

# 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. Compared to random walk-based estimators, 2-forest-based estimators yield leads to a better theoretical bound. 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  $\epsilon$  in  $O\left(\frac{\Delta^2 \bar{d}^2}{\epsilon^2} (\tau + n \min(\log n, \Delta))\right)$  time, where  $\tau$  is the time to sample a spanning tree,  $\bar{d}$  is the average degree, and  $\Delta$  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.

## 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 [56], and quantifying global performance costs in road networks [12]. 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 [2, 47], graph clustering [9, 34], and recommendation systems [24, 57], 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 [13, 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 [51], 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 [51], 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 consisting of exactly two disjoint trees, 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) [14]. 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  $\Delta$  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  $\epsilon$  in  $O\left(\frac{\Delta^2 \bar{d}^2}{\epsilon^2} (\phi + n \min(\log n, \Delta))\right)$  time, where  $\phi$  is the time to sample a spanning tree,  $\bar{d}$  is the average degree, and  $\Delta$  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 Algorithm.** 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 dynamic graphs. A detailed theoretical analysis of the proposed algorithm is also provided.

**New Sample Maintenance Methods.** We propose two novel methods, BSM and ISM, to maintain the correctness of spanning

**Table 1: Frequently used notations.**

| Notation                    | Description   |
|-----------------------------|---|
| $G = (V, E)$                | An undirected graph $G$ with node set $V$ and edge set $E$                                      |
| $n, m$                      | The number of nodes and edges in $G$ , respectively   |
| $d(v)$                      | The degree of node $v$  |
| $\kappa(G)$                 | The Kemeny constant of the graph $G$  |
| $T_1 \cup T_2$              | A 2-forest consisting of two trees, where $T_1$ and $T_2$ denote the node sets of the two trees |
| $\tau$                      | A spanning tree of the graph  |
| $\Gamma, \mathbb{F}$        | The set of all spanning trees and 2-forests of the graph, respectively                          |
| $\text{vol}(T_1)$           | The volume of node set $T_1$ , defined as $\text{vol}(T_1) = \sum_{v \in T_1} d(v)$             |
| $\mathcal{P}, \mathcal{P}'$ | A simple, directed path $\mathcal{P}$ and its reverse path $\mathcal{P}'$                       |
| $\text{Sub}(\tau, u)$       | The node set of the subtree rooted at $u$ in tree $\tau$  |
| $\epsilon$                  | The relative error parameter  |

tree samples as the graph updates, substantially reducing computational cost compared to rerunning static algorithms after each update. BSM adjusts the spanning tree samples by pre-computing changes in the sample space, while ISM 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 dynamic 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, ISM 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.

## 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  $\pi$  of a

random walk on undirected graphs is proportional to node degrees, i.e.,  $\pi_i = d_i/2m$ . Table 1 lists the notations that are frequently used in this paper.

The hitting time  $H(i, j)$  is a fundamental measure in the study of random walks [11, 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  $\mathcal{L} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$ , respectively. Let  $0 = \sigma_1 \leq \dots \leq \sigma_n$  be the eigenvalues of  $\mathcal{L}$ , with the corresponding eigenvectors  $u_1, \dots, u_n$ . 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.,

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

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 Methods.** 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. [52] proposed ApproxKemeny based on Hutchinson's Monte Carlo [21]. It approximates the trace of  $\mathcal{L}^\dagger$ , and thus KC as:  $\kappa(G) = \text{Tr}(\mathcal{L}^\dagger) \approx \frac{1}{M} \sum_{i=1}^M x_i^\top \mathcal{L}^\dagger x_i$ , 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  $\mathcal{L}^\dagger$ , ApproxKemeny transforms the problem into solving Laplacian linear systems:  $x_i^\top \mathcal{L}^\dagger x_i = \|B L^\dagger y_i\|^2$ , 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 ApproxKemeny are limited by the performance of the specific Laplacian solver used.

**Truncated Random Walk Based Methods.** DynamicMC [25] and RefinedMC [51] 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,

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

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 [51].

**Loop-Erased Random Walk (LERW) Based Methods.** Recently, the relationship between  $\mathcal{L}^{-1}$  with  $(I - P_v)^{-1}$  has been independently explored by [51] 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:

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

Both methods, ForestMC proposed in [51] and LEwalk proposed in [29], employ loop-erased random walks (LERWs) to approximate  $\text{Tr}(I - P_v)^{-1}$ , but differ in their computation of  $(\mathcal{L}^\dagger)_{vv}/\pi_v$ : ForestMC [51] 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 [48].

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  $\text{Tr}(I - P_v)^{-1}$ , 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 [4, 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  $O(\epsilon^{-2} \Delta^2 d_{\max}^2 \log^3 n \cdot \text{Tr}(I - P_v)^{-1})$ , 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 map 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).

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

where  $\Gamma$  denotes the set of all spanning trees of  $G$ ,  $r$  is an arbitrarily fixed node,  $\text{vol}(T_1) = \sum_{u \in T_1} d(u)$  represents the volume of the tree containing  $r$ , and  $\mathbb{F}_{r|u}$  is the set of 2-forests in which  $r$  and  $u$  belong to different trees.

PROOF.

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

The first equality follows from Corollary 1.7 in [10], which expresses KC 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  $\text{vol}(T)$ .  $\square$

EXAMPLE 1. As shown in Fig. 1, (a) illustrates a graph  $G$ , (b) displays all its spanning trees, and (c) shows all its 2-forests. Assume node  $v_1$  is selected as  $r$ . Then,  $\mathbb{F}_{v_2|v_1} = \{\mathcal{F}_1, \mathcal{F}_3\}$  and  $\mathbb{F}_{v_3|v_1} = \{F_1, F_2\}$ .

For any given 2-forest, we denote the node set of the tree containing the root  $r$  as  $T_1$ , and the other as  $T_2$ . For example, in  $\mathcal{F}_1$ , we have  $T_1 = \{v_1\}$  and  $T_2 = \{v_2, v_3\}$ . Hence,  $\text{vol}(T_1) = d(v_1) = 2$ . Note that this degree refers to  $v_1$ 's degree in the graph  $G$ , not in the forest.

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  $|\Gamma|$ . 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.

An interesting observation about the 2-forests in  $\mathbb{F}_{r|u}$  is that they can be obtained from a spanning tree by deleting any edge along the path between  $u$  to  $v$ , thereby ensuring that  $u$  and  $v$  lie in different components. Consequently, a single spanning tree yields  $|P_{u \rightarrow v}|$  valid 2-forests, where  $|P_{u \rightarrow v}|$  is the number of edges on the  $u$ - $v$  path.

However, the sets of 2-forests generated from different spanning trees overlap. For example, in Fig. 1, both  $\tau_1$  and  $\tau_2$  can produce  $\mathcal{F}_1$  by deleting one edge. This redundancy prevents a direct correspondence between  $\mathbb{F}_{r|u}$  and  $\Gamma$ . To resolve this issue, we propose a novel key technique called Path Mapping which can systematically avoids such overlaps. The formal definition of Path Mapping is provided 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  $\tau$ , there exists a unique path between  $u$  and  $r$ , denoted by  $\mathcal{P}_{u \rightarrow r}^{(\tau)}$ . By removing all edges shared by both  $\mathcal{P}$  and  $\mathcal{P}_{u \rightarrow r}^{(\tau)}$ , 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  $\mathbb{F}^{\mathcal{P}}(\tau)$ , i.e.,  $\mathbb{F}^{\mathcal{P}}(\tau) = \left\{ \tau \setminus (i, j) \mid (i, j) \in \mathcal{P} \wedge (i, j) \in \mathcal{P}_{u \rightarrow r}^{(\tau)} \right\}$ ,
- *Reverse Path Mapping*: if the removed edge has opposite direction in two path, the obtained 2-forest set is denoted as  $\mathbb{F}^{\mathcal{P}'}(\tau)$ , i.e.,  $\mathbb{F}^{\mathcal{P}'}(\tau) = \left\{ \tau \setminus (i, j) \mid (i, j) \in \mathcal{P} \wedge (j, i) \in \mathcal{P}_{u \rightarrow r}^{(\tau)} \right\}$ .

EXAMPLE 2. Fig. 3 illustrates an example of path mapping. To convert a spanning tree into forests in  $\mathbb{F}_{v_2|v_1}$ , we first identify a simple path from  $v_2$  to  $v_1$ , such as  $\mathcal{P}_{v_2 \rightarrow v_1} : v_2 \rightarrow v_3 \rightarrow v_1$ . Then, for a given spanning tree, e.g.,  $\tau_1$ , we locate the path from  $v_2$  to  $v_1$  within  $\tau_1$ , that is  $\mathcal{P}_{v_2 \rightarrow v_1}^{(\tau_1)} : v_2 \rightarrow v_3 \rightarrow v_1$ . We observe that two edges in  $\mathcal{P}_{v_2 \rightarrow v_1}^{(\tau_1)}$  also appear in  $\mathcal{P}_{v_2}$  and have the same direction. We can obtain two 2-forests,  $\mathcal{F}_3$  and  $\mathcal{F}_1$ , by removing  $(v_2, v_3)$  and  $(v_3, v_1)$  respectively.

As for  $\tau_3$ , the path from  $v_2$  to  $v_1$  is  $\mathcal{P}_{v_2 \rightarrow v_1}^{(\tau_3)} : v_2 \rightarrow v_1$ . Although the edge  $(v_1, v_3)$  also exists in  $\mathcal{P}_{v_2}$  and  $\tau_3$ , it is not part of  $\mathcal{P}_{v_2 \rightarrow v_1}^{(\tau_3)}$ , and thus cannot produce a 2-forest. Therefore, applying path mapping with  $\mathcal{P}_{v_2}$  to  $\tau_3$  results in an empty set.

With Path Mapping, we restrict the deletable edges of a spanning tree to a specific path to reduce redundancy, yet this remains insufficient. To further address this, we distinguish two cases depending

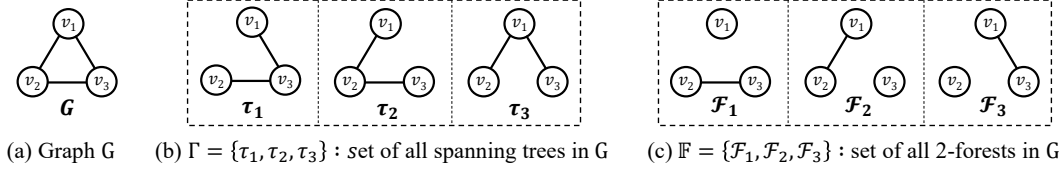


Figure 1: An example graph, its spanning trees and 2-forests

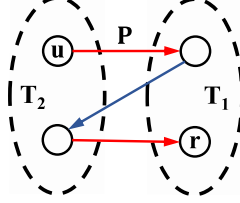


Figure 2: Proof of Lemma 3.5

on whether the deleted edge aligns with the path orientation, introducing forward and reverse Path Mapping. This refinement further eliminates redundancy and ultimately enables our goal: establishing a precise correspondence between  $\Gamma$  and  $\mathbb{F}_{r|u}$  through Path Mapping.

The following theorem formalizes this connection by showing that, given any simple path from  $u$  to  $r$ , the set of 2-forests obtained from all spanning trees in  $\Gamma$  via forward Path Mapping, minus those obtained via reverse Path Mapping, is exactly  $\mathbb{F}_{r|u}$ .

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

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

**PROOF SKETCH.** The forward path mapping covers all of  $\mathbb{F}_{r|u}$ , since there must be an edge in  $\mathcal{P}$  connecting two trees in any 2-forests. Moreover, for each 2-forest, forward mapping produces exactly one more instance than reverse mapping, as  $\mathcal{P}$  contains precisely one additional edge from the tree of  $u$  to that of  $r$ .  $\square$

**LEMMA 3.4.** *Given an arbitrary simple path  $\mathcal{P}$  from  $u$  to  $r$ , the following equation holds:*

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

**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. 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  $\mathcal{P}$  starts from  $u$  and ends at  $r$ , there must be at least one edge  $e$  in  $\mathcal{P}$  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  $\mathcal{P}$  fails to connect  $u$  and  $r$ . Adding the edge 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  $\mathbb{F}^{\mathcal{P}}(T_1 \cup T_2 \cup e)$ .  $\square$

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

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

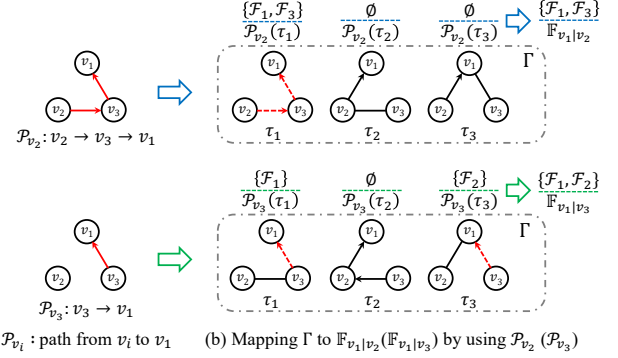


Figure 3: Illustration of Transforming Spanning Trees to 2-Forests Using Path Mapping

**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  $\tau$  is composed of  $T_1 \cup T_2$  and an edge  $e$  in  $\mathcal{P}$ , which leaves from  $T_2$  and enters to  $T_1$  (as the red edges in Fig. 2). Then  $T_1 \cup T_2$  appears in the set obtained by path mapping  $\tau$ , i.e.,  $T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}}(T_1 \cup T_2 \cup (e_1, e_2))$ , for each  $(e_1, e_2) \in \mathcal{P}$ , satisfying  $e_1 \in T_2, e_2 \in T_1$ . Similarly  $\mathbb{F}^{\mathcal{P}'}(\tau')$  has  $T_1 \cup T_2$  if  $\tau'$  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. 2). Therefore

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

The last equality holds because  $\mathcal{P}$  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  $\mathcal{P}$ .  $\square$

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

**EXAMPLE 3.** Fig. 3 illustrates the mapping from the spanning tree set  $\Gamma$  to the 2-forest set  $\mathbb{F}_{v_1|v_2}$ . By selecting a root node and fixing a path from another node to the root, each spanning tree in  $\Gamma$  can be mapped to one or more 2-forests in  $\mathbb{F}_{v_1|v_2}$  by removing edges along the path. For instance, in  $\tau_1$ , removing edges along the fixed path generates  $F_1$  and  $F_3$ , while  $\tau_2$  and  $\tau_3$  yield no mappings. Combining the mappings from all trees exactly recovers  $\mathbb{F}_{v_1|v_2}$ , ensuring that sampling a tree uniformly from  $\Gamma$  induces a uniform distribution over the corresponding 2-forests.

**Discussion.** In this way, we successfully establish a connection between  $\Gamma$  and  $\mathbb{F}_{r|u}$  through path mapping, which lays the foundation for designing a feasible sampling algorithm. Previously, it was difficult to directly sample 2-forests. Now, we can first sample uniform random spanning trees (USTs) and then apply path mapping to transform each spanning tree sample into multiple (or possibly zero) 2-forests. As a result, each sampled 2-forest is obtained with probability  $1/|\Gamma|$ . In the next section, we will present the sampling algorithm in detail and provide the corresponding theoretical analysis.

### 3.2 The Tree-To-Forest Algorithm

In this section, we introduce TTF (Tree-To-Forest), a new sampling-based algorithm for estimating the Kemeny constant. The key idea is to first sample USTs and then transform UST samples into 2-forests by path mapping, which allow us to estimate KC according to Eq. (4). We begin by outlining the algorithmic intuition and establish its correctness, and then present the pseudo-code, implementation details, and complexity analysis.

**High-Level Idea of TTF.** (1) **Identify the root and paths.** We select an arbitrary node  $r$  as the root and construct a breadth-first search (BFS) tree rooted at  $r$ . This BFS tree provides a set of simple paths  $\mathcal{P}_u$  from each node  $u$  to  $r$ . (2) **Sample USTs.** We efficiently sample USTs using Wilson's algorithm [48], which form the basis of our estimator. (3) **From tree to forest.** Each sampled UST is then converted into 2-forests using forward and reverse path mapping with  $\{\mathcal{P}_u \mid u \neq r\}$ . (4) **Estimate KC.** Finally, we estimate KC using Eq. (4) based on the obtained 2-forests.

The core of our algorithm lies in steps (3) and (4). Importantly, we do not need to explicitly transform sampled USTs into 2-forests. Instead, it suffices to compute the contributions that these 2-forests would make to KC. Therefore, we merge the two steps and directly estimate KC from the sampled USTs. We propose two approaches for this task: a naive method and an optimized method.

**Traverse Each Path (Naive Method).** For each sampled UST  $\tau$ , we traverse the path from each node  $u$  to the root  $r$  within the sampled tree, and compare its edges with the pre-determined path  $\mathcal{P}_u$  from step (1). Each matching edge with same orientation indicates a valid 2-forest in  $\mathbb{F}^{\mathcal{P}_u}(\tau)$  (or  $\mathbb{F}^{\mathcal{P}'_u}(\tau)$  when edge has opposite orientation). For each valid 2-forest, we accumulate the volume of the component containing  $r$ , i.e.  $\text{vol}(T_1)$ , and multiply it by  $d(u)$  to contribute to the KC estimate. Repeating this process for all nodes  $u \neq r$  yields the contribution of  $\tau$ , denoted by  $f(\tau)$

**THEOREM 3.6 (CORRECTNESS OF NAIVE METHOD).** *Given  $\omega$  UST samples and a fixed root  $r$  with paths  $\{\mathcal{P}_u \mid u \neq r\}$ . Let  $\hat{\tau}_i$  denote the  $i$ -th sampled UST. Then,  $\tilde{\kappa} = \frac{1}{2m\omega} \sum_{i=1}^{\omega} f(\hat{\tau}_i)$  is an unbiased estimator of  $\kappa(G)$ , where*

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

**PROOF.** Based on Theorem 3.1 and Theorem 3.3, we can represent the set  $\mathbb{F}_{r|u}$  by applying the path mapping technique to the spanning tree set  $\Gamma$ . With this representation, KC can be expressed

as:

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

□

**Traverse by Depth-First Search (Optimized Method).** We further enhance traversal efficiency by maintaining an auxiliary variable `vol_sum` for each node and performing a DFS on the sampled UST  $\tau$ . When visiting a node  $v$ , we update `vol_sum` by adding or subtracting  $2m - \text{vol}(\text{Sub}(\tau, v))$  if the edge connecting  $v$  to its parent  $p(v)$  in  $\tau$  also appears in the BFS tree obtained in step(1), where  $\text{Sub}(\tau, v)$  denotes the subtree rooted at  $v$  in  $\tau$ . After the update, we query the `vol_sum` at  $v$ , multiply it by  $d(v)$  to contribute to the KC estimate, recursively continue the DFS on its children, and finally restore the affected `vol_sum` values by undoing the update.

**THEOREM 3.7 (CORRECTNESS OF OPTIMIZED METHOD).** *Given an arbitrary spanning tree  $\tau$  and a fixed spanning tree  $\tau_0$ , let  $r$  be the root of both trees. Let  $\mathcal{P}_u$  denote the path from node  $u$  to  $r$  in  $\tau_0$ . Then, the following equality holds:*

$$\sum_{u \neq r} \sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u}(\tau)} \text{vol}(T_1) = \sum_{\substack{(v, p(v)) \in \tau_0 \\ u \in S_1}} (2m - \text{vol}(\text{Sub}(\tau, v))), \quad (5)$$

$$\sum_{u \neq r} \sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}'_u}(\tau)} \text{vol}(T_1) = \sum_{\substack{(p(v), v) \in \tau_0 \\ u \in S_2}} (2m - \text{vol}(\text{Sub}(\tau, v))), \quad (6)$$

where  $\text{Sub}(\tau, v)$  is the node set in the subtree rooted at  $v$  within  $\tau$ , and  $S_1 = \text{Sub}(\tau, v) \cap \text{Sub}(\tau_0, v)$ ,  $S_2 = \text{Sub}(\tau, v) \cap \text{Sub}(\tau_0, p(v))$ .  $p(v)$  is the parent node of  $v$  in  $\tau$  and therefore each edge  $(v, p(v))$  is directed towards the root  $r$  in  $\tau$ .

**PROOF.** For each node  $u$ , we apply the forward path mapping  $\mathcal{P}_u$ , which gives

$$\sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u}(\tau)} \text{vol}(T_1) = \sum_{\substack{(v, p(v)) \in \tau_0 \cap \tau \\ u \in \text{Sub}(\tau, v) \cap \text{Sub}(\tau_0, v)}} (2m - \text{vol}(\text{Sub}(\tau, v))).$$

By the definition of path mapping, to obtain a valid 2-forest in  $\mathbb{F}^{\mathcal{P}_u}(\tau)$ , the deleted edge  $(v, p(v))$  in  $\tau$  must appear on both paths from  $u$  to  $r$  in the two spanning trees  $\tau$  and  $\tau_0$ . This implies that  $u$  must belong to the intersection of the subtrees of  $v$  in  $\tau$  and  $\tau_0$ , denoted as  $S_1 = \text{Sub}(\tau, v) \cap \text{Sub}(\tau_0, v)$ . Therefore, summing over all  $u \neq r$ , we obtain

$$\sum_{u \neq r} \sum_{T_1 \cup T_2 \in \mathbb{F}^{\mathcal{P}_u}(\tau)} \text{vol}(T_1) = \sum_{\substack{(v, p(v)) \in \tau \cap \tau_0 \\ u \in S_1}} (2m - \text{vol}(\text{Sub}(\tau, v))).$$

**Algorithm 1: TTF**


---

**Input:** A graph  $G = (V, E)$ , the sample size  $\omega$ , the root node  $r$   
**Output:**  $\tilde{\kappa}$  as the estimation of Kemeny Constant

```

1  $\tilde{\kappa} \leftarrow 0$ ;
2 Generate  $\tau_0$  using BFS with root  $r$ ;
3  $\text{DFS}_{\text{in}}, \text{DFS}_{\text{out}} \leftarrow \tau_0$ ;
4  $\text{vol\_sum} \leftarrow \text{BIT}(|V|)$ ;
5 for  $i \leftarrow 1$  to  $\omega$  do
6    $\tau_i \leftarrow \text{wilson}(G, r)$ ;
7   for each node  $u \in V$  do
8      $\text{vol}[u] \leftarrow \text{the volume of } \text{Sub}(\tau_i, u)$ ;
9    $\tilde{\kappa} \leftarrow \tilde{\kappa} + \text{DFS}(r, \tau_i, \tau_0)$ ;
10 return  $\tilde{\kappa} / (2m \cdot \omega)$ ;
11 Function  $\text{DFS}(v, \tau, \tau_0)$ :
12   Let  $p(v)$  be the parent node of  $v$  in  $\tau$ ;
13   if  $(v, p(v)) \in \tau_0$  then
14      $\text{vol\_sum.Add}(\text{DFS}_{\text{in}}[v], \text{DFS}_{\text{out}}[v], 2m - \text{vol}[v])$ ;
15   else if  $(p(v), v) \in \tau_0$  then
16      $\text{vol\_sum.Add}(\text{DFS}_{\text{in}}[p(v)], \text{DFS}_{\text{out}}[p(v)], \text{vol}[v] - 2m)$ ;
17    $\tilde{\kappa} \leftarrow d(v) \cdot \text{vol\_sum.Query}(\text{DFS}_{\text{in}}[v])$ ;
18   for each node  $i \in \text{Child}_\tau(v)$  do
19      $\tilde{\kappa} \leftarrow \tilde{\kappa} + \text{DFS}(i, \tau, \tau_0)$ ;
20   if  $(v, p(v)) \in \tau_0$  then
21      $\text{vol\_sum.Add}(\text{DFS}_{\text{in}}[v], \text{DFS}_{\text{out}}[v], \text{vol}[v] - 2m)$ ;
22   else if  $(p(v), v) \in \tau_0$  then
23      $\text{vol\_sum.Add}(\text{DFS}_{\text{in}}[p(v)], \text{DFS}_{\text{out}}[p(v)], \text{vol}[v] - 2m)$ ;
24   return  $\tilde{\kappa}$ ;

```

---

Similar equation holds for the reverse path mapping,

$$\sum_{u \neq r} \sum_{T_1 \cup T_2 \in \mathbb{P}'_u(\tau)} \text{vol}(T_1) = \sum_{\substack{(p(v), v) \in \tau_0 \\ u \in S_2}} (2m - \text{vol}(\text{Sub}(\tau, v))).$$

□

The pseudo-code of TTF is presented in Algorithm 1. Given a graph  $G$ , a root node  $r$ , and a sample size  $\omega$ , the algorithm begins by constructing a fixed spanning tree  $\tau_0$  using breadth-first search (BFS), followed by computing its DFN (Lines 2–3). An auxiliary data structure  $\text{vol\_sum}$  is initialized as a BIT of length  $|V|$  (Line 4). It then samples  $\omega$  USTs using Wilson algorithm [48] (Line 6). For each sampled UST  $\tau_i$ , the algorithm first computes the volume for all subtrees (Lines 7–8), and then invokes the DFS function to accumulate contributions to the KC estimator (Line 9). Finally, the algorithm returns  $\tilde{\kappa} / (2m \cdot \omega)$  as the approximation of KC (Line 10).

The core procedure DFS operates as follows. Let  $p(v)$  denote the parent node of the visited node  $v$  in  $\tau$ . The algorithm checks whether the edge  $(v, p(v))$  exists in  $\tau_0$ . If so, it increments  $\text{vol\_sum}$  for all nodes in the of  $\tau_0$  rooted at  $v$  by  $2m - \text{vol}[v]$  (Lines 13–14). Conversely, if the reverse edge  $(p(v), v)$  exists in  $\tau_0$ , it decrements  $\text{vol\_sum}$  for all nodes in  $\text{Sub}(\tau_0, p(v))$  by  $2m - \text{vol}[v]$  (Lines 15–16). After these updates, the algorithm queries  $\text{vol\_sum}[v]$  and incorporates its value into the KC estimation (Line 17). Then, it recursively invokes DFS on each child node of  $v$  (Lines 18–19). Before returning from DFS, related changes made to  $\text{vol\_sum}$  must be

revert to maintain correctness for subsequent computations (lines 20–23).

**Implementation Details.** We describe the key implementation details of our approach, including the traversal strategy and efficient management of the auxiliary structure  $\text{vol\_sum}$ .

**1. Why Use DFS.** Based on Theorem 3.7, in principle, any traversal method can be used to visit all nodes. For each node  $v$ , if the edge  $(v, p(v))$  connecting  $v$  to its parent also exists in the fixed spanning tree  $\tau_0$ , we need to identify the intersection of the subtrees rooted at  $v$  in  $\tau$  and  $\tau_0$ , and update the  $\text{vol\_sum}$  for these nodes by  $2m - \text{vol}(\text{Sub}(\tau, v))$ , or decrease it if the edge  $(v, p(v))$  has opposite directions in the two trees. Explicitly comparing subtrees to find their intersection is costly, with potential complexity  $O(n)$ , which motivates the choice of DFS traversal. By using DFS, we can update the  $\text{vol\_sum}$  values for all nodes in the subtree of  $v$  in  $\tau_0$  without explicitly identifying intersection nodes. While this may temporarily update some nodes incorrectly, these nodes are not visited during the DFS call for  $v$  in  $\tau$ , and restoring  $\text{vol\_sum}$  after DFS ensures correctness when each node is visited.

**2. Implementation of Auxiliary Structure  $\text{vol\_sum}$ .** To efficiently implement these updates, we combine Depth-First Numbering (DFN) with Binary Index Tree (BIT, or Fenwick Tree [14]).

**2.1 Depth-First Numbering.** DFN records the entry and exit times of each node during DFS traversal, denoted as  $\text{DFS}_{\text{in}}$  and  $\text{DFS}_{\text{out}}$ . With this notation, the set of nodes in a subtree rooted at  $u$  can be precisely characterized by the interval  $[\text{DFS}_{\text{in}}[u], \text{DFS}_{\text{out}}[u]]$ . Consequently, updating all nodes within a subtree is equivalent to updating the nodes corresponding to a contiguous interval of DFN. For instance, as illustrated in Fig. 4, the first column shows the entry and exit times of nodes  $v_2, v_3$ , and  $v_4$ . The DFN of the tree  $\tau_0$  is  $v_1 \rightarrow v_4 \rightarrow v_2 \rightarrow v_3$ . The subtree rooted at  $v_2$  corresponds to the interval  $[3, 4]$ , which exactly covers the nodes  $v_2$  and  $v_3$ .

**2.2 Binary Index Tree.** For each visited node, two updates (lines 13–16 and lines 20–23) and one query (line 17) are required. Standard arrays allow  $O(1)$  queries but  $O(n)$  updates, while difference arrays allow  $O(1)$  interval updates but  $O(n)$  queries. The BIT makes a balance so that when storing a difference array using BIT, it supports interval updates and point queries in  $O(\log n)$  time:

- $\text{Add}(l, r, v)$ : add value  $v$  to all elements in the range  $[l, r]$ .
- $\text{Query}(i)$ : query the  $i$ -th element value.

This combination of DFS traversal, DFN ordering, and BIT enables efficient and correct maintenance of the  $\text{vol\_sum}$  auxiliary structure throughout the traversal process.

**EXAMPLE 4.** Fig. 4 illustrates how DFS and BIT efficiently maintain  $\text{vol\_sum}$  and compute  $f(\tau)$  for sampled USTs. The first column shows the fixed tree  $\tau_0$  with its DFN (1 – 4 – 2 – 3), the second column displays the sampled tree  $\tau$  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  $\tau_0$ . Using the BIT, we efficiently update  $\text{vol\_sum}$  by adding  $2m - \text{vol}[4]$  to all nodes in subtree of  $v_4$  in  $\tau_0$ , using  $O(\log n)$  time. After updating  $\text{vol\_sum}$ , we query  $\text{vol\_sum}[v_4]$ , which corresponds to the second element in DFN. Since  $\text{vol\_sum}$  is maintained as a differential array, the actual value is computed as a prefix sum using BIT, also in  $O(\log n)$  time.



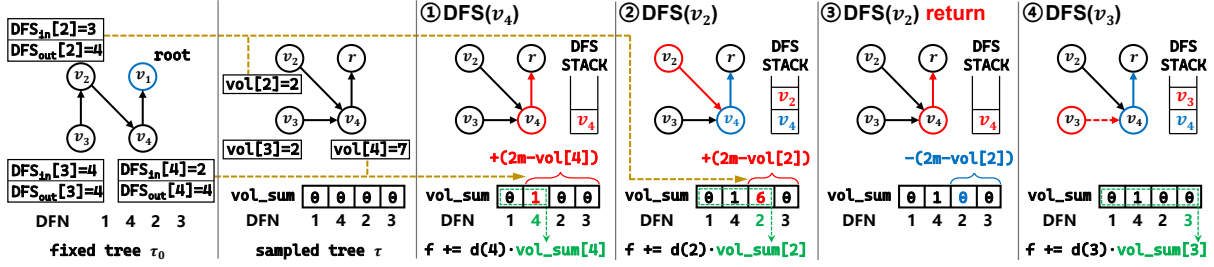


Figure 4: Illustration of the DFS process

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,  $\text{vol\_sum}[2]$  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  $\tau$  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  $\text{vol\_sum}[3]$  in the fourth step.

**THEOREM 3.8 (TIME COMPLEXITY OF ALGORITHM 1).** *The time complexity of Algorithm 1 is*

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

**PROOF.** The time complexity of Wilson Algorithm for generating a UST is  $O(\text{Tr}((I - P_r)^{-1}))$ , 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  $\text{vol\_sum}$  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  $\Delta$  such that  $\Delta < \log n$ , the computation of  $\text{vol\_sum}$  can be limited to within  $\Delta$  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.  $\square$

**THEOREM 3.9 (ERROR BOUND OF ALGORITHM 1).** *If the sample size satisfies  $\omega \geq \frac{8m^2 \Delta_G^2 \log(2/p_f)}{n^2 \epsilon^2}$ , then Algorithm 1 outputs an estimate  $\tilde{\kappa}$  such that  $|\tilde{\kappa} - \kappa| \leq \epsilon \kappa$  with probability at least  $1 - p_f$ .*

**PROOF.** For each sampled tree  $\tau$ , the number of 2-forests mapped by  $\tau_0$  for a node  $u$  is bounded by the distance from  $u$  to the root  $r$ . If we construct  $\tau_0$  such that its depth is minimized, this distance is at most  $\Delta$ , the diameter of the graph. Additionally, for each forest, the volume  $\text{vol}(T_1)$  is bounded by  $2m$ . Consequently,  $f(\tau)$  can be bounded as  $f(\tau) \leq \sum_u d(u) \cdot 2m\Delta \leq 4m^2 \Delta$ .

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

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

This completes the proof.  $\square$

By combining Theorem 3.8 and Theorem 3.9, we conclude that TTF can compute KC in  $O\left(\frac{\Delta^2}{\epsilon^2}(\phi + n \min(\log n, \Delta))\right)$  time to achieve a relative error  $\epsilon$ , where  $\phi = \text{Tr}(I - P_r)^{-1}$  is the time required for UST sampling. Our experiments indicate that time complexity is near-linear in practice. Algorithm 1 details the optimized estimation in  $O(n \log n)$  time. For low-diameter graphs, a naive  $O(n\Delta)$  implementation can also be used, which matches the computational cost of SpanTree [29]. As shown 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 in real-world datasets.

**Discussion.** Compared to prior SOTA methods, our approach offers stronger theoretical guarantees, particularly over LERW-based methods. Although ForestMC [51] provides an error bound, it requires  $O(\epsilon^{-2} \Delta^2 d_{\max}^2 \phi \log^3 n)$  samples to achieve a relative error  $\epsilon$ , where the factor  $d_{\max}^2$  is impractical for most real-world datasets. For instance, the diameter of DBLP used in our experiments is 21, and its maximum degree is 343, which results in an astronomical sample size of  $O(d_{\max}^2) = O(343^{42}) \approx 10^{106}$ , rendering the bound meaningless. The fundamental reason LERW-based methods lack strong theoretical guarantees lies in the high variance of LERWs. Consider a line graph: if a random walk starts at one endpoint and terminates upon reaching the other, the number of steps ranges from at least  $n - 1$  to potentially infinity. This makes it challenging to establish tight bounds for arbitrary graphs, and performance becomes highly dependent on the graph structure. In contrast, our bound depends on the graph diameter, which is typically small in real-world networks, as demonstrated in our experiments.

We identify two closely related works [10, 29]. Chung and Zeng [10] represent entries of the Laplacian pseudoinverse and normalized Laplacian pseudoinverse using rooted 2-forests, which further leads



to a series of forest-based formulas, including KC. Our Theorem 3.1 is derived as a direct extension of their Corollary 1.7. Liao et al. [29] propose the SpanTree method, which to the best of our knowledge is the only existing approach that also estimates the Kemeny constant via UST sampling, and its computational framework is similar to our TTF. We next highlight the distinctions and innovations of our approach compared with these two works.

**Comparison to [10].** Corollary 1.7 in [10] provides a clean and elegant forest formula for KC, offering a combinatorial interpretation in terms of enumerating 2-forests. However, its practical utility is limited, since uniformly sampling 2-forests is computationally intractable on large-scale graphs. Unlike spanning trees, a 2-forest requires an exact partition of the node set into two disjoint connected components, and 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. To the best of our knowledge, no efficient method for direct sampling of 2-forests currently exists, which prevents this formula from being used in practice. This limitation actually motivates our work: by introducing a new formula for KC (Eq. (4)) together with the path mapping technique, we make the theoretical result of [10] practically computable and enable efficient 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 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 (BSM)

After a new edge is added or removed from the graph, the spanning trees of the updated graph  $G'$  largely overlap with those of the original graph, with the only difference being the additional (or missing) spanning trees that involve the updated edge  $e$ , assuming both graphs remain connected. Hence, a natural idea is to adapt the sampled USTs by supplementing or removing the spanning trees that differ between  $G$  and  $G'$ . The resulting set of “valid” spanning trees can then be directly 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  $\Gamma$  denote the spanning tree set of the original graph  $G$ , and let  $\Gamma'$  represent the corresponding

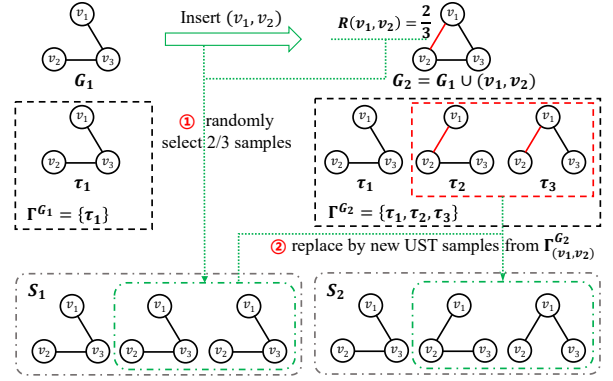


Figure 5: Illustration of Basic Samples Maintenance for Edge Insertion

set of the updated graph  $G'$  after inserting the edge  $e$ . The set  $\Gamma'$  comprises all spanning trees in  $\Gamma$ , along with additional spanning trees that include the newly inserted edge  $e$ . We define the subset of all these new spanning trees in  $\Gamma'$  that contain  $e$  as  $\Gamma'_e$ .

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.,  $|\Gamma'_e|/|\Gamma'| = R(e)$  [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  $\Gamma$ . Our goal is to incorporate the additional spanning trees in  $\Gamma'_e$  that appear after the update. Using the relationship described above, we can determine the expected proportion of these new spanning trees in the samples for  $G'$ . By replacing part of the old samples with these added USTs, we obtain an unbiased estimation on the updated graph.

**EXAMPLE 5.** 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 samples  $S_1$  and replace them with the same number of new USTs that include  $(v_1, v_2)$ . This process is performed using a modified version of Wilson’s algorithm. Finally, we utilize the updated samples  $S_2$  to estimate the KC of  $G_2$ .

**THEOREM 4.1 (CORRECTNESS OF BSM FOR INSERTION CASE).**

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

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

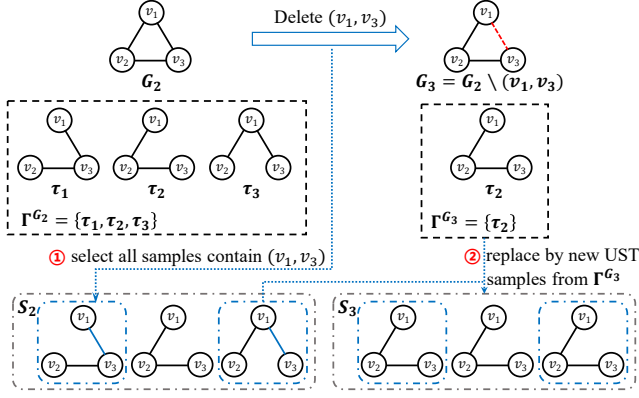


Figure 6: Illustration of BSM for Edge Deletion

PROOF.

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

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  $|\Gamma'_e|/|\Gamma'| = R(e)$ .  $\square$

**Remarks.** Note that the updated samples set are not yet followed the uniform distribution  $U(\Gamma')$ . Nevertheless, Theorem 4.1 guarantees that the estimator remains unbiased when we uniformly sample from both  $\Gamma'$  and  $\Gamma'_e$  and apply appropriate reweighting. Although the error bound may change, we assume that the sample size does not need adjustment under small updates in real-world datasets. Our experiments in Section 5 confirm that this method effectively preserves estimation accuracy in practice.

**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  $\Gamma$  to represent the case without edge  $e$ , and  $\Gamma'$  to represent the spanning trees of the graph that contains the edge  $e$ . However, this time we aim to use samples from  $U(\Gamma')$  to approximate the results under  $U(\Gamma)$ .

**EXAMPLE 6.** 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

**Algorithm 2: BSM**


---

**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      $\tau' \leftarrow \text{wilson}(G', (u, v));$ 
8     Redirect edges in  $\tau'$  to point towards  $r$ ;
9      $f(\tau') \leftarrow \text{DFS}(r, \tau', \tau_0);$ 
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      $\tau' \leftarrow \text{wilson}(G', r);$ 
15      $f(\tau') \leftarrow \text{DFS}(r, \tau', \tau_0);$ 
16      $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau')/|S|, S' \leftarrow S' \cup \tau';$ 
17 return  $\tilde{\kappa}', S';$ 

```

---

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  $G_3$ .

**THEOREM 4.2 (CORRECTNESS OF BSM FOR DELETION CASE).**

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

PROOF.

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

 $\square$ 

The pseudo-code for the basic sample maintenance (BSM) 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 [5, 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 BSM).** *BSM returns an estimate  $\tilde{\kappa}'$  that satisfies the relative-error guarantee. The overall time complexity of BSM is*

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

where  $e = (u, v)$  is the inserted (or deleted) edge,  $R(e)$  denotes its effective resistance in the graph that includes  $e$ , and  $P_e$  is the transition matrix with the  $u$ -th and  $v$ -th rows and columns removed.

**PROOF.** To establish the correctness of BSM, 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  $\Delta$ , which is negligible when considering the relative error, as  $\kappa$  is larger than  $n$  and thus significantly greater than  $\Delta$ . 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  $O(\text{Tr}(I - P_e)^{-1})$  instead of  $O(\text{Tr}(I - P_r)^{-1})$ . 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.  $\square$

**Discussion.** Compared to Algorithm 1, BSM selectively updates only an  $R(e)$  fraction of the samples, and by appropriately reweighting the old and new contributions, the estimator remains unbiased. This strategy substantially reduces the number of samples to be generated and the associated computations. As shown in Table 2, the effective resistance is typically small (can be below 0.1 in social networks), further lowering the computational cost.

## 4.2 Improved Sample Maintenance (ISM)

Although the basic maintenance method significantly reduces computation compared to static algorithms, replacing a portion of the old samples still results in some loss of information. Can we maintain an unbiased estimation while maximizing the utility of the computations contributed by these replaced trees? To address this question, we proposed Improved Sample Maintenance (ISM) methods, discussing the insertion and then deletion scenarios in order.

**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.

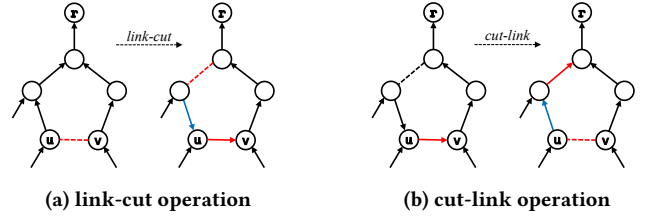


Figure 7: Illustration of link-cut and cut-link operation

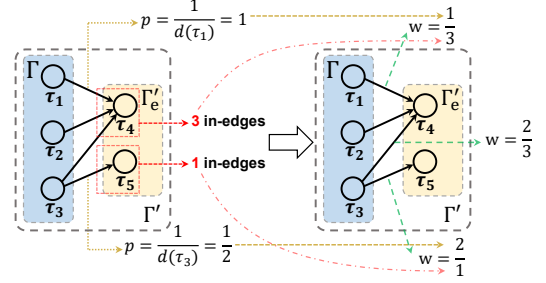


Figure 8: Link-Cut Bipartite Graph  $\mathcal{B}$

Specifically,  $\text{link-cut}(\tau, e)$  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.

However, the spanning trees sampled by *link-cut* are not uniformly distributed. To better understand this, consider an abstract graph 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  $\Gamma$  and  $\Gamma'_e$ , denoted as  $\mathcal{B}$ . Let  $\mathcal{N}_{\mathcal{B}}(\tau)$  represents the set of neighbors of a tree  $\tau$ , corresponding to the spanning trees that can be obtained by performing a link-cut operation on  $\tau$ . 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  $\tau$ .

Clearly, if we randomly transform a tree in  $\Gamma$  into one of its neighbors with equal probability  $1/d(\tau)$  using the link-cut operation, some trees in  $\Gamma'_e$  will have higher selection probabilities than others. Fortunately, this bias can be corrected by appropriately weighting the tree. 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 the estimator remains unbiased, even though the transformation probabilities are not uniform.

**EXAMPLE 7.** Figure 8 illustrates a link-cut bipartite graph for an evolving graph. Here  $\tau_1$  to  $\tau_5$  represent all spanning trees in the updated graph. Among them,  $\tau_1$  to  $\tau_3$  belong to the original tree set  $\Gamma$ , while

$\tau_4$  and  $\tau_5$  are new trees introduced by the update. Each edge in the bipartite graph corresponds to a possible transformation between trees via the link-cut operation.

Due to differences in outdegree, selecting an outgoing edge from  $\tau_1$  is more likely than from  $\tau_3$ . Moreover,  $\tau_4$  can be reached from  $\tau_1, \tau_2, \tau_3$ , whereas  $\tau_5$  can only be reached from  $\tau_3$ . If we uniformly select a tree from  $\Gamma$  and apply the link-cut operation, the probabilities of obtaining the new trees are  $p(\tau_4) = \frac{5}{6}$  and  $p(\tau_5) = \frac{1}{6}$ , which are biased.

By applying appropriate reweighting, we can correct for this bias to preserve the expected contributions of each tree:  $\mathbb{E}[\tau_4] = \frac{1}{3} \times \frac{1}{3} + \frac{1}{3} \times \frac{1}{3} + \frac{1}{3} \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$ ,  $\mathbb{E}[\tau_5] = \frac{1}{3} \times \frac{1}{2} \times 2 = \frac{1}{3}$ . This ensures that the estimator remains unbiased, even though the transformation probabilities are not uniform.

THEOREM 4.4.

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

PROOF.

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

□

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  $\Gamma'_e$ . 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*( $\tau_e, e$ ) 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.

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

### Algorithm 3: ISM

**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      $\tau' \leftarrow \text{link-cut}(\tau, (u, v))$ ;
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       $w(\tau) \leftarrow w(\tau) / w_{\text{sum}}(\tau_{\text{selected}}) \cdot R(u, v)$ ;
12    else
13       $w(\tau) \leftarrow w(\tau) / w_{\text{sum}}(\tau_{\text{unselected}}) \cdot (1 - R(u, v))$ ;
14     $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau) \cdot w(\tau)$ ;
15 if Update is Del  $(u, v)$  then
16    $w_{\text{org}} \leftarrow 0$ ;
17   for each tree  $\tau \in T$  that contains edge  $(u, v)$  do
18      $w_{\text{org}} \leftarrow w_{\text{org}} + w(\tau)$ ;
19      $\tau' \leftarrow \text{cut-link}(\tau, (u, v))$ ;
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        $w(\tau) \leftarrow w(\tau) / w_{\text{sum}}(T_{\text{updated}}) \cdot w_{\text{org}}$ ;
25      $\tilde{\kappa}' \leftarrow \tilde{\kappa}' + f(\tau) \cdot w(\tau)$ ;
26 return  $\tilde{\kappa}', S'$ ;

```

PROOF SKETCH. The proof is similar to that of Theorem 4.4. By using the definition of expectation and the properties of the bipartite graph  $\mathcal{B}$ , we can derive the desired result. □

The implementation of improved sample maintenance (ISM) 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.

THEOREM 4.6 (CORRECTNESS AND TIME COMPLEXITY OF ISM). ISM returns an estimate  $\tilde{\kappa}'$  that satisfies the relative-error guarantee. The overall time complexity ISM is

$$O(R(e)\omega \cdot (m + n \min(\Delta, \log n))),$$

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 Theorem 4.1 and Theorem 4.2, respectively. ISM

**Table 2: Detailed statistics of datasets ( $\Delta$  denotes the graph diameter,  $\phi$  is the expected time of sampling a UST, and  $\bar{R}(e)$  represents the average effective resistance of each edge)**

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

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  $\tau_e$  via a link-cut operation, which is equivalent to the number of trees that can be obtained from  $\tau_0$  via a cut-link operation. Consequently,  $d(\tau_e)$  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 ISM 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.  $\square$

**Discussion.** ISM further accelerates the UST sampling process compared to BSM. 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.

## 5 EXPERIMENTS

### 5.1 Experimental Settings

**Datasets.** We employ 10 real-world datasets including various type of graphs, primarily focusing on collaboration networks, social networks and road networks. All datasets can be downloaded from [15, 23, 42]. 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 2. Following previous studies [25, 51], we approximate KC using ApproxKemeny [52] with  $\epsilon = 0.15$  as the ground truth for small graphs ( $n < 10^6$ ). For larger graphs, ApproxKemeny 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 sufficiently accurate result as the ground truth.

**Dynamic Updates.** To simulate dynamic graph updates, we consider both edge insertions and deletions. For the insertion scenario, we first select 90% of the original graph as the base graph and sample 100 edges from the remaining 10% as candidate insertions. For the deletion scenario, we sample 100 edges directly from the original graph as deletions. To better reflect realistic update behavior, we further adopt a power-law update model, where most updates occur on a small fraction of edges. Specifically, an edge  $(u, v)$  is chosen for insertion or deletion with probability proportional to the product of the degrees of  $u$  and  $v$ , ensuring that edges incident to high-degree nodes are more likely to be updated.

**Different Algorithms.** For static graphs, we compare our proposed method TTF with several representative approaches, including the LERW-based method ForestMC [51], the UST-based method SpanTree [29], the truncated random walks-based methods DynamicMC [25] and RefinedMC [51], and the matrix-related method ApproxKemeny [52]. For dynamic graphs, we select the three most competitive static methods, TTF, ForestMC, and SpanTree, and apply them by re-running after each update as baseline methods, comparing with our two proposed sample-maintenance approaches, BSM and ISM.

**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 ApproxKemeny [52], 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 on Static Graphs

**Exp-I: Error vs. Time on Static Graphs.** In this experiment, we evaluate the performance of static algorithms across all datasets by comparing running time against relative error. As shown in Fig. 9, ApproxKemeny can achieve very accurate results, but its runtime and memory usage are prohibitively high, making it infeasible for large graphs. Methods based on truncated random walks perform reasonably well on some datasets but struggle on road networks, where even extended runtimes fail to achieve high precision. ForestMC exhibits good accuracy on social networks, but its performance degrades on other types of graphs. In contrast, TTF consistently achieves lower error in less time across nearly all datasets, demonstrating both high efficiency and accuracy.

**Exp-II: 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. 10 and Fig. 11. We observe that while the choice of path selection method has a negligible impact on running time, it significantly affects the accuracy of the estimation. As discussed in Theorem 3.9, 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 diameter of the graph, while it can be substantially longer when using DFS



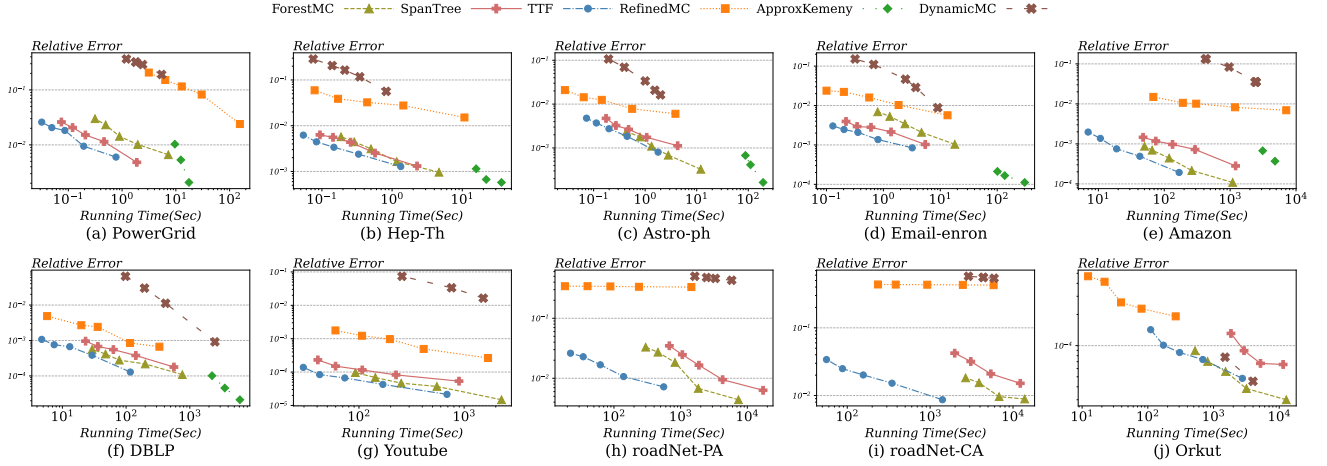


Figure 9: Relative error vs running time for different algorithms

or randomly sampled trees. Consequently, shorter paths tend to reduce the variance of TTF, leading to more accurate estimations.

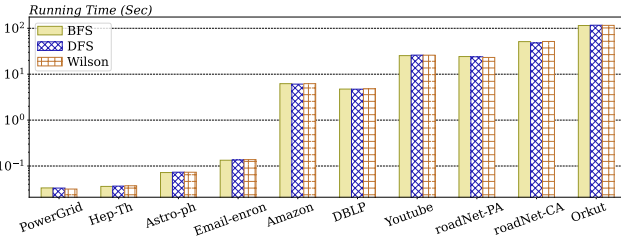


Figure 10: Running time of TTF on each datasets with three different path selection Methods

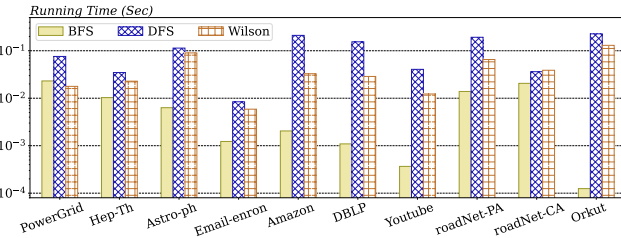


Figure 11: Relative error of TTF on each datasets with three different path selection Methods

**Exp-III: Effect of Root Selection.** There three static methods (TTF, ForestMC and SpanTree) require selecting a root node when applying Wilson’s algorithm, and this choice has a clear impact on the running time. To systematically evaluate this effect, we randomly select root nodes with degrees close to  $\frac{d_{\max}}{5}$ ,  $\frac{2d_{\max}}{5}$ ,  $\dots$ ,  $d_{\max}$ , and measure the average running time. The results, illustrated in Fig. 12, show that in most cases choosing the node with the maximum degree yields the fastest performance. Therefore, we adopt the highest-degree node as the default root in our experiments.

**Exp-IV: Scalability on Synthetically Generated Graphs.** In this experiment, we evaluate the scalability of different methods

on randomly generated graphs. We employ two standard random graph models: (i) Erdős–Rényi graphs [8], where each possible edge is independently included with probability  $p = 5 \times 10^{-4}$ , and (ii) Chung-Lu model [1], where the probability of creating an edge is proportional to the product of the endpoints’ expected degrees, and the degree distribution follows a power law with exponent  $\gamma = 3.5$ . The synthetic graphs are generated using the Networkit toolkit [44], with their number of nodes in the largest connected component varying from  $10^3$  to  $10^6$ . Fig. 13a and Fig. 13b show that TTF and ForestMC scale better, as the running time of other methods increases more rapidly.

**Exp-V: Scalability on Huge Real-World Graphs.** In this experiment, we evaluate the scalability of different methods on two huge real-world datasets: Friendster (65,608,366 nodes and 1,806,067,135 edges) and Twitter (21,297,772 nodes and 265,025,809 edges). We run TTF with sufficient sample size  $T = 1000$  to approximate the ground truth. For all methods, we set the same and smaller sample size, and gradually increase it to compare their relative error and running time, omitting results that exceed 24 hours. Fig. 14 show that TTF and ForestMC remain feasible even on such extremely large graphs, whereas other methods either become prohibitively slow or fail to produce accurate results. Overall, TTF achieves the best trade-off between accuracy and efficiency, demonstrating strong scalability.

**Exp-VI: Parallelization.** We evaluate the parallel performance of different algorithms using 64 threads against their single-threaded performance. All methods except ApproxKemeny naturally parallelize across samples. To ensure fairness, we control the number of samples to make the error levels of all algorithms comparable. As shown in Fig. 15, all algorithms benefit significantly from parallel execution. Although TTF does not achieve the highest speedup, it remains among the most efficient methods.

### 5.3 Experiment Results on Dynamic Graphs

**Exp-VII: Index Size.** This experiment evaluates the index size of both BSM and ISM, compared to the size of graphs. The experimental results are shown in Fig. 16. We observe that the index sizes of both methods grow proportionally with the graph size, and the

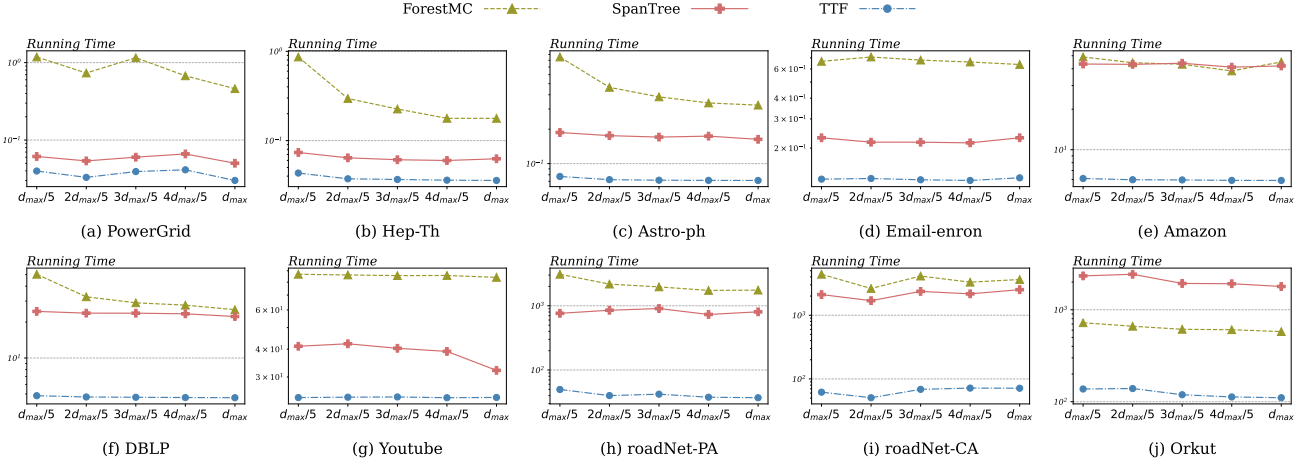
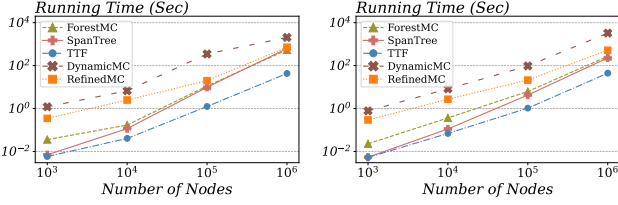


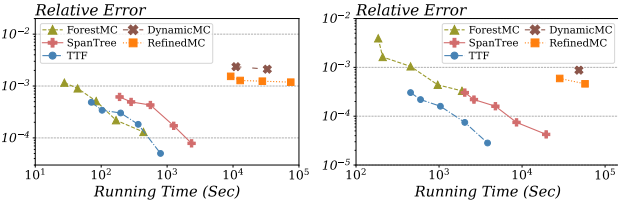
Figure 12: Impact of root node with different degrees on the running time of TTF



(a) Erdős-Rényi model

(b) Chung-Lu model

Figure 13: Scalability performance of different algorithms on random generated graphs



(a) Twitter

(b) Friendster

Figure 14: Relative error vs running time on super-large real-world graphs

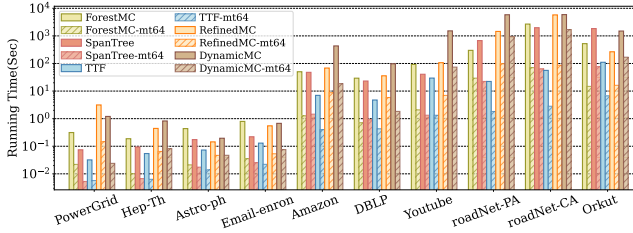


Figure 15: Parallel speedup of different algorithms

growth rate remains stable across all datasets, which empirically validates the  $O(n)$  space complexity discussed in Section 4. In all tested datasets, the maximum index size is 4.9GB, demonstrating that our proposed dynamic algorithms are highly space-efficient while supporting real-time sample maintenance.

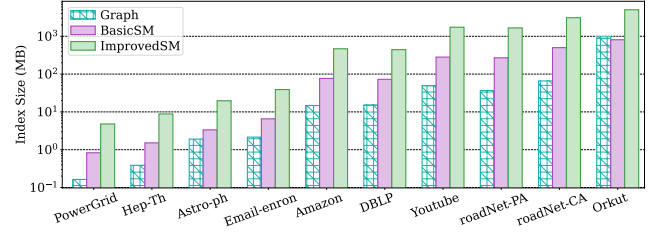


Figure 16: Index size of sample maintenance algorithms

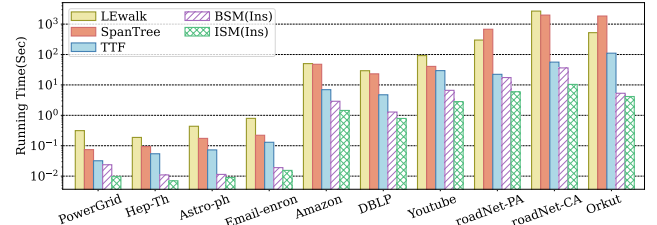


Figure 17: Running time of different algorithms for edges insertions on dynamic graphs

#### Exp-VIII: Performance for Edge Insertion on Dynamic Graphs.

We evaluate the performance of the basic and improved sample maintenance methods, BSM and ISM, 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. 17 and Fig. 18. As expected, both BSM and ISM significantly outperform the static methods in terms of time efficiency. On social networks like Orkut, BSM and ISM are an order of magnitude faster than TTF. However, on road networks such as roadNet-PA and roadNet-CA, the speed-up of BSM 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, ISM still achieves notable speed-ups even on road networks, since link-cut operations are significantly faster than tree sampling, though it produces slightly higher error.



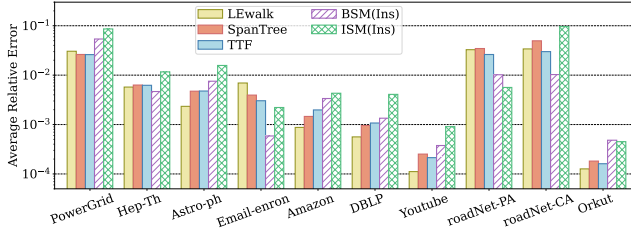


Figure 18: Accuracy performance of different algorithms for edges insertions on dynamic graphs

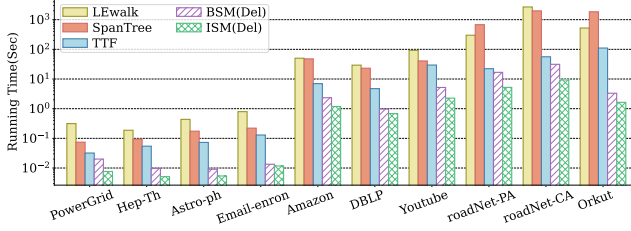


Figure 19: Running time of different algorithms for edges deletions on dynamic graphs

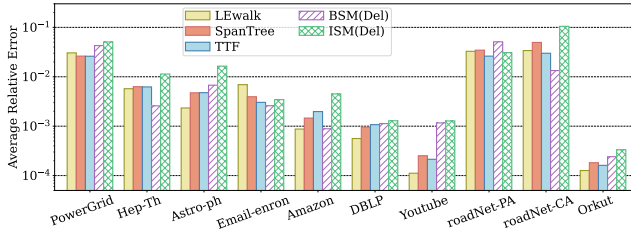


Figure 20: Accuracy performance of different algorithms for edges deletions on dynamic graphs

#### Exp-VIII: Performance for Edge Deletion on Dynamic Graphs.

We also evaluate the performance of BSM and ISM for edge deletions. The results, shown in Fig. 19 and Fig. 20, are generally consistent with those from the insertion scenario. Notably, BSM shows better performance improvements on road networks during deletions than insertions, since it can employ the standard Wilson algorithm for tree sampling. Moreover, both BSM and ISM 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.

### 5.4 Case Studies

**Case Study: Graph Classification.** We conducted a case study to investigate the relationship between the Kemeny constant and different types of graphs. For each dataset, we normalized the Kemeny constant as  $\kappa(G)/n$  and plotted the results. As shown in Fig. 21, different graph types exhibit distinct patterns in their normalized Kemeny constants. Social networks and collaboration networks typically have  $\kappa(G)/n \approx 1$ , whereas road networks generally show larger values. Intuitively, a smaller Kemeny constant indicates that nodes are more tightly connected, which facilitates reaching other nodes from any starting point. This demonstrates that the Kemeny constant is informative for understanding graph connectivity and can be leveraged as a feature for graph classification.

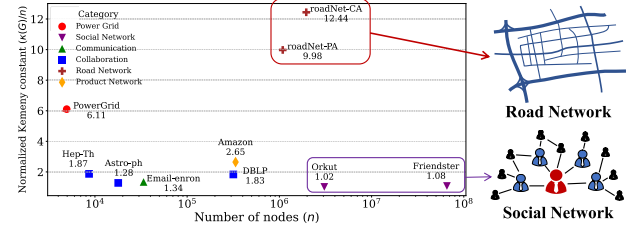


Figure 21: Normalized KC across real-world graphs

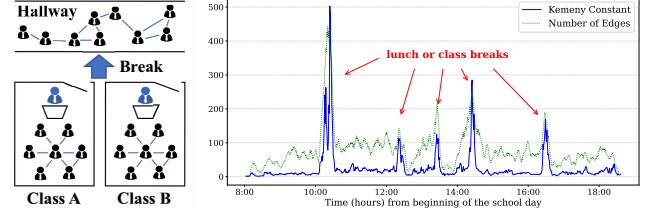


Figure 22: Evolution of Kemeny constant over time

**Case Study: Kemeny Constant in Dynamic Graphs.** We further investigate the evolution of the Kemeny constant in dynamic graphs using the HighSchool dataset [15], which captures contacts among students over a single day. Each student is represented as a node, and an edge is created when two students are in close and face-to-face proximity during a 20-second time window. We divided the data into 5-minute windows and computed the Kemeny constant for each window, updating every minute. Figure 22 shows the number of edges and the corresponding Kemeny constant over time. Although the number of edges fluctuates considerably, the Kemeny constant remains largely stable, with notable changes only during periods such as class breaks or lunch, when the network structure undergoes significant alterations. This demonstrates that the Kemeny constant can effectively capture structural changes in dynamic networks.

## 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, 49, 57], network embedding [16, 40, 60], and complex network analysis [43, 50]. Various random walk-based metrics, personalized PageRank (PPR) [32, 46, 54] and effective resistance (ER) [7, 39, 45, 55] have received a lot of attention. For both PPR and ER, local algorithms such as push are commonly used in recent studies [3, 28, 46], 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.

**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, 53, 59, 61]. Most dynamic algorithms build upon their corresponding static versions. For example, LazyForward [59] and TrackingPPR [36] were both developed as extensions of the push algorithm [3]. Similarly, various index update methods for random walks [6, 19, 35] and power iteration-based dynamic algorithms [26, 58] 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] 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.
- [3] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *FOCS*. IEEE, 475–486.
- [4] 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.
- [5] Luca Avena, Fabienne Castell, Alexandre Gaudillière, and Clothilde Mélot. 2018. Random forests and networks analysis. *Journal of Statistical Physics* 173 (2018), 985–1027.
- [6] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast Incremental and Personalized PageRank. *VLDB* 4, 3 (2010), 173–184.
- [7] Béla Bollobás. 2013. *Modern graph theory*. Vol. 184. Springer Science & Business Media.
- [8] Béla Bollobás and Béla Bollobás. 1998. *Random graphs*. Springer.
- [9] Mo Chen, Jianzhuang Liu, and Xiaou Tang. 2008. Clustering via Random Walk Hitting Time on Directed Graphs. In *AAAI*, Vol. 8. 616–621.
- [10] Fan Chung and Ji Zeng. 2023. Forest formulas of discrete Green’s functions. *J. Graph Theory* 102, 3 (2023), 556–577.
- [11] S Condamine, O Bénichou, V Tejedor, R Voituriez, and Joseph Klafter. 2007. First-passage times in complex scale-invariant media. *Nature* 450, 7166 (2007), 77–80.
- [12] 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.
- [13] Peter G Doyle. 2009. The Kemeny constant of a Markov chain. *arXiv preprint arXiv:0909.2636* (2009).
- [14] Peter M. Fenwick. 1994. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exp.* 24, 3 (1994), 327–336.
- [15] Julie Fournet and Alain Barrat. 2014. Contact Patterns among High School Students. *PLoS ONE* 9, 9 (09 2014), e107878. <https://doi.org/10.1371/journal.pone.0107878>
- [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, Sibow 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] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. <https://networkrepository.com>
- [43] Purnamrita Sarkar and Andrew W Moore. 2011. Random walks in social networks and their applications: a survey. *Social Network Data Analytics* (2011), 43–77.
- [44] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2015. NetworkKit: A Tool Suite for Large-scale Complex Network Analysis. *arXiv:1403.3005 [cs.SI]* <https://arxiv.org/abs/1403.3005>
- [45] Prasad Tetali. 1991. Random walks and the effective resistance of networks. *Journal of Theoretical Probability* 4, 1 (1991), 101–109.
- [46] 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.
- [47] Scott White and Padhraic Smyth. 2003. Algorithms for estimating relative importance in networks. In *KDD*. 266–275.

- [48] David Bruce Wilson. 1996. Generating random spanning trees more quickly than the cover time. In *STOC*. 296–303.
- [49] 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.
- [50] 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.
- [51] Haisong Xia and Zhongzhi Zhang. 2024. Efficient Approximation of Kemeny’s Constant for Large Graphs. *SIGMOD* 2, 3 (2024), 1–26.
- [52] 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.
- [53] Mingji Yang, Hanzhi Wang, Zhewei Wei, Sibao Wang, and Ji-Rong Wen. 2024. Efficient algorithms for personalized pagerank computation: A survey. *IEEE TKDE* (2024).
- [54] 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.
- [55] 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.
- [56] 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.
- [57] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. 2012. Challenging the Long Tail Recommendation. *VLDB* 5, 9 (2012).
- [58] Minji Yoon, Woojeong Jin, and U Kang. 2018. Fast and accurate random walk with restart on dynamic graphs with guarantees. In *WWW*. 409–418.
- [59] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *KDD*. 1315–1324.
- [60] 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.
- [61] Yanping Zheng, Hanzhi Wang, Zhewei Wei, Jiajun Liu, and Sibao Wang. 2022. Instant graph neural networks for dynamic graphs. In *KDD*. 2605–2615.