
    if $(u, f_u) \in \tau_0$ then
        vol_sum.$add$(DFS$_{in}$[$u$], DFS$_{out}$[$u$], vol[$u$]$-2m$) ;
    else if $(f_u, u) \in \tau_0$ then
        vol_sum.$add$(DFS$_{in}$[$f_u$], DFS$_{out}$[$f_u$], $2m-$vol[$f_u$]) ;
    return $\tilde{\kappa}$;
```

To efficiently update the vol_sum values for all nodes within a subtree, we use DFS numbering (DFN) combined with the Binary Indexed Tree (BIT, or Fenwick Tree [15]). DFN records each node's entry and exit times during the DFS traversal, denoted as $\mathrm{DFS_{in}}$ and $\mathrm{DFS_{out}}$, respectively. Based on these, nodes are reordered by their entry times. A key property is that nodes within any subtree appear as a contiguous segment in this ordering, that is starting from the entry time of the subtree's root and ending to its exit time.

When range updates are required, using a regular array provides constant-time queries but can incur linear-time updates. Conversely, differential arrays allow constant-time updates but slow queries. Since each DFS traversal performs two updates and one query, we adopt a BIT to balance both, achieving $O(\log n)$ time per operation. Specifically, we support:

- $Add(l, r, v)$: Add value v to all elements in the range $[l, r]$.
- $Query(i)$: Retrieve the value of the i-th element.

Example 2. *Fig. 4 illustrates how* DFS *and* BIT *efficiently maintain* vol_sum *and compute $f(\tau)$ for sampled USTs. The first column shows the fixed tree τ_0 with its* DFN *$(4-2-3-1)$, the second column displays the sampled tree τ along with its subtree volumes.*

Beginning the DFS from v_4, we examine its outgoing edges (v_4, v_1) and find it also appears in τ_0. Using the BIT*, we efficiently update*

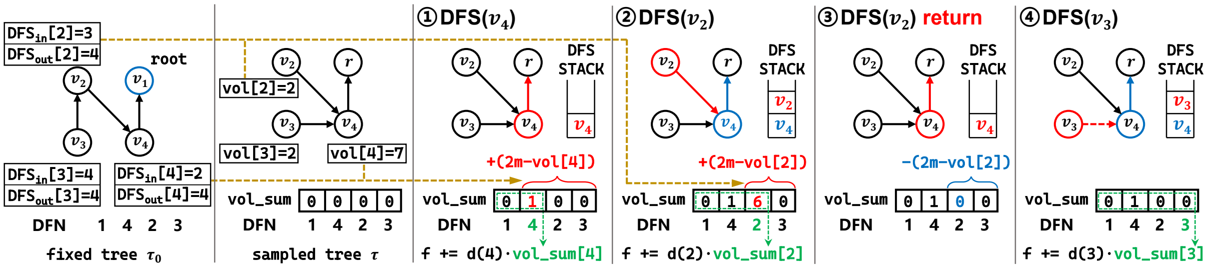


Figure 4: Illustration of the DFS process

vol_sum by adding $2m-$vol[4] to all nodes in subtree of v_4 in τ_0, using $O(\log n)$ time. After updating vol_sum*, we query* vol_sum[v_4]*, which corresponds to the second element in* DFN*. Since* vol_sum *is maintained as a differential array, the actual value is computed as a prefix sum using* BIT*, also in $O(\log n)$ time.*

The same procedure applies to v_2 and v_3. For v_2, since both edges (v_2, v_4) and (v_4, v_1) appear in both trees and lie on the path from v_2 to the root. Thus, vol_sum[2] *is influenced by both the update for (v_2, v_4) and the earlier update for (v_4, v_1). In contrast, the path from v_3 to the root in τ does not include (v_2, v_4). Therefore, before DFS backtracks from v_2, we must remove its contribution (third step), to ensure the correctness of query* vol_sum[3] *in the fourth step.*

Finally, the algorithm for estimating KC is as presented in Algorithm 1. Given a graph G, a root node r, and a sample size ω, the algorithm begins by constructing a fixed spanning tree τ_0 using breadth-first search (BFS), followed by computing its DFN (Lines 2–3). It then samples ω USTs using Wilson algorithm [47] (Line 5). For each sampled UST τ_i, the algorithm first computes the volume for all subtrees (Lines 6–7), and then invokes the DFS function to accumulate contributions to the KC estimator (Line 8). Finally, the algorithm returns $\tilde{\kappa}/(2m \cdot \omega)$ as the approximation of KC (Line 9).

The core procedure DFS operates as follows. Let f_u denote the parent node of the current node u in τ. The algorithm checks whether the edge (u, f_u) exists in τ_0. If so, it increments vol_sum for all nodes in the subtree rooted at u by $2m-$vol$[u]$ (Lines 12–13). Conversely, if the reverse edge (f_u, u) exists in τ_0, it decrements vol_sum for all nodes in $Sub(\tau_0, f_u)$ by $2m-$ vol$[f_u]$ (Lines 14–15). After these updates, the algorithm queries vol_sum$[u]$ and incorporates its value into the KC estimation (Line 16). Then, it recursively invokes DFS on each child node of u (Lines 17–18). Before returning from DFS, related changes made to vol_sum must be revert to maintain correctness for subsequent computations (lines 19-22).

Theorem 3.9 (Time Complexity of Algorithm 1). *The time complexity of Algorithm 1 is*

$$O\left(\omega \cdot \left(\mathrm{Tr}(I-P_r)^{-1} + n\min(\Delta, \log n)\right)\right).$$

Proof. The time complexity of Wilson Algorithm for generating a UST is $O\left(\mathrm{Tr}\left((I-P_r)^{-1}\right)\right)$, where P_r is the transition probability matrix with its r-th column and row removed. Calculating the volume of each subtree requires $O(n)$ time. For the DFS function, traversing all nodes takes $O(n)$ time. For each node, updating and querying the vol_sum incurs an additional $O(\log n)$ cost, due to the properties of the Binary Indexed Tree. Consequently, the time complexity of each iteration is $O(n \log n)$.

However, if the graph has a small diameter Δ such that $\Delta < \log n$, the computation of vol_sum can be limited to within Δ steps from each node to the root, effectively reducing the complexity to $O(n\Delta)$. Therefore, we use $O(n \cdot \min(\Delta, \log n))$ to capture both scenarios. Multiplying by the number of samples T, the overall time complexity of the algorithm is as stated in the theorem. □

Theorem 3.10 (Error Bound of Algorithm 1). *If the sample size satisfies $\omega \ge \frac{8m^2\Delta_G^2\log(2/p_f)}{n^2\varepsilon^2}$, then Algorithm 1 outputs an estimate $\tilde{\kappa}$ such that $|\tilde{\kappa} - \kappa| \le \varepsilon\kappa$ with probability at least $1 - p_f$.*

Proof. For each sampled tree τ, the number of 2-forests mapped by τ_0 for a node u is bounded by the distance from u to the root r. If we construct τ_0 such that its depth is minimized, this distance is at most Δ, the diameter of the graph. Additionally, for each forest, the volume vol(T_1) is bounded by $2m$. Consequently, $f(\tau)$ can be bounded as $f(\tau) \le \sum_u d(u) \cdot 2m\Delta \le 4m^2\Delta$.

Applying Hoeffding's inequality [18], we derive the following bound on the probability of deviation:

$$\begin{aligned}\Pr\left(\left|\frac{1}{2mT}\sum_{i=1}^{T} f(\tau_i) - \kappa\right| \ge \varepsilon\kappa\right) &\le \Pr\left(\left|\frac{1}{2mT}\sum_{i=1}^{T} f(\tau_i) - \kappa\right| \ge \varepsilon \cdot n\right) \\ &\le 2\exp\left(-\frac{2\varepsilon^2 n^2 T}{(4m\Delta)^2}\right) \\ &\le 2\exp\left(-\frac{2\varepsilon^2 n^2 \cdot 8m^2\Delta^2 \log(\frac{2}{p_f})}{16m^2\Delta^2 \cdot \varepsilon^2 n^2}\right) \\ &\le p_f.\end{aligned}$$

This completes the proof. □

Combined these two theorems, we can obtain that Algorithm 1 can compute KC in $O\left(\frac{\Delta^2\tilde{d}^2}{\epsilon^2}(\phi + n\min(\log n, \Delta))\right)$ to achieve a relative error ϵ, where $\phi = \mathrm{Tr}(I-P_r)^{-1}$. As observed in our experiments, this time complexity is near-linear in practice. Algorithm 1 details an efficient strategy for processing each sampled UST in $O(n\log n)$ time. Additionally, as discussed earlier, a simpler implementation with $O(n\Delta)$ complexity can be employed to handle low-diameter graphs. This alternative matches the computational cost of SpanTree [29], which also requires $O(n\Delta)$ time for additional computation on each UST. which we include in our experimental comparison. To evaluate the practical impact of this improvement, we compare our approach against SpanTree in the experimental study. As demonstrated in Section 5, our method consistently outperforms SpanTree across all tested graphs, validating the effectiveness of the $O(n\log n)$ design over the $O(n\Delta)$ baseline.

Comparison to [11]. Corollary 1.7 in [11] also presents a clean and elegant forest formula for KC, but its practical utility is limited by the difficulty of uniformly sampling 2-forests. Unlike spanning trees, a 2-forest requires an exact partition of the node set into two disjoint connected components. A naive strategy that first partitions the nodes and then generates a spanning tree in each part incurs exponential overhead, with $O(2^{|V|})$ possible partitions. Moreover, even with a direct sampling method, constructing an unbiased estimator is nontrivial, as the coefficient involves the total number of spanning trees ($1/|\Gamma|$), rather than the number of 2-forests ($1/|\mathbb{F}|$). To address these challenges, we introduce two novel formulas for KC, Eq. (4) and Eq. (5), which enable efficient and unbiased estimation through sampling.

Comparison to SpanTree **[29].** While our method also leverages USTs, similar to SpanTree proposed in [29], the foundational principles differ significantly. SpanTree estimates KC based on Eq. (3) via an electrical interpretation, aggregating current flow across sampled USTs. In contrast, our approach is built upon the forest formulas of KC (Eq. (4) and Eq. (5)) and introduces a novel path-mapping technique to convert trees into 2-forests, which further allows us to incorporate the optimization of BIT to enhance computational efficiency. Experimental results confirm that our method dominates SpanTree across all tested scenarios.

4 KC COMPUTATION ON DYNAMIC GRAPHS

In this section, we focus on the problem of calculating KC on dynamic graphs. Our algorithms are improved based on the static algorithm presented in Section 3, where we store needed information for those sampled spanning trees and maintain the correctness of samples to efficiently update KC. In Section 4.1, we first present a basic maintenance method, and in Section 4.2, we introduce an improved method that more efficiently maintains samples. Theoretical analysis of both algorithms is proved respectively.

4.1 Basic Samples Maintenance

After a new edge is added or removed from the graph, the spanning trees of the new graph G' are very similar to those of the original graph, with the only difference being the presence (or absence) of all spanning trees that contain the edge e, as we assume that the graphs before and after the update are both connected. Therefore, a natural idea is to adjust the sampled USTs, ensuring that the resulting set still represents a uniform distribution of spanning trees for the updated graph. These new "valid" spanning trees can then be used to estimate the KC.

Insertion Case. We first consider the scenario where a new edge $e = (u, v)$ is inserted into the graph. Let Γ denote the spanning tree set of the original graph G, and let Γ' represent the corresponding set of the updated graph G' after inserting the edge e. The set Γ' comprises all spanning trees in Γ, along with additional spanning trees that include the newly inserted edge e. We define the subset of all these new spanning trees in Γ' that contain e as Γ'_e.

A well-established fact states that the proportion of spanning trees containing a given edge is equal to the effective resistance of that edge, i.e., $\frac{|\Gamma'_e|}{|\Gamma'|} = R(e)$ [17, 33]. Here, $R(e)$ denotes the effective resistance between nodes u and v for edge $e = (u, v)$ in G'. Before the update, we already have some sampled spanning trees that are uniformly distributed in Γ. Our goal is to transform these samples into trees that are uniformly distributed in Γ'. By using the relationship mentioned above, we can ensure that how many spanning trees containing e are expected to have in the samples for G'. Therefore, replacing part of old samples by these "new" USTs help us gain a correct sample set under the uniform distribution for updated graph.

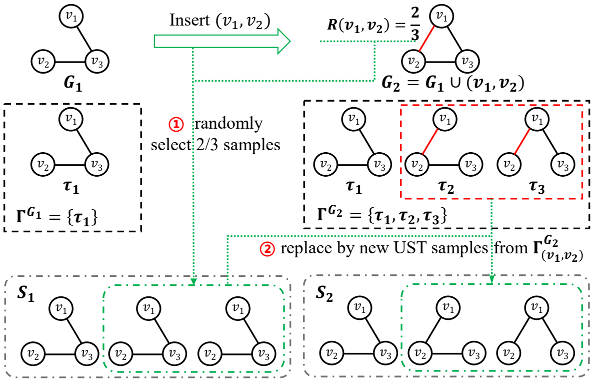

v_1
v_2
v_3
G_1
Insert (v_1, v_2)
$R(v_1, v_2) = \frac{2}{3}$
$G_2 = G_1 \cup (v_1, v_2)$
τ_1
$\Gamma^{G_1} = \{\tau_1\}$
① randomly select 2/3 samples
τ_1
τ_2
τ_3
$\Gamma^{G_2} = \{\tau_1, \tau_2, \tau_3\}$
② replace by new UST samples from $\Gamma^{G_2}_{(v_1,v_2)}$
S_1
S_2

Figure 5: Illustration of Basic Samples Maintenance for Edge Insertion

Example 3. *Fig. 5 illustrates the process of maintaining the UST-samples when an edge is inserted. When the edge (v_1, v_2) is added to G_1, we first compute its effective resistance in the updated graph G_2, which is $R(v_1, v_2) = \frac{2}{3}$. According to the properties of effective resistance, in the new graph, the edge (v_1, v_2) should appear in approximately $\frac{2}{3}$ of the USTs. To update the sample set accordingly, we randomly select $\frac{2}{3}$ of the USTs from the original sample set S_1 and replace them with the same number of new USTs that include (v_1, v_2). This replacement is performed using a modified version of Wilson's algorithm. Finally, we utilize the updated sample set S_2 to estimate the* KC *of G_2.*

Theorem 4.1.

$$\mathbb{E}_{\tau\sim U(\Gamma')}[f(\tau)] = (1 - R(e))\mathbb{E}_{\tau\sim U(\Gamma)}[f(\tau)] + R(e)\mathbb{E}_{\tau\sim U(\Gamma_e)}[f(\tau)], \quad (7)$$

where $U(\Gamma)$ represents the uniform distribution over the set Γ, i.e., τ is sampled with probability $\frac{1}{|\Gamma|}$ for all $\tau \in \Gamma$. The function $f(\tau)$ corresponds to the calculation performed on the given spanning tree τ. In this paper, it refers to the definition in Lemma 3.6.

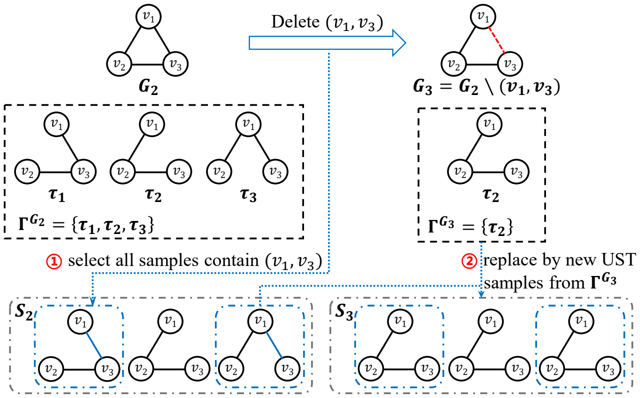

v_1
v_2
v_3
G_2
Delete (v_1, v_3)
$G_3 = G_2 \setminus (v_1, v_3)$
τ_1
τ_2
τ_3
$\Gamma^{G_2} = \{\tau_1, \tau_2, \tau_3\}$
τ_2
$\Gamma^{G_3} = \{\tau_2\}$
① select all samples contain (v_1, v_3)
② replace by new UST samples from Γ^{G_3}
S_2
S_3

Figure 6: Illustration of Basic Sample Maintenance for Edge Deletion

Proof.

$$
\begin{aligned}
\mathop{\mathbb{E}}_{\tau\sim U(\Gamma')}[f(\tau)] =& \frac{1}{|\Gamma'|}\sum_{\tau\in\Gamma'} f(\tau) \\
=& \frac{1}{|\Gamma'|}\left(\sum_{\tau\in\Gamma} f(\tau) + \sum_{\tau\in\Gamma'_e} f(\tau)\right) \\
=& \frac{1}{|\Gamma'|}\sum_{\tau\in\Gamma} f(\tau) + \frac{1}{|\Gamma'|}\sum_{\tau\in\Gamma'_e} f(\tau) \\
=& \frac{(1-R(e))}{|\Gamma|}\sum_{\tau\in\Gamma} f(\tau) + \frac{R(e)}{|\Gamma'_e|}\sum_{\tau\in\Gamma'_e} f(\tau) \\
=& (1-R(e))\mathop{\mathbb{E}}_{\tau\sim U(\Gamma)}[f(\tau)] \\
& + R(e)\mathop{\mathbb{E}}_{\tau\sim U(\Gamma_e)}[f(\tau)].
\end{aligned}
$$

The second line holds because of $\Gamma \cup \Gamma'_e = \Gamma'$ and $\Gamma \cap \Gamma'_e = \emptyset$. And the forth line can be deduced by $|\Gamma'_e|/|\Gamma'| = R(e)$. □

Deletion Case. As for deleting an edge e, it is more easier to find the way to maintain samples. Only those sampled trees which contains e should not exist in new samples. So we just need to remove all these tree and replace them by regular USTs for new graph. To avoid confusion, we still use Γ to represent the case without edge e, and Γ' to represent the spanning trees of the graph that contains the edge e. However, the sample distribution now is expected to be converted from $U(\Gamma')$ to $U(\Gamma)$.

Example 4. *Fig. 6 illustrates the process of updating the UST-samples when an edge is deleted. When the edge (v_1, v_3) is removed from G_2, all spanning trees that contain (v_1, v_3) must be excluded from the updated sample set. To achieve this, we first identify all USTs in the original sample set S_2 that include (v_1, v_3). These trees are then replaced with the same number of newly generated USTs from the updated graph G_3, resulting in the new sample set S_3. Finally, updated samples in S_3 are used to approximate the* KC *of G_3.*

Lemma 4.2.

$$\mathbb{E}_{\tau\sim U(\Gamma)}[f(\tau)] = \mathbb{E}_{\tau\sim U(\Gamma')}[f(\tau) \mid e \notin \tau].$$

Algorithm 2: BasicSM

```
Input: A graph $G = (V, E)$, root $r$, an updated edge (u, v), original
       sample set $S$
Output: $\tilde{\kappa}'$ as an updated estimation, updated sample set $S'$
1  $\tilde{\kappa}' \leftarrow \tilde{\kappa}(G)$, $S' \leftarrow S$;
2  if Update is Ins $(u, v)$ then
3      Calculate $R(u, v)$ in $G \cup (u, v)$;
4      Randomly select $\lceil R(u, v) \cdot |S| \rceil$ trees from $S$;
5      for each selected tree $\tau$ do
6          $\tilde{\kappa}' \leftarrow \tilde{\kappa}' - f(\tau)/|S|$,   $S' \leftarrow S' \setminus \tau$;
7          $\tau' \leftarrow$ wilson $(G', (u, v))$;
8          Redirect edges in $\tau'$ to point towards $r$;
9          $f(\tau') \leftarrow$DFS $(r, \tau', \tau_0)$;
10         $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau')/|S|$,   $S' \leftarrow S' \cup \tau'$;
11 if Update is Del $(u, v)$ then
12     for each tree $\tau \in S$ that contains edge $(u, v)$ do
13         $\tilde{\kappa}' \leftarrow \tilde{\kappa}' - f(\tau)/|S|$,   $S' \leftarrow S' \setminus \tau$;
14         $\tau' \leftarrow$ wilson $(G', r)$;
15         $f(\tau') \leftarrow$DFS $(r, \tau', \tau_0)$;
16         $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau')/|S|$,   $S' \leftarrow S' \cup \tau'$;
17 return $\tilde{\kappa}'$, $S'$;
```

Proof.

$$
\begin{aligned}
\mathbb{E}_{\tau\sim U(\Gamma')}[f(\tau) \mid e \notin \tau] &= \sum_{\tau\in\Gamma'} f(\tau)\Pr[\tau \sim U(\Gamma') \mid e \notin \tau] \\
&= \sum_{\tau\in\Gamma} f(\tau)\frac{\Pr[\tau \sim U(\Gamma') \wedge \tau \in \Gamma]}{\Pr[e \notin \tau]} \\
&= \sum_{\tau\in\Gamma} f(\tau)\frac{1}{|\Gamma'|}\Big/\frac{|\Gamma|}{|\Gamma'|} \\
&= \frac{1}{|\Gamma|}\sum_{\tau\in\Gamma} f(\tau) \\
&= \mathbb{E}_{\tau\sim U(\Gamma)}[f(\tau)]
\end{aligned}
$$

□

The pseudo-code for the basic sample maintenance (BasicSM) algorithm, covering both edge insertion and deletion cases, is presented in Algorithm 2. For each sampled tree, both $f(\tau)$ and its corresponding edges need to be stored. In the case of edge insertion, we first compute the effective resistance $R(u, v)$ in the updated graph to determine the number of indices that need to be replaced (Lines 3–4). For the computation of single-pair effective resistance, we employ the state-of-the-art method Bipush [31]. For the selected trees that are to be discarded, their contribution to the estimator $\tilde{K}$ should be subtracted (Line 6). To sample a uniform spanning tree that includes the newly inserted edge (u, v), A modified Wilson algorithm can be applied by setting u and v as the root [6, 41] and then redirecting all tree edges toward the root node r (Lines 7–8). Finally, we compute $f(\tau')$ for these newly generated trees following the approach in Algorithm 1 and store them as part of the updated samples (lines 9–10). In deletion case, we just check all samples whether contains (u, v). For those trees with (u, v), we replace them with uniformly spanning trees without (u, v) by using the regular Wilson algorithm on the graph G', and recalculate their contribution for $\tilde{K}$ (Lines 12-16).

Theorem 4.3 (Correctness and Time Complexity of Algorithm 2). *Algorithm 2 returns an estimate $\tilde{\kappa}'$ that satisfies the relative-error guarantee. The overall time complexity of algorithm 2 is*

$$O\left(R(e)\omega \cdot \left(\mathrm{Tr}(I-P_e)^{-1} + n\min(\Delta, \log n)\right)\right),$$

where e denotes the inserted (or deleted) edge, and $R(e)$ represents the effective resistance of edge e in the graph that includes e.

Proof of Theorem 4.3. To establish the correctness of Algorithm 2, we leverage Theorem 4.1 and Theorem 4.2 to construct an unbiased estimator for the KC of the updated graph. However, to accelerate the computation, we avoid recalculating $f(\tau)$ for unchanged USTs, even though its value undergoes slight variations due to changes in the degrees of the updated edge's endpoints. Fortunately, the resulting error in the estimator $\tilde{\kappa}'$ can be bounded by Δ, which is negligible when considering the relative error, as κ is larger than n and thus significantly greater than Δ. Experimental results further confirm that the estimation maintains a high level of accuracy.

Regarding time complexity, only $R(e) \cdot T$ USTs are updated. The time complexity for constructing and querying these USTs remains similar to the static case. The only exception arises in the insertion case, where the modified Wilson algorithm incurs a complexity of $O(\mathrm{Tr}(I-P_e)^{-1})$ instead of $O(\mathrm{Tr}(I-P_r)^{-1})$. However, the difference between these two complexity is minimal and remains within the same level. Consequently, the overall time complexity for updating aligns with the statement of the theorem. □

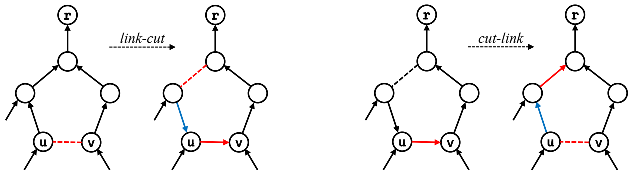
Discussion. Compared to Algorithm 1, Algorithm 2 offers an efficient strategy for maintaining UST samples correctness by selectively updating only a $R(e)$ fraction of samples. This reduces the sample size from ω to $R(e)\omega$. As shown in Table 1, the average effective resistance is relatively small, typically below 0.1 in social networks. Although the time complexity of sampling USTs changes to $\mathrm{Tr}(I-P_e)^{-1}$, the impact on performance is negligible. Experimental results further highlight the substantial improvements in computational efficiency achieved through this method.

4.2 Improved Sample Maintenance

Although the basic maintenance method significantly reduces computation compared to static algorithms, discarding old sample still leads to a certain degree of information loss. Is it possible to preserve the correctness of samples while maximizing the utility of the computational results from these trees? To address this, we discuss the insertion and deletion scenarios separately.

Insertion Case. When considering to transform a tree that does not contain edge e into one that does, what happens if we simply add edge e to the tree? This operation creates a cycle, and by removing any edge other than e from this cycle, a new tree that includes e can be obtained. We refer to this process as a *link-cut* operation. Specifically, *link-cut*(τ, e) is defined as linking the edge e and cutting one of the other edges in the cycle created by e and path in tree.

Using this method to generate spanning trees containing edge e is not only significantly faster than the Wilson algorithm but also preserves parts of the tree's structure, allowing us to recompute only the affected parts. As illustrated in Fig. 7a, after linking the edge (u, v) and cutting a randomly selected edge in the cycle, all nodes whose paths to the root are altered lie within the subtree of u. This enables us to leverage the previous results for unaffected nodes, further reducing the computational cost.



(a) link-cut operation **(b) cut-link operation**

Figure 7: Illustration of link-cut and cut-link operation

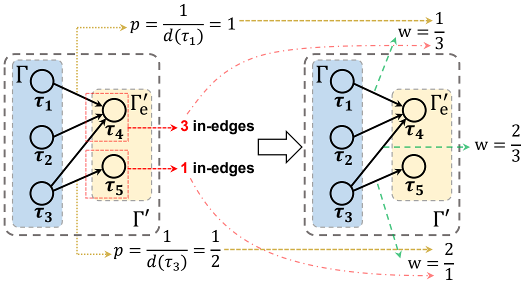


Figure 8: Link-Cut Bipartite Graph $\mathcal{B}$

However, the spanning trees sampled by *link-cut* are not uniformly distributed. To better understand this, consider a representation where each spanning tree is treated as a node, and an edge exists between two nodes if one tree can be transformed into the other through a link-cut operation. This forms a bipartite graph between Γ and Γ'_e, denoted as $\mathcal{B}$. Let $\mathcal{N}_{\mathcal{B}}(\tau)$ represents the set of neighbors of a tree τ, corresponding to the spanning trees that can be obtained by performing a link-cut operation on τ. The degree of each tree $\tau \in \Gamma$, denoted as $d(\tau)$, is determined by the length of the cycle formed when edge e is added to τ.

Clearly, if we randomly transform a tree in Γ into one of its neighbors with equal probability $1/d(\tau)$ using the link-cut operation, some trees in Γ'_e will have higher selection probabilities than others. Fortunately, this bias can be corrected by appropriately weighting the tree, as long as we know the relationship between these non-uniform probabilities and the uniform distribution. As illustrated in Figure 8, the non-uniformity of link-cut sampling arises primarily due to two factors:

- The out-degree of nodes $\tau \in \Gamma$ are different.
- The in-degree of nodes $\tau_e \in \Gamma'_e$ are different.

By appropriately reweighting based on these two factors, we can ensure that trees in Γ are transformed into Γ'_e uniformly.

Example 5. *Figure 8 illustrates a link-cut bipartite graph for an evolving graph. Here τ_1 to τ_5 represent all spanning trees in the updated graph. Among them, τ_1 to τ_3 belong to the original graph tree set Γ, while τ_4 and τ_5 are new trees due to the update. Each edge in the bipartite graph represents a possible transformation between trees via the link-cut operation.*

As previously mentioned, due to the difference of outdegree, the probability of selecting an outgoing edge from τ_1 higher than that *from τ_3. Additionally, τ_4 can be obtained from τ_1, τ_2, and τ_3, whereas τ_5 can only be obtained from τ_3. If we uniformly select a tree from Γ and apply the link-cut operation, the probability of obtaining τ_4 is $p(\tau_4) = \frac{1}{3} \times 1 + \frac{1}{3} \times 1 + \frac{1}{3} \times \frac{1}{2} = \frac{5}{6}$, while the probability of obtaining τ_5 is only $p(\tau_5) = \frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$.*

By incorporating reweighting to correct these biases, as illustrated on the right side of Figure 8, we achieve a uniform distribution over Γ_e'. Specifically, the corrected probabilities become: $p(\tau_4) = \frac{1}{3} \times \frac{1}{3} + \frac{1}{3} \times \frac{1}{3} + \frac{1}{3} \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$ and $p(\tau_5) = \frac{1}{3} \times \frac{1}{2} \times 2 = \frac{1}{3}$.

Theorem 4.4.

$$\mathop{\mathbb{E}}_{\tau \sim U(\Gamma_e')} [f(\tau)] = \frac{1 - R(e)}{R(e)} \mathop{\mathbb{E}}_{\tau \sim U(\Gamma)} \left[\mathop{\mathbb{E}}_{\tau_e \sim U(\mathcal{N}_B(\tau))} \left[f(\tau_e) \frac{d(\tau)}{d(\tau_e)} \right] \right]. \quad (8)$$

Proof.

$$\begin{aligned} & \mathop{\mathbb{E}}_{\tau \sim U(\Gamma)} \left[\mathop{\mathbb{E}}_{\tau_e \sim U(\mathcal{N}_{\mathcal{B}}(\tau))} \left[f(\tau_e) \frac{d(\tau)}{d(\tau_e)} \right] \right] \\ =& \frac{1}{|\Gamma|} \sum_{\tau \in \Gamma} \mathop{\mathbb{E}}_{\tau_e \sim U(\mathcal{N}_{\mathcal{B}}(\tau))} \left[f(\tau_e) \frac{d(\tau)}{d(\tau_e)} \right] \\ =& \frac{1}{|\Gamma|} \sum_{\tau \in \Gamma} \left(\frac{1}{d(\tau_e)} \sum_{\tau_e \in \mathcal{N}_{\mathcal{B}}(\tau)} f(\tau_e) \frac{d(\tau)}{d(\tau_e)} \right) \\ =& \frac{1}{|\Gamma|} \sum_{\tau \in \Gamma} \sum_{\tau_e \in \mathcal{N}_{\mathcal{B}}(\tau)} f(\tau_e) \frac{1}{d(\tau_e)} \\ =& \frac{1}{|\Gamma|} \sum_{\tau_e \in \Gamma_e'} d(\tau_e) (f(\tau_e) \frac{1}{d(\tau_e)}) \\ =& \frac{R(e)}{1 - R(e)} \frac{1}{|\Gamma_e'|} \sum_{\tau_e \in \Gamma_e'} f(\tau_e) \\ =& \frac{R(e)}{1 - R(e)} \mathop{\mathbb{E}}_{\tau \sim U(\Gamma_e')} [f(\tau)] \end{aligned}$$

□

Assigning the weight of trees obtained via the link-cut operation as $d(\tau)/d(\tau_e)$ provides an estimate for $f(\tau)$ under the uniform distribution in Γ_e'. By substituting Eq. (8) into the second term on the right-hand side of Eq. (7), we obtain an unbiased estimator of KC for the updated graph G'.

Deletion Case. Similarly, we can transform a tree containing edge e into one without it through the *cut-link* operation. *Cut-link*(τ_e, e) involves cutting e from a spanning tree, resulting in a 2-forest, and then linking another edge from graph G to reconnect the 2-forest. This operation changes the paths of all nodes within the subtree of the newly linked edge, while other paths remain unchanged. As in the insertion case, we can correct the distribution by assigning weights, with only slight differences in implementation.

Theorem 4.5.

$$\mathop{\mathbb{E}}_{\tau \sim U(\Gamma)} [f(\tau)] = \frac{R(e)}{1 - R(e)} \mathop{\mathbb{E}}_{\tau_e \sim U(\Gamma_e')} \left[\mathop{\mathbb{E}}_{\tau \sim U(\mathcal{N}_B(\tau_e))} \left[f(\tau) \frac{d(\tau_e)}{d(\tau)} \right] \right]. \quad (9)$$

The implementation of improved sample maintenance (ImprovedSM) algorithm is shown in Algorithm 3. The main distinction lies in the fact that the new spanning tree is obtained directly through the link-cut process, which ensures that only a portion of the nodes in the original tree are affected. Updates of $f(\tau')$ are performed exclusively for these affected nodes, and the corresponding weights are computed (Lines 6–8, 19–21). Additionally, after each update, the weights of all trees need to be normalized (Lines 10–13, 23–24). Initially, the weight of each tree is set to $1/|\omega|$, so there is no need to divide by $|\omega|$ again when calculating KC in the final step.

Algorithm 3: ImprovedSM

```
Input: A graph $G = (V, E)$, root $r$, an updated edge $(u, v)$, original
       sample set $S$
Output: updated estimation $\tilde{\kappa}'$, updated sample set $S'$
1  $\tilde{\kappa}' \leftarrow 0$;
2  if Update is Ins $(u, v)$ then
3      Compute $R(u, v)$ in $G \cup (u, v)$;
4      Randomly select $\lceil R(u, v) \cdot |S| \rceil$ trees in $S$;
5      for each selected tree $\tau$ do
6          $\tau' \leftarrow$ link-cut $(\tau, (u, v))$;
7          Update $f(\tau')$;
8          Calculate weight as $w(\tau') = \frac{d(\tau)}{d(\tau')} w(\tau)$;
9      for each tree $\tau \in S'$ do
10         if $\tau$ is updated then
11             $w(\tau) \leftarrow w(\tau)/w_{sum}(\tau_{selected}) \cdot R(u, v)$;
12         else
13             $w(\tau) \leftarrow w(\tau)/w_{sum}(\tau_{unselected}) \cdot (1 - R(u, v))$;
14         $\tilde{\kappa}' \leftarrow \tilde{\kappa'} + f(\tau) \cdot w(\tau)$;
15 if Update is Del $(u, v)$ then
16     $w_{org} \leftarrow 0$ ;
17     for each tree $\tau \in T$ that contains edge $(u, v)$ do
18         $w_{org} \leftarrow w_{org} + w(\tau)$;
19         $\tau' \leftarrow$ cut-link $(\tau, (u, v))$;
20         Update $f(\tau')$;
21         Calculate weight as $w(\tau') = \frac{d(\tau')}{d(\tau)} w(\tau)$;
22     for each tree $\tau \in S'$ do
23         if $\tau$ is updated then
24             $w(\tau) \leftarrow w(\tau)/w_{sum}(T_{updated}) \cdot w_{org}$;
25         $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau) \cdot w(\tau)$;
26 return $\tilde{\kappa}', S'$;
```

Theorem 4.6 (Correctness and Time Complexity of Algorithm 3). *Algorithm 3 returns an estimate $\tilde{\kappa}'$ that satisfies the relative-error guarantee. The overall time complexity algorithm 3 is*

$$O\left(R(e)T \cdot (m + n \min(\Delta, \log n))\right),$$

where e denotes the inserted (or deleted) edge, and $R(e)$ represents the effective resistance of edge e in the graph that includes e.

Proof. By combining Theorem 4.4 and Theorem 4.5, we substitute them into Lemma 4.1 and Lemma 4.2, respectively. Algorithm 3 produces an estimator for KC on the updated graph while maintaining a relative-error guarantee, under the assumption that the required sample size remains at the same level even if the maximum value of $f(\tau)$ changes.

For any $\tau \in \Gamma$, its degree $d(\tau)$ in $\mathcal{B}$ can be computed in $O(n)$ time by counting the length of the unique path between the endpoints of the inserted edge e. Similarly, for $\tau_e \in \Gamma_e'$, its degree $d(\tau_e)$ is determined by the number of trees that can be transformed into τ_e via a link-cut operation, which is equivalent to the number of trees that can be obtained from τ_0 via a cut-link operation. Consequently,

Table 1: Detailed statistics of datasets (Δ denotes the graph diameter, ϕ is the expected time of sampling a UST, and $\bar{R}(e)$ represents the average effective resistance of each edge)

Dataset	n	m	Δ	ϕ	$\bar{R}(e)$
PowerGrid	4,941	6,594	46	4.4×10^4	0.749
Hep-th	8,638	24,806	17	1.7×10^4	0.348
Astro-ph	17,903	196,972	14	2.3×10^4	0.091
Email-enron	33,696	180,811	11	5.0×10^4	0.186
Amazon	334,863	925,872	44	9.0×10^5	0.362
DBLP	317,080	1,049,866	21	5.9×10^5	0.302
Youtube	1,134,890	2,987,624	20	1.8×10^6	0.380
roadNet-PA	1,087,562	1,541,514	786	1.7×10^7	0.706
roadNet-CA	1,957,027	2,760,388	849	3.5×10^7	0.709
Orkut	3,072,441	117,185,083	9	3.1×10^6	0.026

$d(\tau_e)$ equals the number of edges in G that cross between T_1 and T_2, where $T_1 \cup T_2 = \tau_e \setminus e$. This computation has a time complexity of $O(n)$. Therefore, the overall complexity of updating for each tree is $O(n)$.

The time complexity for updating $f(\tau')$ remains $O(n \log n)$, as its worst case requires recomputing the function for the entire tree. Hence, the overall time complexity of Algorithm 3 is $O(R(e)T(m + n \log n))$. In the case of scale-free graphs, where $m = O(n \log n)$, this complexity simplifies to $O(R(e)Tn \log n)$, representing a significant improvement over its static counterpart. □

Discussion. Algorithm 3 further accelerates the UST sampling process compared to Algorithm 2. By leveraging *link-cut* and *cut-link* operations, it can construct the required spanning tree more efficiently than Wilson's algorithm. Although computing the weights introduces an $O(m)$ time complexity, in practice, it is significantly faster than sampling USTs. Furthermore, since the tree structure changes only slightly, previously computed results of $f(\tau)$ can be partially reused, which reduces the overall computational cost. While this approach may increase the variance, the resulting estimates remain within an acceptable range and are comparable to the other methods. These observations are further validated by our experimental results.

5 EXPERIMENTS

5.1 Experimental Settings

Datasets. We employ 10 real-world datasets including various type of graphs, primarily focusing on social networks and road networks. All datasets can be downloaded from [23]. Since Kemeny constant is defined on connected graphs, we focus on the largest connected component for each graph in this study. The detailed statistics of each dataset are summarized in Table 1. To simulate dynamic updates, we randomly select 100 edges from each graph to represent edge insertions and another 100 edges for edge deletions, thereby modeling typical update scenarios in dynamic graph settings. As for ground truth, we approximate KC using ApproxyKemeny with $\epsilon = 0.15$ as the ground truth for small graphs ($n < 10^6$), following previous studies [25, 50]. For larger graphs, ApproxyKemeny fails to provide results within an acceptable time, so we employ our proposed method TTF with a large sample size $T (T = 10^5)$ to obtain a high-precision estimation as the ground truth.

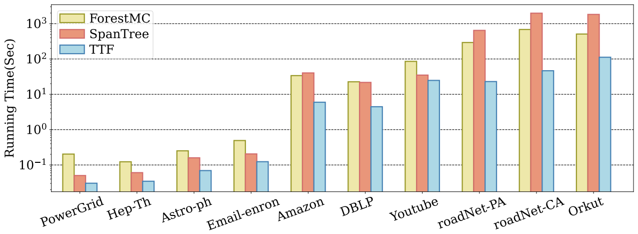


Figure 9: Running time of different algorithms for KC estimation on static graphs

Different algorithms. For static graphs, We employ our method TTF (Algorithm 1), along with SpanTree [29] and ForestMC [50]. SpanTree is the only existing method based on USTs prior to our work, while ForestMC represents the current SOTA method for KC approximation. We exclude DynamicMC and RefinedMC from comparison, as they have been shown to be less efficient than ForestMC in previous work [50]. Regarding the number of samples, both TTF and SpanTree provide theoretical guarantees on the sample size required to achieve a ϵ-relative error. Although ForestMC does not have a feasible sample size with guarantee, we set its sample size equal to that of the other two methods for fair comparison. For dynamic graphs, we use the three static algorithms by re-running them after each updates as baseline methods, and evaluate two proposed sample maintenance methods: BasicSM and ImprovedSM. In all experiments across both static and dynamic scenarios, we set $\epsilon = 0.5$ by default, and choose the node with maximum degree as the root when sampling USTs or simulating LERWs, following prior studies [29, 50].

Experimental environment. All algorithms used in our experiment are implemented in C++ and compiled with g++ 11.2.0 using the -O3 optimization flag, except for ApproxyKemeny [51], which is implemented in Julia. We conduct all experiments on a Linux server with a 64-core 2.9GHz AMD Threadripper 3990X CPU and 128GB memory. Each experiment is repeated five times to avoid accidental anomalies.

5.2 Experiment results

Exp-I: Overall Performance on Static Graphs. In this experiment, we evaluate the performance of different algorithms on static graphs in terms of running time and estimation accuracy, as shown in Fig. 9 and Fig. 10. The results demonstrate that our proposed method TTF consistently outperforms the other two algorithms across all datasets. Compared to SpanTree, TTF achieves substantial speedup by reducing the additional computational complexity from $O(n\Delta)$ to $O(n \log n)$. Compared to ForestMC, the advantage mainly stems from the reduced number of required samples. While ForestMC achieves high accuracy on certain datasets like Youtube and Orkut, its performance varies significantly across different graphs, indicating a strong sensitivity to graph structure. In contrast, TTF maintains robust performance with consistently high accuracy and low running time across all datasets. For instance, on the roadNet-CA dataset, all three algorithms achieve comparable accuracy, but TTF completes in just 47 seconds, whereas ForestMC and SpanTree take 683 seconds and 1983 seconds, respectively.

Exp-II: Performance on Static Graphs with Varying ϵ. In this experiment, we evaluate the performance of static algorithms by

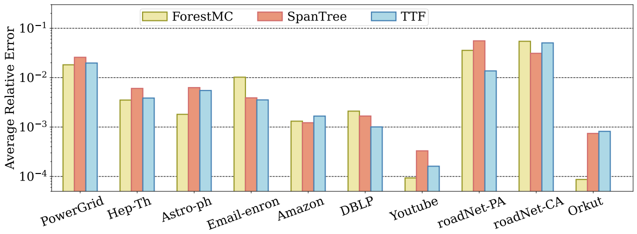
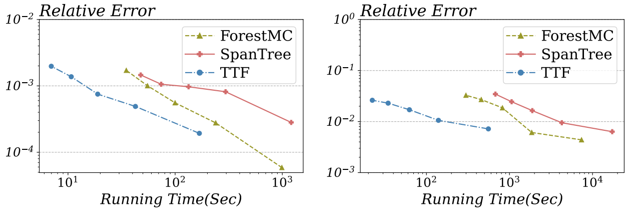


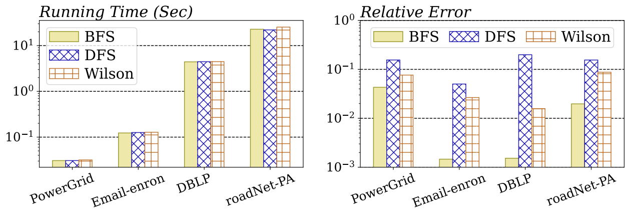
Figure 10: Relative error of different algorithms for KC estimation on static graphs



(a) Amazon

(b) roadNet-PA

Figure 11: Performance of different algorithms for varying ϵ from 0.5 to 0.1



(a) Running Time (Sec)

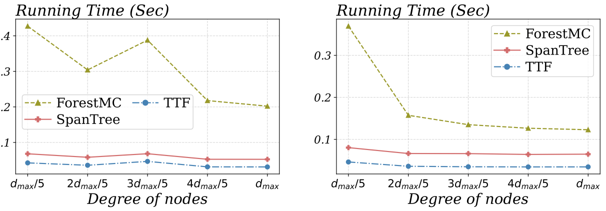
(b) Relative Error

Figure 12: Performance of TTF on 4 datasets with three different path selection methods

varying ϵ from 0.5 to 0.1. We select two representative datasets, Amazon and roadNet-PA, and compare running time against relative error. As shown in Fig. 11, TTF requires less time than the other two methods to achieve the same level of accuracy. For example, in roadNet-PA, TTF achieves a relative error of 0.03 in just 26 seconds, whereas ForestMC requires 297 seconds and SpanTree takes 676 seconds to reach comparable accuracy. Overall, TTF demonstrates significantly better efficiency and accuracy compared to the other two methods.

Exp-III: Effect of Path Selection. In Algorithm 1, we use BFS to construct a spanning tree that defines the set of paths used for path mapping. To investigate the impact of different path selection strategies, we also experiment with DFS and Wilson's algorithm. The results are presented in Fig. 12. We observe that while the choice of path selection method has a negligible impact on running time, it significantly affects estimation accuracy. As discussed in Theorem 3.10, the estimation value is bounded by the maximum length of paths. When the spanning tree is generated via BFS, the maximum path length is bounded by the graph diameter, whereas it can be substantially longer when using DFS or randomly sampled trees. Consequently, shorter paths tend to reduce the variance of TTF, leading to more accurate estimations.

Exp-IV: Effect of Root Selection. All three static methods involve selecting a root node when applying Wilson's algorithm, and this choice significantly affects the running time. Following common practice in prior work [29, 50], we investigate the impact of root selection by dividing all nodes into five groups based on degree, and measuring the average running time when selecting roots from each group. Results on PowerGrid and Hep-th are presented in Fig. 15. We observe that selecting a root with a higher degree generally leads to lower running time. Therefore, we adopt the highest-degree node as the default root in our experiments.



(a) PowerGrid

(b) Hep-th

Figure 13: Impact of root node with different degree on running time for static algorithms

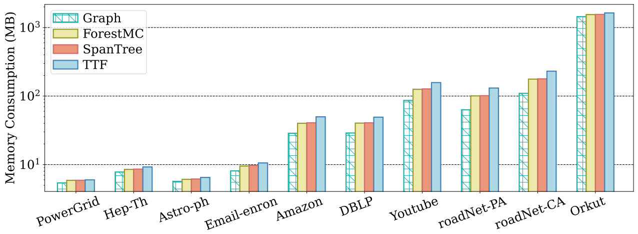
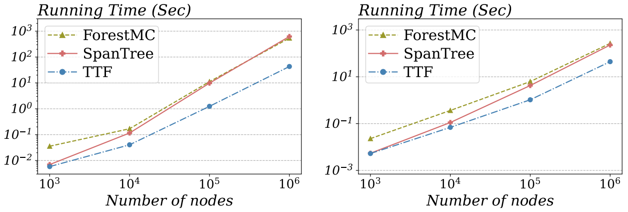


Figure 14: Memory consumption of different algorithms



(a) Erdős–Rényi model

(b) Power-law graph model

Figure 15: Scalability performance of different algorithms on random generated graphs

Exp-V: Memory Consumption. This experiment evaluates the memory consumption during each algorithm running, compared to storing pure graph. The experimental results are shown in Fig 14. In all datasets, TTF requires less than 2× the memory consumption of Graph, which stores only $n-1$ edges for each UST in the index. This demonstrates that TTF maintains $O(n)$ space complexity for each UST. Across all datasets, the maximum memory consumption of TTF is around 1.6 GB, highlighting its space efficiency.

Exp-VI: Scalability. In this experiment, we evaluate the scalability of the three methods on randomly generated graphs. Fig. 15a shows the results on Erdős–Rényi graphs [9] with $p = 5 \times 10^{-4}$, while Fig. 15b presents the results on power-law graphs [1] with exponent $\gamma = 3.5$. The results demonstrate that TTF and ForestMC exhibit better scalability than SpanTree, as the running time of SpanTree increases more rapidly with the number of nodes.

Exp-VII: Performance for Edge Insertion on Dynamic Graphs. We evaluate the performance of the basic and improved sample

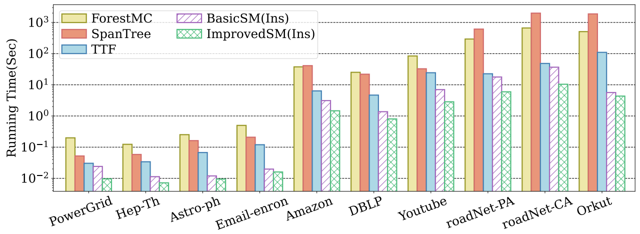


Figure 16: Running time of different algorithms for edges insertions on dynamic graphs

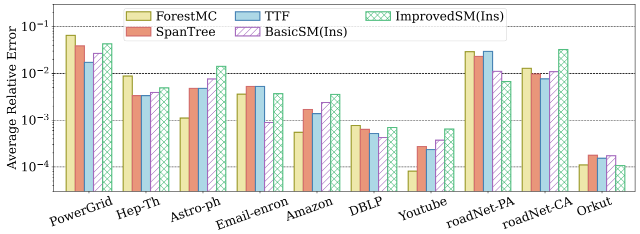


Figure 17: Accuracy performance of different algorithms for edges insertions on dynamic graphs

maintenance methods, BasicSM and ImprovedSM, for edge insertions, comparing their accuracy and running time against three static methods that recompute KC after each update. The results are shown in Fig. 16 and Fig. 17. As expected, both BasicSM and ImprovedSM significantly outperform the static methods in terms of time efficiency. On social networks like Orkut, BasicSM and ImprovedSM are an order of magnitude faster than TTF. However, on road networks such as roadNet-PA and roadNet-CA, the speed-up of BasicSM is limited, as the average effective resistance of edges is close to one, which means nearly all samples are required to be resampled. Additionally, sampling a UST that includes a specific edge might be slower regular Wilson algorithm. In contrast, ImprovedSM still achieves notable speed-ups even on road networks, since link-cut operations are significantly faster than tree sampling, though this comes at the cost of slightly higher error.

Exp-VIII: Performance for Edge Deletion on Dynamic Graphs. We also evaluate the performance of BasicSM and ImprovedSM for edge deletions. The results, shown in Fig. 18 and Fig. 19, are generally consistent with those from the insertion scenario. Notably, BasicSM shows better performance improvements on road networks during deletions than insertions, since it can employ the standard Wilson algorithm for tree sampling. Moreover, both BasicSM and ImprovedSM are slightly faster for deletions than insertions. This is likely because edge insertions require computing effective resistance, which introduces additional computational overhead, whereas deletions do not.

6 RELATED WORK

Random walk computation. Random walks have long served as a fundamental tool in graph analysis, supporting a wide range of applications such as recommendation [38, 48, 56], network embedding [16, 40, 59], and complex network analysis [42, 49]. Various random walk-based metrics, personalized PageRank (PPR) [32, 45, 53] and effective resistance (ER) [8, 39, 44, 54] have received a lot of attention. For both PPR and ER, local algorithms such as push are commonly used in recent studies [4, 28, 45], as they operate efficiently on a small portion of the graph without requiring full access to its structure. Although KC is also inherently related to random walks, it represents a fundamentally global measure, distinct from these problems, which makes existing techniques developed for PPR and ER unsuitable for direct application to KC computation.

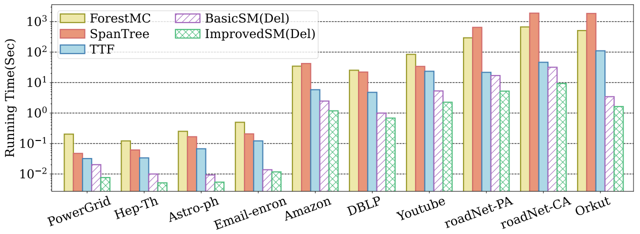


Figure 18: Running time of different algorithms for edges deletions on dynamic graphs

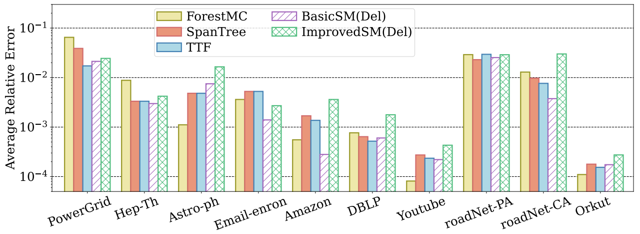


Figure 19: Accuracy performance of different algorithms for edges deletions on dynamic graphs

Algorithms on Dynamic Graphs. Dynamic graphs, also known as evolving graphs or graphs in an incremental setting, naturally arise in real-world applications where graph data is continuously updated. The design of algorithms for various graph problems in dynamic graphs has been an active research area for decades. In particular, dynamic algorithms for random walk-based metrics, such as Personalized PageRank, have been extensively studied [19, 26, 52, 58, 60]. Most dynamic algorithms build upon their corresponding static versions. For example, LazyForward [58] and TrackingPPR [36] were both developed as extensions of the push algorithm [4]. Similarly, various index update methods for random walks [7, 19, 35] and power iteration-based dynamic algorithms [26, 57] have been explored. Recently, Liao et al. investigated the connection between USTs and both Personalized PageRank [27, 30] and effective resistance [28] successively. These methods provide a foundation for extending our index maintenance strategy, as shown in Section 4, to these problems, facilitating the development of efficient dynamic algorithms.

7 CONCLUSION

In this work, we study the problem of approximating Kemeny constant problem for both static and dynamic graphs. We propose two novel formulas of Kemeny constant and design an biased estimator based on spanning trees and 2-forests. For static graphs, we develop a sampling-based algorithm with stronger theoretical guarantees. The proposed method outperforms SOTA approaches in both efficiency and accuracy on real-world datasets. For dynamic graphs, we introduce two sample maintenance strategies that efficiently preserve the correctness of samples instead of recomputing from scratch for each update. Both two methods are faster than static algorithms, with only a slight loss in precision. Extensive experiments on real-world graphs demonstrate the effectiveness and efficiency of our solutions.

REFERENCES

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

[2] David J Aldous. 1990. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discret. Math.* 3, 4 (1990), 450–465.

[3] Diego Altafini, Dario A Bini, Valerio Cutini, Beatrice Meini, and Federico Poloni. 2023. An edge centrality measure based on the Kemeny constant. *SIAM J. Matrix Anal. Appl.* 44, 2 (2023), 648–669.

[4] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *FOCS*. IEEE, 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. *ESA* 173 (2020), 6:1–6:24.

[6] Luca Avena, Fabienne Castell, Alexandre Gaudillière, and Clothilde Mélot. 2018. Random forests and networks analysis. *Journal of Statistical Physics* 173 (2018), 985–1027.

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

[8] Béla Bollobás. 2013. *Modern graph theory*. Vol. 184. Springer Science & Business Media.

[9] Béla Bollobás and Béla Bollobás. 1998. *Random graphs*. Springer.

[10] Mo Chen, Jianzhuang Liu, and Xiaoou Tang. 2008. Clustering via Random Walk Hitting Time on Directed Graphs. In *AAAI*, Vol. 8. 616–621.

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

[12] S Condamin, O Bénichou, V Tejedor, R Voituriez, and Joseph Klafter. 2007. First-passage times in complex scale-invariant media. *Nature* 450, 7166 (2007), 77–80.

[13] Emanuele Crisostomi, Stephen Kirkland, and Robert Shorten. 2011. A Google-like model of road network dynamics and its application to regulation and control. *Internat. J. Control* 84, 3 (2011), 633–651.

[14] Peter G Doyle. 2009. The Kemeny constant of a Markov chain. *arXiv preprint arXiv:0909.2636* (2009).

[15] Peter M. Fenwick. 1994. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exp.* 24, 3 (1994), 327–336.

[16] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD*. 855–864.

[17] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. 2016. Efficient Algorithms for Spanning Tree Centrality. In *IJCAI*, Vol. 16. 3733–3739.

[18] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. *The collected works of Wassily Hoeffding* (1994), 409–426.

[19] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on evolving graphs with an incremental index-update scheme. *SIGMOD* 1, 1 (2023), 1–26.

[20] Jeffrey J Hunter. 2014. The role of Kemeny's constant in properties of Markov chains. *Communications in Statistics-Theory and Methods* 43, 7 (2014), 1309–1321.

[21] Michael F Hutchinson. 1989. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics-Simulation and Computation* 18, 3 (1989), 1059–1076.

[22] John G Kemeny, J Laurie Snell, et al. 1969. *Finite markov chains*. Vol. 26. van Nostrand Princeton, NJ.

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

[24] Mark Levene and George Loizou. 2002. Kemeny's constant and the random surfer. *The American mathematical monthly* 109, 8 (2002), 741–745.

[25] Shiju Li, Xin Huang, and Chul-Ho Lee. 2021. An efficient and scalable algorithm for estimating Kemeny's constant of a Markov chain on large graphs. In *KDD*. 964–974.

[26] Zihao Li, Dongqi Fu, and Jingrui He. 2023. Everything evolves in personalized pagerank. In *WWW*. 3342–3352.

[27] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, Hongchao Qin, and Guoren Wang. 2023. Efficient personalized pagerank computation: The power of variance-reduced monte carlo approaches. *SIGMOD* 1, 2 (2023), 1–26.

[28] 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. *SIGMOD* 1, 1 (2023), 1–27.

[29] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, and Guoren Wang. 2023. Scalable Algorithms for Laplacian Pseudo-inverse Computation. *arXiv preprint arXiv:2311.10290* (2023).

[30] 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.

[31] 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. *SIGMOD* 2, 3 (2024), 1–27.

[32] Haoyu Liu and Siqiang Luo. 2024. BIRD: Efficient Approximation of Bidirectional Hidden Personalized PageRank. *VLDB* 17, 9 (2024), 2255–2268.

[33] László Lovász. 1993. Random walks on graphs. *Combinatorics, Paul erdos is eighty* 2, 1-46 (1993), 4.

[34] Sam Alexander Martino, João Morado, Chenghao Li, Zhenghao Lu, and Edina Rosta. 2024. Kemeny Constant-Based Optimization of Network Clustering Using Graph Neural Networks. *The Journal of Physical Chemistry B* 128, 34 (2024), 8103–8115.

[35] Dingheng Mo and Siqiang Luo. 2021. Agenda: Robust personalized pageranks in evolving graphs. In *CIKM*. 1315–1324.

[36] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient pagerank tracking in evolving networks. In *KDD*. 875–884.

[37] Rushabh Patel, Pushkarini Agharkar, and Francesco Bullo. 2015. Robotic surveillance and Markov chains with minimal weighted Kemeny constant. *IEEE Trans. Automat. Control* 60, 12 (2015), 3156–3167.

[38] Bibek Paudel and Abraham Bernstein. 2021. Random walks with erasure: Diversifying personalized recommendations on social and information networks. In *WWW*. 2046–2057.

[39] Pan Peng, Daniel Lopatta, Yuichi Yoshida, and Gramoz Goranci. 2021. Local algorithms for estimating effective resistance. In *KDD*. 1329–1338.

[40] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD*. 701–710.

[41] 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.

[42] Purnamrita Sarkar and Andrew W Moore. 2011. Random walks in social networks and their applications: a survey. *Social Network Data Analytics* (2011), 43–77.

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

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

[45] Hanzhi Wang, Zhewei Wei, Junhao Gan, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2022. Edge-based local push for personalized PageRank. *VLDB* 15, 7 (2022), 1376–1389.

[46] Scott White and Padhraic Smyth. 2003. Algorithms for estimating relative importance in networks. In *KDD*. 266–275.

[47] David Bruce Wilson. 1996. Generating random spanning trees more quickly than the cover time. In *STOC*. 296–303.

[48] Feng Xia, Haifeng Liu, Ivan Lee, and Longbing Cao. 2016. Scientific article recommendation: Exploiting common author relations and historical preferences. *IEEE Trans. Big Data* 2, 2 (2016), 101–112.

[49] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. 2019. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence* 4, 2 (2019), 95–107.

[50] Haisong Xia and Zhongzhi Zhang. 2024. Efficient Approximation of Kemeny's Constant for Large Graphs. *SIGMOD* 2, 3 (2024), 1–26.

[51] Wanyue Xu, Yibin Sheng, Zuobai Zhang, Haibin Kan, and Zhongzhi Zhang. 2020. Power-law graphs have minimal scaling of Kemeny constant for random walks. In *WWW*. 46–56.

[52] Mingji Yang, Hanzhi Wang, Zhewei Wei, Sibo Wang, and Ji-Rong Wen. 2024. Efficient algorithms for personalized pagerank computation: A survey. *IEEE TKDE* (2024).

[53] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *VLDB* 13, 5 (2020), 670–683.

[54] Renchi Yang and Jing Tang. 2023. Efficient estimation of pairwise effective resistance. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[55] Serife Yilmaz, Ekaterina Dudkina, Michelangelo Bin, Emanuele Crisostomi, Pietro Ferraro, Roderick Murray-Smith, Thomas Parisini, Lewi Stone, and Robert Shorten. 2020. Kemeny-based testing for COVID-19. *PLOS ONE* 15, 11 (2020), 1–19.

[56] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. 2012. Challenging the Long Tail Recommendation. *VLDB* 5, 9 (2012).

[57] Minji Yoon, Woojeong Jin, and U Kang. 2018. Fast and accurate random walk with restart on dynamic graphs with guarantees. In *WWW*. 409–418.

[58] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *KDD*. 1315–1324.

[59] Xiaohan Zhao, Adelbert Chang, Atish Das Sarma, Haitao Zheng, and Ben Y Zhao. 2013. On the embeddability of random walk distances. *VLDB* 6, 14 (2013), 1690–1701.

[60] Yanping Zheng, Hanzhi Wang, Zhewei Wei, Jiajun Liu, and Sibo Wang. 2022. Instant graph neural networks for dynamic graphs. In *KDD*. 2605–2615.