

# Tree+DAG: An Efficient Index for Resistance Distance Computation on Core-Periphery Graphs

MEIHAO LIAO, Beijing Institute of Technology, China

BAO XING, Beijing Institute of Technology, China

RONG-HUA LI, Beijing Institute of Technology, China

GUOREN WANG, Beijing Institute of Technology, China

Resistance distance computation is a fundamental problem in graph data management. To avoid the high latency of online evaluation, recent work builds offline indexes. Existing approaches, however, fall into two families with complementary weaknesses: loop-erased random-walk methods (e.g., LEIndex) are efficient on fast-mixing graphs but suffer high variance on sparse or tree-like topologies, whereas Cholesky-based methods (e.g., TreeIndex) perform well on small-treewidth graphs but are costly to construct for large, dense networks. In this work, we revisit the probabilistic structure of loop-erased random walks and reveal a previously unexplored connection between their distribution and the Cholesky decomposition of the inverse Laplacian. Building on this insight, we present Tree+DAG, an index that unifies deterministic matrix decomposition with loop-erased random walk sampling. We first apply a core-tree decomposition to split the graph into a tree-like periphery and a dense core. The periphery is isolated via TREEINDEX—a compact hierarchical tree labelling that exploits the nonzero pattern of the Cholesky factors of the inverse Laplacian. In the core, random walks mix rapidly and thus erase few cycles; we propose COREINDEX, which stores the DAG encoding the erased cycle structure over the core. This design yields near-linear index size and build time, and low query latency in practice on graphs with a core-periphery structure. Extensive experiments on 10 real-world datasets—including road, social, and web graphs with up to hundreds of millions of edges—show that Tree+DAG delivers at least an order-of-magnitude faster query performance than LEIndex. For example, on Twitter with 21, 297, 772 nodes and 265, 025, 545 edges, Tree+DAG attains absolute error  $10^{-2}$  with a modest index (3.1 GB,  $1.5\times$  the graph) built in 48 minutes, delivering over 3 orders of magnitude faster queries— $8 \times 10^{-5}$  seconds vs.  $7 \times 10^{-1}$  seconds for LEIndex.

CCS Concepts: • **Networks** → **Network algorithms**; • **Mathematics of computing** → **Probabilistic algorithms**.

Additional Key Words and Phrases: resistance distance, graph indexing, core-periphery graphs

## ACM Reference Format:

Meihao Liao, Bao Xing, Rong-Hua Li, and Guoren Wang. 2026. Tree+DAG: An Efficient Index for Resistance Distance Computation on Core-Periphery Graphs. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 291 (September 2026), 27 pages. <https://doi.org/10.1145/3837129>

## 1 Introduction

Resistance distance computation is a fundamental task in graph data management, serving as both a structural distance and a node-similarity measure. Effective resistance has been directly used for node-similarity queries [35, 61], graph clustering [2], robust routing [48], and resistance-aware GNN methods [5, 6]. It is also closely related to broader studies of structural bottlenecks and

---

Authors' Contact Information: Meihao Liao, [mhliao@bit.edu.cn](mailto:mhliao@bit.edu.cn), Beijing Institute of Technology, Beijing, China; Bao Xing, [3120245641@bit.edu.cn](mailto:3120245641@bit.edu.cn), Beijing Institute of Technology, Beijing, China; Rong-Hua Li, [lironghuabit@126.com](mailto:lironghuabit@126.com), Beijing Institute of Technology, Beijing, China; Guoren Wang, [wanggrbit@126.com](mailto:wanggrbit@126.com), Beijing Institute of Technology, Beijing, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART291

<https://doi.org/10.1145/3837129>

oversquashing [40, 51, 63]. Despite its importance, scaling resistance distance to large graphs remains challenging because exact evaluation typically requires solving large Laplacian linear systems.

This paper studies the *single-pair resistance-distance query*, a standard workload in prior resistance-distance computation studies [15, 35, 39, 45, 61]. Given a large graph and a query pair  $(s, t)$ , the goal is to return their resistance distance, denoted by  $r(s, t)$ , with low online latency after preprocessing. This setting arises when applications repeatedly evaluate selected pairs rather than materialize all-pairs distances, e.g., origin–destination robustness in transportation networks [48] or trust/suspicion scores between social accounts and trusted users [9, 47]. Such repeated point-to-point workloads motivate an index that amortizes preprocessing over many queries.

Scalable methods for this workload can be understood by the graph structure they exploit. Although near-linear Laplacian solvers [13, 20] provide a principled route to high-accuracy answers, they remain too costly for repeated online queries on large graphs. *Random walk-based* methods approximate  $r(s, t)$  by sampling random walks [15, 35, 45, 61]; they typically perform well on fast-mixing graphs but, *as a class*, deteriorate on slow-mixing topologies—requiring long trajectories and incurring high variance [46]. Among index-based solutions, LEIndex [33, 39] compresses information by using loop-erased random walks to sample spanning forests rooted at a small landmark set  $\mathcal{V}_l$  and storing projections onto  $\mathcal{V}_l$ , yet it inherits the slow-mixing limitations and further introduces approximation error from the landmark-based Schur complement. In contrast, *Cholesky-based* methods (e.g., TreeIndex [37]) exploit the sparsity pattern of the Cholesky factorization of the Laplacian inverse and are exact on low-treewidth graphs such as road networks; however, their preprocessing cost and label size scale with the tree-height parameter  $h_G$  [37], which is large in dense, expander-like graphs.

The tension above is especially visible in core–periphery graphs, a common structure in real-world networks [32, 42]: the core is dense and expander-like, so random walks mix rapidly, while the periphery is sparse and close to a tree, so direct elimination is efficient. This heterogeneity makes neither random walk-based nor Cholesky-based approaches universally efficient. To address this, we revisit loop-erased random walks (LERW) through the lens of Wilson stacks: the erased cycles induce a directed acyclic graph (DAG) that is *order-invariant*—cycles can be sampled and traversed in any order while yielding the same DAG [33]. We then introduce a *node-based* LERW ordering and show that its  $i$ -th step matches a rank-one term in the Cholesky decomposition of the inverse Laplacian, giving a layer-wise identity that expresses entries of the Laplacian inverse  $L_v^{-1}$ —and hence resistance distance—as expectations of cycle counts per layer.

Building on these insights, we present Tree+DAG, a hybrid index that unifies deterministic Cholesky-based computation with LERW sampling. We first apply a core–tree decomposition [42] and split  $\mathcal{V} = \mathcal{V}_T \cup \mathcal{V}_C$ . For index construction, on the tree-like periphery  $\mathcal{V}_T$  we run TREEINDEX up to the boundary, storing the nonzeros of the Cholesky columns localized to the hierarchical forest structure [37]. On the dense core  $\mathcal{V}_C$ , we build COREINDEX: we draw LERW samples once, organize all erased cycles into a cycle DAG, assign each cycle to a *layer* via the *node-based* ordering, and keep only cycles whose layers lie in  $\mathcal{V}_C$ . For query processing, a query  $r(s, t)$  performs no random walks: it (i) accumulates the tree contributions along the paths from  $s$  and  $t$  to their roots in TREEINDEX, and (ii) reads from COREINDEX only the pre-stored cycles incident to  $s$  or  $t$  and aggregates their layer-wise contributions. Empirically, the number of such core cycles per node is very small on large graphs (0.040 on Twitter), yielding low endpoint-local query latency, while the index builds in near-linear time and space in practice.

Extensive experiments on 10 large real-world datasets—including road, social, and web graphs with up to tens of millions of nodes and hundreds of millions of edges—show that Tree+DAG consistently outperforms state-of-the-art (SOTA) methods. At matched accuracy, it answers queries

at least an order of magnitude faster than LEIndex. Its index-construction time and index size are also reduced compared to previous settings.

In summary, this paper makes two main technical advances:

**New theoretical insights.** We establish a layer-wise expectation identity connecting loop-erased random walks with the Cholesky decomposition of the Laplacian inverse. Exploiting the order-invariance of Wilson stacks, we define a *node-based* LERW ordering in which the  $i$ -th traversal layer corresponds to a *rank-one* term in the Cholesky factorization. This yields closed-form, layer-wise identities that connect expected cycle counts to entries of the corresponding Cholesky factors,  $L_o^{-1}$ , and resistance distance.

**Novel index structure.** We introduce Tree+DAG, a hybrid index tailored to core-periphery graphs. Using a core-tree decomposition, vertices are partitioned into a tree-like periphery  $\mathcal{V}_T$  and a dense core  $\mathcal{V}_C$ . On  $\mathcal{V}_T$  we *deterministically* build a distance labelling via TREEINDEX, storing only Cholesky nonzeros. In the core, guided by our layer-wise theory, we run LERW once, assign erased cycles to layers, and retain only cycles whose layers lie in  $\mathcal{V}_C$ , forming a compact cycle-based COREINDEX. Queries perform *no sampling*: they read periphery labels along the two rootward paths and aggregate pre-stored layer contributions from core cycles incident to the endpoints. This design yields near-linear index size and construction time and low query latency in practice.

We also report an empirical evaluation on 10 real-world graphs with up to tens of millions of nodes and hundreds of millions of edges. The results show that Tree+DAG consistently outperforms SOTA baselines in both speed and accuracy; for example, on Twitter (21,297,772 nodes; 265,025,545 edges), it attains absolute error  $10^{-2}$  using a 3.1 GB index ( $1.5\times$  the graph) built in 48 minutes, answering queries in  $8 \times 10^{-5}$  seconds versus  $7 \times 10^{-1}$  seconds for LEIndex. For reproducibility, we provide the source code in our experiments at <https://github.com/mhliao0516/CoreTree>.

## 2 Preliminaries

Given an undirected graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  with  $n$  nodes and  $m$  edges. Let  $\mathbf{A}$  denote the adjacency matrix,  $\mathbf{D}$  denote the degree matrix, the Laplacian matrix  $\mathbf{L}$  is defined as  $\mathbf{L} = \mathbf{D} - \mathbf{A}$ . Throughout the paper, we assume  $\mathcal{G}$  is connected. Resistance distance has an electrical-network definition: view every edge as a unit resistor, inject one unit of current at  $s$ , and extract one unit at  $t$ . If  $\phi$  is the resulting voltage vector, defined up to an additive constant, then

$$\mathbf{L}\phi = \mathbf{e}_s - \mathbf{e}_t, \quad r(s, t) = \phi_s - \phi_t. \quad (1)$$

It is well known that resistance distance is a metric [50]: it satisfies non-negativity and identity of indiscernibles, symmetry, and the triangle inequality. Resistance distance is also equivalent to a random-walk quantity: if  $\kappa(s, t)$  denotes the commute time, i.e., the expected number of steps for a random walk to travel from  $s$  to  $t$  and then return from  $t$  to  $s$ , then [50]

$$\kappa(s, t) = 2m \cdot r(s, t). \quad (2)$$

Suppose that the eigen-decomposition of  $\mathbf{L}$  is  $\mathbf{L} = \sum_{i=1}^n \lambda_i \mathbf{u}_i \mathbf{u}_i^T$ , where  $0 = \lambda_1 \leq \dots \leq \lambda_n$  are the eigenvalues of  $\mathbf{L}$  and  $\mathbf{u}_i$  is the corresponding eigenvector of  $\lambda_i$  for  $i = 1, \dots, n$ . We can define the Moore-Penrose pseudo-inverse of  $\mathbf{L}$  as  $\mathbf{L}^\dagger = \sum_{i=2}^n \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^T$ . The resistance distance then is defined as:

$$r(s, t) = (\mathbf{e}_s - \mathbf{e}_t)^T \mathbf{L}^\dagger (\mathbf{e}_s - \mathbf{e}_t), \quad (3)$$

where  $\mathbf{e}_s$  ( $\mathbf{e}_t$ ) is the  $s$ -th ( $t$ -th) standard basis vector. Computing the Laplacian pseudoinverse  $\mathbf{L}^\dagger$  is prohibitively expensive on large graphs; in the worst case it requires  $O(n^3)$  time. Consequently, many algorithms present alternative formulas to express resistance distance and avoid direct computation of  $\mathbf{L}^\dagger$ . In particular, random walk-based approaches [15, 45, 61] typically rely on the

following formula:

$$r(s, t) = \sum_{i=0}^{\infty} \left( \frac{p_i(s, s)}{d_s} + \frac{p_i(t, t)}{d_t} - \frac{p_i(s, t)}{d_t} - \frac{p_i(t, s)}{d_s} \right), \quad (4)$$

where  $p_i(x, y)$  denotes the probability that a random walk starting from  $x$  is at  $y$  after  $i$  steps. For a connected non-bipartite graph, the infinite series converges; in practice, random-walk-based methods truncate the series at a sufficiently large step  $K$ . Also, Landmark [35] shows that the resistance distance can be represented using  $\mathbf{L}_v^{-1}$ , the inverse of the submatrix of  $\mathbf{L}$  obtained by removing the rows and columns corresponding to a landmark node  $v$ . That is,

$$r(u, v) = \mathbf{e}_u^T \mathbf{L}_v^{-1} \mathbf{e}_u. \quad (5)$$

$$r(s, t) = (\mathbf{e}_s - \mathbf{e}_t)^T \mathbf{L}_v^{-1} (\mathbf{e}_s - \mathbf{e}_t), \quad s, t \neq v. \quad (6)$$

Here, resistance distance is connected to random walks via the fact that the  $s, t$ -th entry of  $\mathbf{L}_v^{-1}$  is the expected number of visits to  $t$  by a random walk starting at  $s$  that terminates upon hitting  $v$ , divided by the degree of  $t$ . This formula further extends to multiple landmarks. Suppose  $\mathcal{U}_i$  and  $\mathcal{V}_i$  form a partition of  $\mathcal{V}$ . The Schur complement of  $\mathbf{L}$  with respect to  $\mathcal{V}_i$  is defined as:

$$\mathbf{L}/\mathcal{V}_i = \mathbf{L}_{\mathcal{V}_i \mathcal{V}_i} - \mathbf{L}_{\mathcal{V}_i \mathcal{U}_i} \mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1} \mathbf{L}_{\mathcal{U}_i \mathcal{V}_i}, \quad (7)$$

where  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}$  represents the submatrix of  $\mathbf{L}$  indexed by rows and columns corresponding to  $\mathcal{U}_i$ . The  $s, t$ -th entry of  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$  is the expected number of visits to  $t$  by a random walk starting at  $s$  that terminates upon hitting  $\mathcal{V}_i$ , divided by the degree of  $t$ . Using the Schur complement, we can express the resistance distance in terms of  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$ —in other words, in terms of random walks from  $\mathcal{U}_i$  that are absorbed at  $\mathcal{V}_i$ .

**LEMMA 2.1 (RESISTANCE DISTANCE IN TERMS OF  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$  [39]).** *Let  $\mathbf{p}_u$  be the  $u$ -th row of the matrix  $\mathbf{P}_r = \mathbf{P}_f = -\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1} \mathbf{L}_{\mathcal{U}_i \mathcal{V}_i}$  for  $u \in \mathcal{U}_i$ , where  $\mathbf{p}_u(v)$  is the probability that a random walk from  $u$  hits  $v \in \mathcal{V}_i$  and the probability that in a random spanning forest with root set  $\mathcal{V}_i$ ,  $u$  is rooted at  $v$ . Then,*

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

$$r(u_1, u_2) = (\mathbf{e}_{u_1} - \mathbf{e}_{u_2})^T (\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}) (\mathbf{e}_{u_1} - \mathbf{e}_{u_2}) + (\mathbf{p}_{u_1} - \mathbf{p}_{u_2})^T (\mathbf{L}/\mathcal{V}_i)^\dagger (\mathbf{p}_{u_1} - \mathbf{p}_{u_2}); \quad (8)$$

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

$$r(u, v) = \mathbf{e}_u^T \mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1} \mathbf{e}_u + (\mathbf{p}_u - \mathbf{e}_v)^T (\mathbf{L}/\mathcal{V}_i)^\dagger (\mathbf{p}_u - \mathbf{e}_v); \quad (9)$$

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

$$r(v_1, v_2) = (\mathbf{e}_{v_1} - \mathbf{e}_{v_2})^T (\mathbf{L}/\mathcal{V}_i)^\dagger (\mathbf{e}_{v_1} - \mathbf{e}_{v_2}). \quad (10)$$

Given this identity,  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$  is accessible through  $\mathcal{V}_i$ -absorbed random walks starting in  $\mathcal{U}_i$ ; the bottleneck is the Schur complement term  $(\mathbf{L}/\mathcal{V}_i)^\dagger$ , which does not scale to large graphs. To address this issue, TreeIndex [37] gives a new way to compute the resistance distance by using the *Cholesky decomposition* of the Laplacian inverse. Given a node ordering  $\mathcal{O} = (v_1, v_2, \dots, v_n)$ , suppose that  $\mathcal{U}_i$  is the set of nodes before  $v_i$  in  $\mathcal{O}$ ,  $\mathcal{S}[:, v_i]$  is the  $v_i$ -th column of  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$ , they show that  $\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1}$  can be decomposed as:

$$\mathbf{L}_{\mathcal{U}_i \mathcal{U}_i}^{-1} = \sum_{i=1}^{|\mathcal{U}_i|} \frac{\mathcal{S}[:, v_i] \mathcal{S}[:, v_i]^T}{\mathcal{S}[v_i, v_i]}. \quad (11)$$

According to Eq.(11), we can construct  $L_{u_i u_i}^{-1}$  incrementally for  $L_{u_1 u_1}^{-1}, L_{u_2 u_2}^{-1}, \dots, L_v^{-1}$  via successive rank-one outer-product updates. The resistance distance  $r(s, t)$  then follows from Eq. (5)-(6).

Core-periphery structure is common in large graphs [32, 42, 43]. Following this line of work, we use the term to describe graphs that contain a dense, well-connected core and a sparse periphery that is close to a tree. In this paper, this structure is exposed by the core-tree decomposition used by Tree+DAG: it peels off a tree-like periphery  $\mathcal{V}_T$  that can be handled efficiently by Cholesky-based labels, and leaves a residual core  $\mathcal{V}_C$  where random walks mix more rapidly and loop-erased walks retain only a small number of cycles per vertex in practice.

**DEFINITION 1 (SINGLE-PAIR RESISTANCE-DISTANCE QUERY).** *Given a preprocessed core-periphery graph  $\mathcal{G}$  and an online query pair  $(s, t)$ , return the resistance distance  $r(s, t)$ , or an accurate approximation  $\hat{r}(s, t)$ , with low online latency.*

This query model captures applications that repeatedly evaluate selected pairs rather than all pairs, such as origin-destination robustness in transportation networks [48] and trust-region scoring in social networks [9, 47].

## 2.1 Existing solutions and their defects

A direct way to answer a single-pair query is to use a Laplacian solver with right-hand side  $\mathbf{e}_s - \mathbf{e}_t$ . Although such solvers are powerful for Laplacian linear systems, each online pair still requires solving a large system over the whole graph, so the cost is paid repeatedly when many pairs are queried. This motivates specialized scalable methods that avoid solving such a system for every queried pair. Existing approaches mainly fall into two categories: *random walk-based algorithms* (e.g., LEIndex) and *Cholesky decomposition-based algorithms* (e.g., TreeIndex), with each excelling on different graph structures.

*Random walk-based algorithms* work best on graphs with rapid mixing and large spectral gap. Since resistance distance is closely tied to random walks, many methods approximate it via random-walk sampling. A common drawback, however, is that the necessary walk length can be large on certain graph structures. Based on Eq. (4), GEER [61] and BiSPER [15] sample  $k$ -step random walks, but  $k$  must often be large to achieve acceptable accuracy. Landmark [35] instead samples  $v$ -absorbed random walks following Eq. (5) and Eq. (6); since  $r(s, t)$  is invariant to the choice of  $v$ , it selects an easy-to-hit node to reduce complexity. In practice, such a node may be hard to identify or ineffective, so walk lengths can still be large. To further accelerate queries, the state-of-the-art LEIndex [39] builds an index using loop-erased walks (according to Lemma 2.1). Notice that an index would require sampling and storing  $\mathcal{V}_l$ -absorbed walks from every node in  $\mathcal{U}_i$ , which is infeasible on large graphs; LEIndex leverages loop-erased walks to compactly summarize this information. It then approximates the Schur complement by the uniform random spanning forests rooted at a landmark node set obtained from the loop-erased walks. At query time, only  $L_{u u}^{-1}$  needs to be computed, corresponding to walks that terminate at the landmark set rather than a single landmark. Because the Schur complement is estimated from samples, taking its (exact) pseudoinverse introduces additional approximation error. To control this, LEIndex restricts the landmark set to be very small (e.g., the 10 highest-degree nodes). Even after projecting onto this landmark set, the required lengths of the corresponding  $\mathcal{V}_l$ -absorbed walks can remain large, limiting query performance.

*Cholesky decomposition-based algorithms* are suitable for graphs that are easily separable and have low treewidth. TreeIndex [37] is a recent index-based method tailored to such graphs: it uses a tree decomposition to construct a distance labelling. Exploiting the fact that these graphs can be partitioned into a hierarchical tree structure by removing small vertex cut sets, TreeIndex leverages the nonzero pattern of the Cholesky decomposition of the inverse Laplacian (Eq. (11)). In particular,

when a small separator disconnects the graph into  $\mathcal{U}_1$  and  $\mathcal{U}_2$ , the blocks  $L_{\mathcal{U}_1\mathcal{U}_1}^{-1}$  and  $L_{\mathcal{U}_2\mathcal{U}_2}^{-1}$  can be constructed independently. Based on Eq. (11),  $S[v_i, :]$  becomes sparse if the vertex ordering follows a bottom-up order induced by the tree decomposition. The nonzero entries of  $S[v_i, :]$  are stored as labels for all  $v_i \in \mathcal{V}$  except the root node  $v$ . The resistance distance  $r(s, t)$  then depends only on the labels along the paths from  $s$  and  $t$  to the root node  $v$  in the tree decomposition. The authors show that the total label size is bounded by  $O(n \cdot h_{\mathcal{G}})$  and can be constructed in  $O(n \cdot h_{\mathcal{G}}^2 \cdot d_{\max})$  time, while each query is answered in  $O(h_{\mathcal{G}})$ , where  $h_{\mathcal{G}}$  is the height of the tree. Unlike random walk-based algorithms, TreeIndex is an exact, deterministic method. However, it is practical primarily for low-treewidth graphs. On large graphs with relatively large  $h_{\mathcal{G}}$ , both the index construction time and the label size become prohibitive, limiting its applicability.

Taken together, the two families are complementary: walk-based methods thrive on fast-mixing, expander-like regions, whereas Cholesky-based indices excel on low-treewidth, easily separable parts. Real networks, however, rarely fit a single regime—they combine dense cores with sparse, tree-like peripheries. We next detail the challenges of unifying these paradigms into a single index.

## 2.2 Challenges of combination

The core-periphery structure explains why the two algorithmic families should be combined rather than chosen independently. In the periphery, the tree-like decomposition makes Cholesky-based labels compact and exact. In the core, stronger connectivity supports faster random-walk mixing: by Cheeger's inequality [4], if  $\lambda_2$  is the second-smallest eigenvalue of the normalized Laplacian and  $\phi(\mathcal{G})$  is the conductance, then  $\frac{\phi(\mathcal{G})^2}{2} \leq \lambda_2 \leq 2\phi(\mathcal{G})$ ; equivalently, for the random-walk transition matrix  $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ , the spectral gap is  $1 - \mu_2 = \lambda_2$ , where  $\mu_2$  is the second-largest eigenvalue of  $\mathbf{P}$ . Thus, larger conductance implies faster mixing, while small conductance indicates bottlenecks and long random-walk trajectories. The core-periphery setting therefore exposes the complementarity between Cholesky-based indexing and loop-erased random-walk sampling, but combining them raises two nontrivial challenges.

A natural idea is to deploy the two methods on different parts of the graph. However, such a combination poses two significant challenges. First, as discussed above, index-based approaches cannot store all per-node random-walk samples, and loop-erased random walks (LERW) are therefore a natural compressed alternative. Yet existing LERW analysis mainly relates the induced spanning forest to resistance distance and does not characterize the finer random-walk events needed to match individual Cholesky terms. Second, although TreeIndex achieves very fast query time on tree-like graphs, there is no comparably efficient offline index for the random-walk part; existing walk-based approaches still rely on online sampling or store insufficient summaries, which is impractical for repeated queries on large graphs.

We address these challenges by developing new theoretical links in Section 3. We show that the two families are connected by a finer, node-wise correspondence: leveraging the order-invariance of Wilson stacks, a *node-based* LERW ordering makes the  $i$ -th traversal layer coincide with a *rank-one* term in the Cholesky decomposition of the Laplacian inverse. This yields a layer-wise identity tying expected cycle counts to entries of the Cholesky column (Lemma 3.1) and, after summing over layers, closed-form expressions for  $L_{\sigma}^{-1}$  and  $r(s, t)$ . Building on these insights, Section 4 presents Tree+DAG, a hybrid framework that aligns each technique with the part of the graph where it is most effective: a core-tree decomposition partitions  $\mathcal{V}$  into a tree-like periphery handled by TreeIndex and a dense core summarized by a compact cycle DAG, obtained by assigning erased cycles to layers and retaining only those in  $\mathcal{V}_C$ . Queries then combine path-local periphery contributions with layer-wise core contributions from cycles incident to the endpoints, yielding efficient resistance-distance queries.

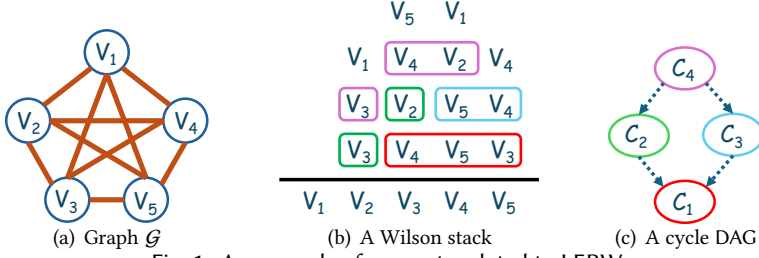


Fig. 1. An example of concepts related to LERW

### 3 New Theoretical Insights

In this section, we provide the theoretical foundation that connects loop-erased random walks with the Cholesky decomposition of the inverse Laplacian matrix. Recall that TreeIndex incrementally computes  $L_{\mathcal{U}_i}^{-1} \mathbf{u}_i = \sum_{i=1}^{|\mathcal{U}_i|} \frac{S[:,v_i]S[:,v_i]^T}{S[v_i,v_i]}$  for the node set  $\mathcal{U}_i$ , we first introduce the concept of loop-erased walk, the Wilson stack and the resulting cycle DAG in Section 3.1. We show that the cycles can be sampled in any ordering while yielding the same Wilson stack. From this viewpoint, a loop-erased walk is simply an ordering of loop erasures guided by an underlying random walk. Next, we introduce a *node-based* loop-erasure order (Section 3.2) in which, at the  $v_i$ -th step, the number of cycles encountered yields an unbiased layer-wise estimator of  $d_u S[u, v_i]^2 / S[v_i, v_i]$ . Equivalently, we decompose the LERW trajectory (the Wilson stack) into a sequence of partial trajectories. This decomposition allows us to first use TreeIndex to construct  $L_{\mathcal{U}_i}^{-1} \mathbf{u}_i$ , and then use LERW to estimate the remaining contribution  $\sum_{i=|\mathcal{U}_{i+1}|}^{n-1} \frac{S[:,v_i]S[:,v_i]^T}{S[v_i,v_i]}$  term, thereby obtaining  $L_v^{-1}$ .

#### 3.1 Loop-erased random walk, Wilson stack and cycle DAG

Loop-erased random walk was introduced by Wilson [57] as a method for sampling uniform spanning trees. Fix a root  $v$  and a node ordering  $\mathcal{O} = (v_1, \dots, v_n = v)$ , it maintains a set  $\mathcal{F}$  initialized as  $\{v\}$ . Following the node ordering  $\mathcal{O}$ , for each node  $u$  that is not in  $\mathcal{F}$ , it starts a (simple) random walk from  $u$  until it first hits  $\mathcal{F}$ . Then, it applies chronological loop erasure to the trajectory to obtain a simple path  $\gamma$  from  $u$  to  $\mathcal{F}$ , and add the vertices and edges of  $\gamma$  to  $\mathcal{F}$ . The process continues until all vertices have been processed. Because all loops are erased, when all vertices have been processed,  $\mathcal{F}$  forms a spanning tree.

**EXAMPLE 1.** Given the graph illustrated in Fig. 1(a), Fig. 2(a) shows an example of a LERW with root  $v_1$ . The random walk starts from  $v_2$ , passing through the cycles  $(v_3, v_4, v_5)$ ,  $(v_2, v_3)$ ,  $(v_4, v_5)$  and  $(v_2, v_3, v_4)$ , finally hits  $v_1$ . Then cycles are erased and the path  $(v_2, v_1)$  is added to  $\mathcal{F}$ . The process continues until forming a spanning tree.

The trajectory of a LERW can be recorded using a Wilson stack. We emphasize that “stack” here is not the LIFO data structure. The term follows Wilson’s “stack of cards” metaphor [57], where each vertex carries a single randomly chosen outgoing neighbor—conceptually like flipping a card—and updates occur by resampling that card when loops are erased.

**DEFINITION 2 (WILSON STACK).** Given a loop-erased walk trajectory with root  $v$ , the Wilson stack maintains for all  $u \neq v$  a stack  $(N_1(u), N_2(u), \dots)$ , where  $N_k(u)$  is the neighbor sampled on the  $k$ -th visit to  $u$ .

**EXAMPLE 2.** Fig. 1(b) illustrates a Wilson stack. For example, the stack of  $v_2$  is  $(v_3, v_3, v_1)$ , indicating  $v_2$  jumps to  $v_3$  on the first visit.

An important structural observation is that the erased cycles produced by LERW form a directed acyclic graph (DAG).

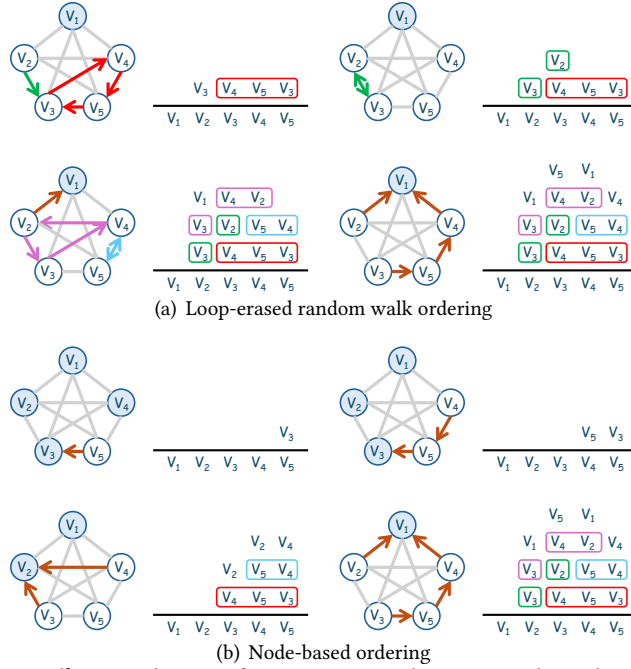


Fig. 2. Different orderings of constructing and traversing the Wilson stack

**DEFINITION 3 (CYCLE DAG).** Given a loop-erased walk trajectory with root  $v$ , a cycle DAG is a directed acyclic graph, where each node  $C_u$  is a cycle in the Wilson stack, an edge  $(C_u, C_v)$  indicates that  $C_u$  must be visited after  $C_v$  has been visited.

**EXAMPLE 3.** Fig. 1(c) illustrates a cycle DAG corresponding to the Wilson stack in Fig. 1(b). There are 4 cycles. For example, the edge  $(C_4, C_2)$  indicates that  $C_4$  must be visited after  $C_2$  has been visited.

As shown by Wilson [57], the ordering of erasing loops is irrelevant. In our example, it is not important whether cycle  $C_2$  is erased before  $C_4$  or after. The results are twofold. First, the Wilson stack can be built in an order-invariant way: initialize every node with an empty stack and repeatedly let each node add a randomly sampled neighbor; among the top entries, if there is a cycle, select any such cycle and—for every node on it—reveal one additional random neighbor by adding it onto its stack; continue until there is no cycle on the top layer. This process yields the same final stack regardless of the order in which cycles are erased. Second, once a Wilson stack has been fixed, it can be traversed in any order with the same law as sampling from scratch: starting from the first layer of the stacks, at each step choose any cycle, mark the entries of that cycle visited. The traversal process repeats until all nodes have been visited.

Leveraging this property, the Wilson stack compactly encodes per-node random-walk information in far less space. Starting from any node, traversing the stack produces a sample sharing the same distribution as a random walk absorbed at  $v$ . The expected running time of an LERW with root  $v$  is  $\text{Tr}(\mathbf{L}_v^{-1}\mathbf{D})$ , whereas the total expected cost of running all per-node random walks is  $\mathbf{1}^T \mathbf{L}_v^{-1} \mathbf{D} \mathbf{1}$ , which can be thousands of times larger on large real-world graphs.

### 3.2 LERW interpretation of resistance distance

Motivated by the fact that the Wilson stack can be traversed in any arbitrary order while producing samples with the same distribution, we introduce a new traversal order. Fix a node ordering  $O = (v_1, \dots, v_n = v)$  and define  $\mathcal{U}_i = \{v_1, \dots, v_i\}$ , the ordering is *node-based*. Intuitively, at step  $v_i$

we restrict the traversal to  $\mathcal{U}_i$ . Consequently, the cycles completed before step  $i$  encode  $\mathbf{L}_{\mathcal{U}_{i-1}\mathcal{U}_{i-1}}^{-1}$ , while those completed after step  $i$  encode  $\mathbf{L}_{\mathcal{U}_i\mathcal{U}_i}^{-1}$ . According to Eq. (11), the cycles encountered exactly at step  $i$  capture the difference between these two matrices, which coincides with  $\mathcal{S}[:, v_i]$ .

**DEFINITION 4 (NODE-BASED ORDERING).** *Given a Wilson stack, the node-based ordering of traversing the stack is defined upon a node ordering  $\mathcal{O} = (v_1, \dots, v_n = v)$  as follows: at step  $v_i$ , we restrict the traversal to  $\mathcal{U}_i = \{v_1, \dots, v_i\}$ . For each cycle appearing at the top layer of the stacks of the nodes in  $\mathcal{U}_i$ , we select any such cycle and mark the stack elements visited. The step  $v_i$  is completed when there is no cycle on the top layer of the stacks of the nodes in  $\mathcal{U}_i$ .*

In the view of cycle DAG, the traversal goes from bottom to top. At step  $v_i$ , we start at the bottom and visit the cycles whose nodes lie entirely in  $\mathcal{U}_i$ . We continue until the next cycle in topological order uses a node outside  $\mathcal{U}_i$ . This does not mean that all cycles contained in  $\mathcal{U}_i$  have been visited. Some may still appear higher in DAG, but they are not yet visible because their predecessors involve nodes in  $\mathcal{V} \setminus \mathcal{U}_i$ .

**EXAMPLE 4.** Fig. 2(b) illustrates an example of traversing the Wilson stack in Fig. 1(b) following node-based ordering defined upon the node ordering  $\mathcal{O} = (v_5, v_4, v_3, v_2, v_1)$ . At the steps corresponding to  $v_5, v_4$ , there is no cycle encountered. At the step corresponding to  $v_3$ , the cycle  $C_1$  and  $C_3$  are encountered since they involve only nodes in  $\mathcal{U}_3$ . At the step corresponding to  $v_2$ , the cycle  $C_2$  and  $C_4$  are encountered.

Under this ordering, the LERW trajectory (the Wilson stack) decomposes into a sequence of partial trajectories. At step  $i$  (corresponding to node  $v_i$ ), several cycles may be encountered. The following lemma characterizes the distribution of the cycles encountered at step  $i$ . Due to the space limit, all proofs are provided in the full version [38].

**LEMMA 3.1 (LOOP-ERASED RANDOM WALK INTERPRETATION OF THE CHOLESKY DECOMPOSITION).** *Let  $C^{v_i}[u]$  denote the number of cycles that pass through node  $u$  at the  $i$ -th step when traversing the Wilson stack under the node-based ordering. Specifically,  $C^{v_i}[v_i]$  equals the number of cycles that pass through node  $v_i$  at the  $i$ -th step, plus 1. Then, for each node  $u \in \mathcal{U}_i$ ,*

$$E[C^{v_i}[u]] = \frac{\mathcal{S}[u, v_i]^2}{\mathcal{S}[v_i, v_i]} \cdot d_u, \quad (12)$$

where  $\mathcal{S}[:, v_i]$  denotes the Cholesky factor defined in Eq. (11). Equivalently, for each node  $u \in \mathcal{U}_i$ ,

$$\mathcal{S}[u, v_i] = \frac{\sqrt{E[C^{v_i}[u]] \cdot E[C^{v_i}[v_i]]}}{\sqrt{d_u \cdot d_{v_i}}}. \quad (13)$$

This lemma naturally provides a *layer-wise* interpretation of the Cholesky factor  $\mathcal{S}[:, v_i]$  via LERW. The layer-wise cycle count  $C^{v_i}[u]$  is an unbiased estimator of  $d_u \mathcal{S}[u, v_i]^2 / \mathcal{S}[v_i, v_i]$ . Aggregating these layer-wise counts yields a consistent estimator of  $\mathbf{L}_v^{-1}$  and resistance distance.

**LEMMA 3.2.** *For any  $s, t \in \mathcal{V} \setminus \{v\}$ , let  $\max s, t$  denote the larger index of  $s$  and  $t$  in the node ordering  $\mathcal{O} = (v_1, \dots, v_n)$ , we have:*

$$(\mathbf{L}_v^{-1})_{st} = \sum_{i=\max\{s,t\}}^{n-1} \frac{\sqrt{E[C^{v_i}[s]]E[C^{v_i}[t]]}}{\sqrt{d_s \cdot d_t}}. \quad (14)$$

**LEMMA 3.3 (LOOP-ERASED WALK INTERPRETATION OF RESISTANCE DISTANCE).** *Given  $s, t \in \mathcal{V} \setminus \{v\}$ , we have:*

$$r(s, t) = \sum_{i=1}^{n-1} \left( \sqrt{\frac{E[C^{v_i}[s]]}{d_s}} - \sqrt{\frac{E[C^{v_i}[t]]}{d_t}} \right)^2. \quad (15)$$

**Algorithm 1:** Index Construction of Tree+DAG**Input:** Graph  $\mathcal{G}$ , degree threshold  $d$ , sample size  $\omega$ **Output:** Index structure Tree+DAG

- 1  $(\mathcal{V}_T, \mathcal{V}_C, \mathcal{T}) \leftarrow \text{CORETREEDecompose}(\mathcal{G}, d);$
- 2  $\text{TreeIndex} \leftarrow \text{BuildTreeIndex}(\mathcal{G}, \mathcal{V}_T, \mathcal{T});$
- 3  $\text{CoreIndex} \leftarrow \text{BuildCoreIndex}(\mathcal{V}, \mathcal{V}_C)$  for  $\omega$  samples;
- 4 **return**  $\text{TreeIndex}$  and  $\text{CoreIndex}$ ;

When drawing multiple LERW samples, the desired quantities admit Monte Carlo approximations. In Section 4, we leverage this observation to build an efficient DAG-based index. Consequently, the resistance distance  $r(s, t)$  can be answered by traversing only the cycles that involve  $s$  and  $t$ .

#### 4 The Proposed Solution: Tree+DAG

Building upon the theoretical results in Section 3, in this section, we present Tree+DAG, a unified index structure that integrates deterministic Cholesky-based computation for sparse regions and probabilistic cycle DAG-based approximation for dense regions.

##### 4.1 Overall framework

Real-world networks often exhibit a core-periphery structure [42]: a sparse, tree-like periphery where finding sparse cuts is efficient, and a dense, fast-mixing core where random walks are well-behaved. The proposed Tree+DAG index is motivated by this data property. We first partition the node set into two disjoint subsets,  $\mathcal{V} = \mathcal{V}_T \cup \mathcal{V}_C$ , where  $\mathcal{V}_T$  denotes the tree-like periphery and  $\mathcal{V}_C$  denotes the dense core. Without loss of generality, we assume that  $\mathcal{V}_T$  is the first  $|\mathcal{V}_T|$  nodes in the node ordering  $\mathcal{O} = (v_1, \dots, v_n)$ . The last node  $v$  lies in the core part, we select it as an easy-to-hit node (e.g., the node with the highest degree) to reduce the time complexity following previous study [35].

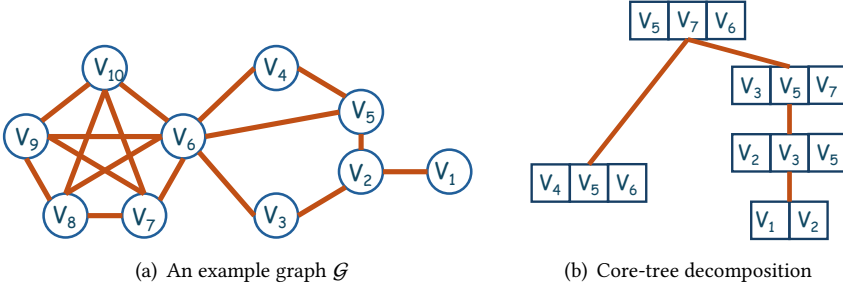
Building on Eq. (6), Eq. (5) and Eq. (11), we aim to compute and store  $\mathbf{L}_v^{-1}$  efficiently for fast queries. To this end, we decompose  $\mathbf{L}_v^{-1}$  as:

$$\mathbf{L}_v^{-1} = \sum_{k=1}^{|\mathcal{V}_T|} \frac{\mathbf{S}[:, v_k] \mathbf{S}[:, v_k]^\top}{\mathbf{S}[v_k, v_k]} + \sum_{k=|\mathcal{V}_T|+1}^{n-1} \frac{\mathbf{S}[:, v_k] \mathbf{S}[:, v_k]^\top}{\mathbf{S}[v_k, v_k]}. \quad (16)$$

Recall that TreeIndex [37] efficiently and exactly constructs the first term of Eq. (16) for graphs with small treewidth by partitioning the graph into a hierarchical tree structure and exploiting the sparsity of the Cholesky factor. As the graph becomes dense and small vertex cuts disappear, however, the time complexity of TreeIndex grows sharply, rendering it impractical for large graphs.

Accordingly, Tree+DAG stores the first term of Eq. (16) with a truncated TreeIndex on  $\mathcal{V}_T$  and approximates the second term with COREINDEX, which retains the cycle information assigned to core layers under the node-based ordering.

For a resistance-distance query  $r(s, t)$ , Lemma 3.2, together with the results of TreeIndex [37], computes  $\mathbf{L}_v^{-1}$  by combining the index structure of TreeIndex and COREINDEX. We independently compute  $(\mathbf{L}_v^{-1})_{ss}$ ,  $(\mathbf{L}_v^{-1})_{tt}$  and  $(\mathbf{L}_v^{-1})_{st}$ , and combine them to get the final result. Intuitively, a query proceeds in two phases: (i) traverse the TreeIndex from  $s$  and  $t$  to their respective root vertices, and (ii) consult the DAG index to account for cycles involving  $s$  and  $t$ . Because these rootward paths are short and the number of relevant cycles is small under our construction, queries run extremely fast.

Fig. 3. An example graph  $\mathcal{G}$  and core-tree decomposition

#### 4.2 Index structure and construction

The Tree+DAG construction consists of three steps: (i) core-tree decomposition, (ii) TreeIndex construction and (iii) cycle DAG construction. The overall construction process is shown in Algorithm 1.

*Step (i): Core-Tree Decomposition.* The first step is to expose the graph's core-periphery structure by partitioning  $\mathcal{V}$  into a tree-like periphery  $\mathcal{V}_T$  and a dense core  $\mathcal{V}_C$ . We adopt a minimum-degree elimination heuristic following [32, 42]: iteratively select a vertex of minimum degree, remove it, and add fill-in edges so that its neighbors form a clique. Upon eliminating a vertex  $u$ , we assign its parent to be the smallest-index neighbor present at that time. The resulting elimination forest guarantees that (i) every neighbor of  $u$  becomes an ancestor of  $u$ , and (ii) removing all ancestors of  $u$  disconnects  $u$ 's child branches. If the process runs to completion, it yields a valid tree decomposition. On real-world graphs, however, intermediate degrees can blow up. We therefore truncate the process as soon as the current neighborhood size exceeds a threshold  $d$ . This ensures that the subgraph induced by the eliminated vertices has treewidth at most  $d$ , while the remaining vertices are treated as the core  $\mathcal{V}_C$ . The core-tree decomposition runs in  $O(n(\text{tw}(\mathcal{G})^2 + \log n))$  time [42]. Full algorithmic details are provided in the full version [38].

**EXAMPLE 5.** Fig. 3(b) illustrates a core-tree decomposition with threshold  $d = 4$ . Starting from  $v_1$ , the algorithm iteratively eliminates  $v_2, v_3, v_4, v_5$ . When  $v_3$  is removed, its current neighbors are  $v_5$  and  $v_7$ , which are recorded in the elimination tree. We stop before eliminating  $v_6$  because its neighborhood size would exceed the threshold  $d$ . The resulting sets are  $\mathcal{V}_T = \{v_1, v_2, v_3, v_4, v_5\}$  and  $\mathcal{V}_C = \{v_6, v_7, v_8, v_9, v_{10}\}$ .

*Step (ii): TreeIndex Construction.* After partitioning the graph, we apply TreeIndex [37] to the tree-like periphery  $\mathcal{V}_T$ , truncating the construction once all nodes in  $\mathcal{V}_T$  have been processed. This truncation is safe, because it computes

$$\mathbf{L}_{\mathcal{V}_T \mathcal{V}_T}^{-1} = \sum_{i=1}^{|\mathcal{V}_T|} \frac{\mathbf{S}[:, v_i] \mathbf{S}[:, v_i]^\top}{\mathbf{S}[v_i, v_i]}. \quad (17)$$

Since the TreeIndex structure may be a forest rather than a single tree,  $\mathbf{L}_{\mathcal{V}_T \mathcal{V}_T}^{-1}$  can be a block-diagonal matrix, with each block corresponding to a connected component separated by  $\mathcal{V}_C$ .

**Index structure.** TreeIndex exploits the sparsity of the Cholesky factor  $\mathbf{S}$  by storing only its non-zero entries consistent with the hierarchical tree  $\mathcal{T}$  from the core-tree decomposition. For each  $v_i \in \mathcal{V}_T$ , an entry  $\mathbf{S}[u, v_i]$  is nonzero if and only if  $u$  is a descendant of  $v_i$  in  $\mathcal{T}$  (equivalently,  $v_i$  lies on the path from  $u$  to the root), i.e.,  $u \in \text{SubTree}(v_i)$ . Consequently, the index size is  $\sum_{u \in \mathcal{V}_T} |\text{SubTree}(u)|$ , which in the worst case is  $O(|\mathcal{V}_T| \cdot h_{\mathcal{V}_T})$ , where  $h_{\mathcal{V}_T}$  is the height of  $\mathcal{T}$  restricted to  $\mathcal{V}_T$ . In practice, it scales with the average path length  $\bar{h}_{\mathcal{V}_T}$  from each node to the root, i.e.,  $O(|\mathcal{V}_T| \cdot \bar{h}_{\mathcal{V}_T})$ , while  $\bar{h}_{\mathcal{V}_T}$  is very small (3.3 on Twitter when  $d = 290$ ).

**EXAMPLE 6.** Fig. 4(b) illustrates the index structure of TreeIndex with tree  $\mathcal{T}$ . Each node  $v_i$  stores the nonzeros in the column  $\mathcal{S}[:, v_i]$  restricted to its subtree. For example,  $v_3$  stores  $\mathcal{S}[v_1, v_3]$ ,  $\mathcal{S}[v_2, v_3]$ , and  $\mathcal{S}[v_3, v_3]$ , since  $v_1, v_2$ , and  $v_3$  lie in  $\text{SubTree}(v_3)$ . Summing over all nodes gives an index size of 12 in this example. In the whole  $\mathcal{G}$  (Fig. 4(a)), it requires 42.

**Index construction.** TreeIndex follows a bottom-up manner for construction. For each node  $v_i \in \mathcal{V}_T$ , we compute the column  $\mathcal{S}[:, v_i]$  by querying each neighbor  $u \in \mathcal{N}(v_i) \cap \mathcal{U}_i$ , traversing from  $u$  toward  $v_i$  and aggregating the subtrees encountered. The construction time is

$$\sum_{u \in \mathcal{N}(v_i) \cap \mathcal{U}_i} \sum_{w \in \mathcal{P}_{u \leadsto \text{root}}} |\text{SubTree}(w)|,$$

which is bounded in the worst case by  $O(|\mathcal{V}_T| \cdot h_{\mathcal{V}_T}^2 \cdot d_{\max}^{\mathcal{V}_T})$ , where  $d_{\max}^{\mathcal{V}_T}$  is the maximum degree of nodes in  $\mathcal{V}_T$  in the original graph. In practice, the time complexity can be expressed by  $\mathcal{T}_{\text{cons}}$ . Due to space limits, further details of TreeIndex are provided in the full version [38].

Empirically, after the core-tree decomposition, TreeIndex excels on the tree portion: for graphs that exhibit construction time bottlenecks, the first 90% of nodes build quickly, whereas the final 10% can take up to  $10^3 \times$  longer [37]. This corroborates that the decomposition exposes a pronounced core-periphery structure and that TreeIndex is best suited to the tree part.

**Step (iii): COREINDEX Construction.** The dense core  $\mathcal{V}_C$  is where our main contribution lies. In this step, we compute

$$\mathbf{L}_v^{-1} - \begin{bmatrix} \mathbf{L}_{\mathcal{V}_T \mathcal{V}_T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \sum_{i=|\mathcal{V}_T|+1}^{n-1} \frac{\mathcal{S}[:, v_i] \mathcal{S}[:, v_i]^\top}{\mathcal{S}[v_i, v_i]}. \quad (18)$$

Because the sparse portion of  $\mathcal{G}$  has already been handled by TREEINDEX, random walks within the core are efficient. Nevertheless, applying random walks to the core alone is not sufficient. By Lemma 3.1, Eq. (18) admits a representation in terms of cycles in the DAG. Crucially, not all cycles are required: it suffices to consider those encountered at the steps corresponding to nodes in  $\mathcal{V}_C$ .

**Index structure.** The COREINDEX is a collection of samples on cycle DAG obtained via LERW sampling. It mirrors a Wilson stack: each sampled cycle is assigned to a node  $v_i$ , indicating that it is traversed in layer  $v_i$ . For each node, we maintain a stack that stores its incident cycles together with their layer indices, enabling efficient lookup of sampled cycles that involve query nodes  $s$  and  $t$ . Let  $|C|$  denote the number of nodes in the cycle  $C$ . The index size of COREINDEX is  $O(\sum_{v_i \in \mathcal{V}_C} \sum_{C \in \mathcal{C}^{v_i}} |C|)$ . In practice, the index size is very small: under our core-tree decomposition, each node  $u$  stores, on average, only few cycles (0.040 for Twitter). The resulting COREINDEX occupies only few of the full Wilson stack (1.2% for Twitter, see Table 2), indicating that the core component is highly compact and well suited to LERW sampling.

**EXAMPLE 7.** Fig. 4(e) illustrates the index structure of COREINDEX: each sampled cycle is assigned to the layer corresponding to the node  $v_i$ , at which it is traversed under the node-based ordering  $O = (v_1, \dots, v_{10})$ . For example, node  $v_4$  stores cycle  $C_9$  in layer  $v_6$  and cycle  $C_{10}$  in layer  $v_9$ . In total, the index stores 2 cycles.

**Index construction.** Although only the cycles whose layers lie in  $\mathcal{V}_C$  must be stored, those layers depend on cycles sampled in earlier layers, which makes it difficult to construct the index. To address this, we propose a two-phase construction process for COREINDEX. (i) *Local LERW sampling.* Start a loop-erased walk from each node in  $\mathcal{V}_C$ , stopping each walk upon first hitting the maintained tree  $\mathcal{F}$ . For each node encountered in these random walks, we add it to a queue  $Q$ . After all nodes in  $\mathcal{V}_C$  have been added into  $\mathcal{F}$ , we continue by repeatedly popping a node from  $Q$ , growing a walk until it hits  $\mathcal{F}$ , until  $Q$  is empty. We record all erased cycles during the process; denote the resulting

**Algorithm 2:** BUILD COREINDEX ( $\mathcal{V}, \mathcal{V}_C$ )**Input:** Graph  $\mathcal{G}$ , node set  $\mathcal{V}_C$ , root node  $v$ **Output:** LAYER[ $C$ ] for each  $C \in \mathcal{C}^{\mathcal{V}_C}$ , STACK[ $u$ ] for each  $u \in \mathcal{V}$ 

```

1 Initialize  $\mathcal{C}_{walk}^{\mathcal{V}_C} \leftarrow \emptyset$ ,  $\mathcal{F} \leftarrow \{v\}$ , and STACK[ $u$ ]  $\leftarrow \emptyset$  for each  $u \in \mathcal{V}$ ;
2 Initialize a queue  $Q \leftarrow \mathcal{V}_C \setminus \{v\}$ , ENQUEUED  $\leftarrow \mathcal{V}_C \setminus \{v\}$ , and PROCESSED  $\leftarrow \emptyset$ ;
3 while  $Q \neq \emptyset$  do
4    $x \leftarrow \text{pop}(Q)$ ;
5   if  $x \notin \mathcal{F}$  and  $x \notin \text{PROCESSED}$  then
6      $(W, P, C_x) \leftarrow \text{LOOPERASED WALK}(x, \mathcal{F})$ ;
7      $\mathcal{F} \leftarrow \mathcal{F} \cup V(P)$ ;
8     for each erased cycle  $C \in C_x$  do
9        $\mathcal{C}_{walk}^{\mathcal{V}_C} \leftarrow \mathcal{C}_{walk}^{\mathcal{V}_C} \cup \{C\}$ ;
10      for each node  $u \in C$  do
11        STACK[ $u$ ].push( $C$ );
12      for each node  $u \in W \setminus \mathcal{F}$  do
13        if  $u \notin \text{ENQUEUED}$  then
14          Q.push( $u$ ); ENQUEUED  $\leftarrow \text{ENQUEUED} \cup \{u\}$ ;
15      PROCESSED  $\leftarrow \text{PROCESSED} \cup \{x\}$ ;
16 Initialize maxID[ $C$ ]  $\leftarrow \max_{u \in C} \text{pos}_O(u)$  for each  $C \in \mathcal{C}_{walk}^{\mathcal{V}_C}$ ;
17 Traverse STACK from bottom to top,
18 for each topological edge  $C' \rightsquigarrow C$  ( $C'$  must be visited before  $C$ ) do
19   maxID[ $C$ ]  $\leftarrow \max(\text{maxID}[C], \text{maxID}[C'])$ ;
20  $\mathcal{C}^{\mathcal{V}_C} \leftarrow \{C \mid \text{maxID}[C] > |\mathcal{V}_T|, C \in \mathcal{C}_{walk}^{\mathcal{V}_C}\}$ ;
21 LAYER[ $C$ ]  $\leftarrow \text{maxID}[C]$  for each  $C \in \mathcal{C}^{\mathcal{V}_C}$ ;
22 for each node  $u \in \mathcal{V}$  do
23   STACK[ $u$ ]  $\leftarrow \{C \in \text{STACK}[u] \mid C \in \mathcal{C}^{\mathcal{V}_C}\}$ ;
24 return LAYER[ $C$ ] for  $C \in \mathcal{C}^{\mathcal{V}_C}$ , STACK[ $u$ ] for each  $u \in \mathcal{V}$ ;

```

set by  $\mathcal{C}_{walk}^{\mathcal{V}_C}$ . By the DAG structure, the collected cycles necessarily include every cycle whose assigned layer lies in  $\mathcal{V}_C$ , though some may be assigned to layers outside  $\mathcal{V}_C$ ; thus i.e.  $\mathcal{C}^{\mathcal{V}_C} \subset \mathcal{C}_{walk}^{\mathcal{V}_C}$ ; (ii) *Layer assignment by DAG propagation*. Thus, we traverse the cycle DAG from bottom to top in reverse topological order. For each cycle  $C \in \mathcal{C}_{walk}^{\mathcal{V}_C}$ , we maintain a maxID value maxID[ $C$ ], which is initialized as the maximum index-under the fixed ordering  $O = (v_1, \dots, v_n)$ -among the nodes in  $C$ . Then, for each edge  $(C \rightsquigarrow C')$ , if maxID[ $C'$ ] > maxID[ $C$ ], we update maxID[ $C$ ]  $\leftarrow$  maxID[ $C'$ ]. After propagation, assign  $C$  to the layer of node  $v = \text{maxID}[C]$  in the *node-based* ordering. Finally, retain only those cycles whose assigned layer lies in  $\mathcal{V}_C$  to form the COREINDEX. The expected index construction time is  $O(\sum_{C \in \mathcal{C}_{walk}^{\mathcal{V}_C}} |C|)$ . In practice, the index construction time can save as only a small portion compared to sampling whole LERW from scratch. The following lemma also shows the correctness of this construction process.

LEMMA 4.1. *The construction process of COREINDEX is correct.*

EXAMPLE 8. Fig. 4(c)-(e) illustrates the construction process of COREINDEX. We first sample LERW samples locally within  $\mathcal{V}_C$ ; the resulting trajectory and cycle set  $\mathcal{C}_{walk}^{\mathcal{V}_C}$  are shown in Fig. 4(d). We then assign a layer to each cycle: its layer is the largest node  $v_j$  (under the ordering  $O$ ) that is reachable from that cycle. For example, cycle  $C_{10}$  reaches  $v_9$ , so it is placed in layer  $v_9$ .

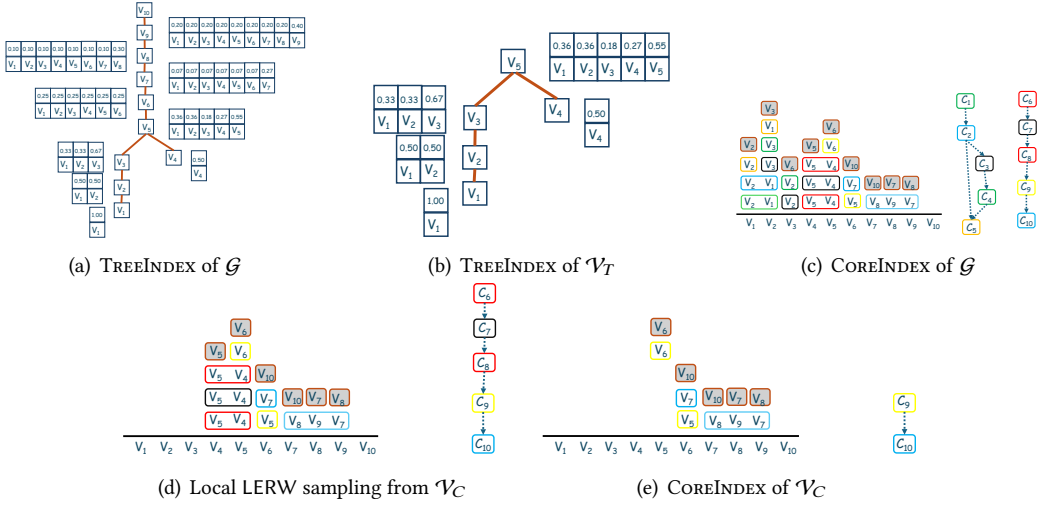


Fig. 4. Index structure of Tree+DAG. (a)-(b): TREEINDEX of  $\mathcal{G}$  and  $\mathcal{V}_T$  (after truncation). (c)-(e) The COREINDEX of  $\mathcal{G}$ . To obtain COREINDEX of  $\mathcal{V}_C$ , we only need to perform local LERW sampling (d) and remove cycles of former layers, obtaining (e). Both TREEINDEX and COREINDEX can reduce sizes by exploiting the core-periphery structure of  $\mathcal{G}$

*Index analysis.* Fig. 4 illustrates the overall structure of Tree+DAG. Fig. 4(a) and Fig. 4(c) show the full TREEINDEX and COREINDEX. After applying the core-tree decomposition, only the smaller component needs to be stored (Fig. 4(b) and Fig. 4(e)), substantially reducing space. We now analyze the space usage and construction time of Tree+DAG.

LEMMA 4.2 (INDEX SPACE). *The index space of Tree+DAG is*

$$O(|\mathcal{V}_T| \cdot \bar{h}_{\mathcal{V}_T}) + O(\omega \cdot \sum_{C \in \mathcal{C}^{\mathcal{V}_C}} |C|).$$

Compared to building indexes on the entire graph- $O(n \cdot \bar{h}_{\mathcal{G}})$  for TREEINDEX and  $O(\sum_{C \in \mathcal{C}^{\mathcal{V}}} |C|)$  for COREINDEX-Tree+DAG significantly reduces the index size via core-tree decomposition. In practice, the index size grows linearly with  $n$ . For example, on Dbpedia, the total index size is  $O(6.8 \times n)$  when  $d = 10$  and the sample size  $\omega = 100$ .

LEMMA 4.3 (INDEX CONSTRUCTION TIME). *The index construction time of Tree+DAG is*

$$O\left(\sum_{\substack{v_i \in \mathcal{V}_T, u \in N(v_i) \cap \mathcal{U}_i \\ w \in \mathcal{P}_{u \rightarrow v_i}}} |\text{SubTree}(w)| + \omega \left( |\mathcal{V}_{\text{walk}}^{\mathcal{V}_C}| + \sum_{C \in \mathcal{C}^{\mathcal{V}_C}} |C| \right)\right).$$

Compared to building indexes on the entire graph, the index building time of Tree+DAG is also much smaller. In practice, the index building time also grows linearly. For example, on the Dbpedia dataset, the theoretical index construction time is  $O(46 \times n)$  in practice when  $d = 10$  and  $\omega = 100$ .

### 4.3 Query Processing

We now introduce our query processing algorithm with Tree+DAG. According to Eq. (11) Eq. (6) and Eq. (5), the resistance distance  $r(s, t)$  can be represented by:

$$\sum_{i=1}^{|\mathcal{V}_T|} \frac{(S[s, v_i] - S[t, v_i])^2}{S[v_i, v_i]} + \sum_{|\mathcal{V}_T|+1}^{n-1} \frac{(S[s, v_i] - S[t, v_i])^2}{S[v_i, v_i]}. \quad (19)$$

**Algorithm 3:** Query processing with Tree+DAG**Input:** TREEINDEX, COREINDEX, query nodes  $s, t$ , sample size  $\omega$ **Output:** Estimated resistance distance  $\widehat{r}(s, t)$ 


---

```

1  $\widehat{r}(s, t) \leftarrow 0$ ;
2 for each node  $v_i \in \mathcal{P}_{s \leadsto \text{root}(s)} \cup \mathcal{P}_{t \leadsto \text{root}(t)}$  do
3    $\widehat{r}(s, t) \leftarrow \widehat{r}(s, t) + \frac{(S[s, v_i] - S[t, v_i])^2}{S[v_i, v_i]}$ ;
4  $\widehat{S}_s \leftarrow \emptyset, \widehat{S}_t \leftarrow \emptyset$ ;
5 for each cycle  $C \in \mathcal{C}^{\mathcal{V}_C}[s]$  do
6   if  $(\text{LAYER}[C], x) \notin \widehat{S}_s$  then
7      $\widehat{S}_s.\text{push}(\text{LAYER}[C], 0)$ ;
8      $\widehat{S}_s[\text{LAYER}[C]] \leftarrow \widehat{S}_s[\text{LAYER}[C]] + \frac{1}{\omega}$ .
9 for each cycle  $C \in \mathcal{C}^{\mathcal{V}_C}[t]$  do
10  if  $(\text{LAYER}[C], x) \notin \widehat{S}_t$  then
11     $\widehat{S}_t.\text{push}(\text{LAYER}[C], 0)$ ;
12     $\widehat{S}_t[\text{LAYER}[C]] \leftarrow \widehat{S}_t[\text{LAYER}[C]] + \frac{1}{\omega}$ .
13 if  $s \in \mathcal{V}_C$  ( $t \in \mathcal{V}_C$ ) then
14    $\widehat{S}_s[s] \leftarrow \widehat{S}_s[s] + 1$  ( $\widehat{S}_t[t] \leftarrow \widehat{S}_t[t] + 1$ );
15 for each  $(\text{LAYER}[C], x)$  in  $\widehat{S}_s \cup \widehat{S}_t$  do
16    $\widehat{r}(s, t) \leftarrow \widehat{r}(s, t) + \left( \sqrt{\frac{\widehat{S}_s[\text{LAYER}[C]]}{d_s}} - \sqrt{\frac{\widehat{S}_t[\text{LAYER}[C]]}{d_t}} \right)^2$ .
17 return  $\widehat{r}(s, t)$ ;

```

---

Putting the locality properties of TREEINDEX and COREINDEX together, we establish that:

LEMMA 4.4 (RESISTANCE DISTANCE WITH CORE-TREE DECOMPOSITION). *Given  $s, t \in \mathcal{V}$ , we have:*

$$r(s, t) = \sum_{v_i \in \mathcal{P}_{s \leadsto \text{root}(s)} \cup \mathcal{P}_{t \leadsto \text{root}(t)}} \frac{(S[s, v_i] - S[t, v_i])^2}{S[v_i, v_i]} + \sum_{v_i \in \mathcal{V}_C} \left( \sqrt{\frac{E[C^{v_i}[s]]}{d_s}} - \sqrt{\frac{E[C^{v_i}[t]]}{d_t}} \right)^2, \quad (20)$$

According to Lemma 4.4, a striking feature of TREEINDEX is that queries can be answered by touching only a small fraction of the index. This property persists under truncation: we can compute  $r(s, t)$  by accumulating contributions along the paths from  $s$  and  $t$  to their roots in  $\mathcal{T}$ . Moreover, by Lemma 4.4, COREINDEX shares this locality: the second term is obtained by accessing only the cycles involving  $s$  and  $t$  in  $\mathcal{C}^{\mathcal{V}_C}$ . While Lemma 4.4 in principle requires accounting for the contribution of every layer  $v_i \in \mathcal{V}_C$ , in practice only a few layers are visited with high probability—those touched by the sampled cycles through  $s$  and  $t$ . Consequently, we approximate only the few large, non-negligible entries and safely omit near-zero terms, yielding high query efficiency.

The overall query processing algorithm is illustrated in Algorithm 3. Given nodes  $s, t$  and the constructed TREEINDEX and COREINDEX, we first traverse the paths from  $s$  and  $t$  to their roots in  $\mathcal{T}$  and accumulate the contribution of each node (Line 2-3). For the core component, we maintain two sets of pairs  $(\text{LAYER}[C], x)$  for  $s$  and  $t$ , recording each cycle's contribution to the Cholesky factor associated with its layer (Line 4-12). By Lemma 3.1, we refine the estimate by adding the cycle contribution whenever  $s$  or  $t$  lies in  $\mathcal{V}_C$  (Line 13-14), and then aggregate the contributions over all sampled layers to form the final estimate (Line 15-16).

LEMMA 4.5 (QUERY TIME). *The query time with Tree+DAG is*

$$O\left(|\mathcal{P}_{s \rightsquigarrow \text{root}(s)}| + |\mathcal{P}_{t \rightsquigarrow \text{root}(t)}| + \omega \cdot (|C^{\mathcal{V}_C}[s]| + |C^{\mathcal{V}_C}[t]|)\right).$$

*In average, the query time is  $O(\bar{h}_{\mathcal{V}_T} + \omega \cdot \overline{C^{\mathcal{V}_C}[\cdot]})$ .*

In practice, the average query time over TREEINDEX is bounded by  $\bar{h}_{\mathcal{V}_T}$ , the mean path length to the roots, while the expected query time over COREINDEX can be bounded by  $\overline{C^{\mathcal{V}_C}[\cdot]}$ , the average number of cycles involving a node in  $\mathcal{V}$ . Both quantities are very small after the core-tree decomposition. For example, on the Twitter dataset we have  $\bar{h}_{\mathcal{V}_T} = 3.3$  and  $\overline{C^{\mathcal{V}_C}[\cdot]} = 0.040$  with  $\omega = 100$  and  $d = 290$ .

LEMMA 4.6 (LAYER-WISE UNBIASEDNESS). *The layer-wise cycle-count estimator*

$$\hat{\mu}_i(u) = \frac{1}{\omega} \sum_{j=1}^{\omega} C_i^{(j)}[u]$$

*is unbiased for  $\mu_i(u) = \mathbb{E}[C^{v_i}[u]]$ , i.e.,  $\mathbb{E}[\hat{\mu}_i(u)] = \mu_i(u)$ .*

Since the tree part of Tree+DAG is computed exactly by TREEINDEX, the sampling error comes solely from the core part. For a fixed query pair  $(s, t)$ , we analyze the estimation error of COREINDEX.

For the  $i$ -th core layer  $v_i \in \mathcal{V}_C$ , let  $C_i[s]$  and  $C_i[t]$  denote the retained cycle count passing through  $s$  and  $t$  in a single Wilson sample, with expectations  $\mu_i^s = \mathbb{E}[C_i[s]]$  and  $\mu_i^t = \mathbb{E}[C_i[t]]$ . Using  $\omega$  independent samples, the sample means are  $\hat{\mu}_i^s = \frac{1}{\omega} \sum_{j=1}^{\omega} C_i^{(j)}[s]$  and  $\hat{\mu}_i^t = \frac{1}{\omega} \sum_{j=1}^{\omega} C_i^{(j)}[t]$ . The true core contribution and its plug-in estimator are

$$R_C(s, t) = \sum_{v_i \in \mathcal{V}_C} \left( \sqrt{\frac{\mu_i^s}{d_s}} - \sqrt{\frac{\mu_i^t}{d_t}} \right)^2, \quad \hat{R}_C(s, t) = \sum_{v_i \in \mathcal{V}_C} \left( \sqrt{\frac{\hat{\mu}_i^s}{d_s}} - \sqrt{\frac{\hat{\mu}_i^t}{d_t}} \right)^2.$$

Let  $d_{\min} = \min(d_s, d_t)$ . We use three query-local quantities in the bound:  $\kappa_{st} = \sum_{v_i \in \mathcal{V}_C} (\mu_i^s + \mu_i^t)$  denotes the total expected cycle mass incident to the query endpoints,  $\Sigma_{st} = \sum_{v_i \in \mathcal{V}_C} [\text{Var}(C_i[s]) + \text{Var}(C_i[t])]$  denotes the total variance of these cycle counts, and  $\ell_{st} = |\{v_i \in \mathcal{V}_C : \mu_i^s + \mu_i^t > 0\}|$  denotes the number of active core layers contributing to the query.

THEOREM 4.7 (ESTIMATION ERROR BOUND). *For any  $\delta \in (0, 1)$ , with probability at least  $1 - \delta$ :*

$$|\hat{R}_C(s, t) - R_C(s, t)| \leq \frac{4}{d_{\min}} \sqrt{\kappa_{st} \left( \frac{2\ell_{st}\Sigma_{st}}{\omega\delta} \right)^{1/2}} + \frac{2}{d_{\min}} \left( \frac{2\ell_{st}\Sigma_{st}}{\omega\delta} \right)^{1/2}. \quad (21)$$

*Consequently, to guarantee  $|\hat{R}_C(s, t) - R_C(s, t)| \leq \epsilon$  with probability at least  $1 - \delta$ , it suffices to take*

$$\omega = O\left( \frac{\ell_{st} \Sigma_{st} \kappa_{st}^2}{\delta \epsilon^4 d_{\min}^4} + \frac{\ell_{st} \Sigma_{st}}{\delta \epsilon^2 d_{\min}^2} \right). \quad (22)$$

Theorem 4.7 gives a query-local sample-complexity guarantee for COREINDEX: to achieve absolute core-estimation error at most  $\epsilon$  with probability at least  $1 - \delta$ , it suffices to choose the sample size according to the active-layer count  $\ell_{st}$ , the total cycle-count variance  $\Sigma_{st}$ , and the expected endpoint cycle mass  $\kappa_{st}$ . Thus, the required number of Wilson samples can be bounded by local cycle statistics around the query nodes rather than the total graph size.

In practice, the quantities appearing in this bound are small after the core-tree decomposition on the graph families targeted by Tree+DAG. First, the number of retained cycles incident to an endpoint is small, as shown by  $\overline{C^{\mathcal{V}_C}[\cdot]}$  in Table 2. For example, on Twitter with  $d = 290$ , a node is

Table 1. Statistics of datasets ( $\bar{d}$  is the average degree;  $d_{med}$  is the median degree;  $d_{max}$  is the maximum degree;  $\mu_2$  is the second-largest eigenvalue of the transition matrix  $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ )

| Dataset    | $n$        | $m$         | $\bar{d}$ | $d_{med}$ | $d_{max}$ | $\mu_2$   | Type           |
|------------|------------|-------------|-----------|-----------|-----------|-----------|----------------|
| DBLP       | 1,167,956  | 5,200,418   | 8.91      | 4         | 1,545     | 0.9985185 | Coauthorship   |
| RoadNet-PA | 1,087,562  | 1,541,514   | 2.83      | 3         | 9         | 0.9999994 | Road Network   |
| RoadNet-CA | 1,957,027  | 2,760,388   | 2.82      | 3         | 12        | 0.9999998 | Road Network   |
| Wikitalk   | 2,388,953  | 4,656,682   | 3.90      | 1         | 100,029   | 0.9831960 | Communication  |
| Stack      | 2,572,345  | 28,177,464  | 21.91     | 4         | 44,065    | 0.9491429 | Interaction    |
| Youtube    | 3,216,075  | 9,369,874   | 5.83      | 2         | 91,751    | 0.9993319 | Social Network |
| Dbpedia    | 3,915,921  | 12,577,253  | 6.42      | 2         | 469,692   | 0.9996738 | Web Graph      |
| Friendster | 8,556,203  | 45,604,593  | 10.66     | 2         | 3,004     | 0.9981833 | Social Network |
| Twitter    | 21,297,772 | 265,025,545 | 24.89     | 3         | 698,112   | 0.9998529 | Social Network |
| Sina       | 58,655,820 | 261,321,018 | 8.91      | 2         | 278,489   | 0.9988599 | Social Network |

involved in only 0.040 retained core cycles on average per Wilson sample. Second, the number of active layers  $\ell_{st}$  cannot exceed the number of retained cycles touched by the two query endpoints, since a layer contributes only when one of these endpoint cycles is assigned to that layer. Third, the measured cycle-count variance is also modest: Table 2 reports  $\widehat{\Sigma}_{st} = 0.159$  on Twitter with  $d = 290$ , averaged over 500 random query pairs. The theorem is therefore best viewed as a conservative worst-case concentration bound over query-local parameters. When the two rootward paths and endpoint incident-cycle lists remain small, query latency is largely insensitive to the total graph size on the evaluated core-periphery graphs; the empirical convergence in Section 5 is also typically better than the bound suggests, as verified by the accuracy analysis.

Moreover, as with TREEINDEX, accuracy can be further improved by computing a larger portion of the TREEINDEX exactly. In our experiments, to reach the same accuracy, Tree+DAG is at least an order of magnitude faster than previous methods.

## 5 Experiments

### 5.1 Experimental setting

**Datasets.** To evaluate our approach, we employed a collection of 10 large datasets, exhibiting diverse sizes and structures, as detailed in Table 1. To better characterize skewed degree distributions, Table 1 reports both the average degree and the median degree. These datasets comprise both road and social networks and were sourced from [28, 30]. There are road networks, social networks, collaboration networks and web graphs. As indicated by the second-largest transition-matrix eigenvalue  $\mu_2$ , the road networks have  $\mu_2$  very close to 1, corresponding to a small spectral gap and indicating that they are comparatively easy to cut. In contrast, the other datasets generally exhibit smaller  $\mu_2$  values, consistent with an underlying core-periphery organization. We generate 1,000 random node pairs for each dataset to evaluate query performance. Following previous studies [15, 35, 39], we generate ground truth resistance by invoking the push [35] algorithm with threshold below  $10^{-17}$ . We use the absolute error to evaluate the performance of different algorithms. The absolute error is defined as  $|\widehat{r}(s, t) - r(s, t)|$ , where  $r(s, t)$  is the true resistance distance and  $\widehat{r}(s, t)$  is the estimation.

**Different methods.** We implement our approach as Tree+DAG. The algorithm has two parameters: the degree threshold  $d$  and the sample size  $\omega$ . The threshold  $d$  governs the split between  $\mathcal{V}_T$  and  $\mathcal{V}_C$  in the core-tree decomposition—larger  $d$  yields a larger  $\mathcal{V}_T$ . We select three values of  $d$  for each dataset:  $d = 10$ ,  $d = 50$ , and  $d = \max$ , which is obtained by setting a threshold 1,000 seconds for the construction of TREEINDEX. We further analyze how performance varies with  $d$  and outline guidelines for selecting  $d$  within Exp-VI. The sample size  $\omega$  controls the approximation accuracy. Unless otherwise specified, we set  $\omega = 500$  by default. We compare our approach with two SOTA index-based methods: TreeIndex [37] and LEIndex [39]. Specifically, TreeIndex [37] is a special case

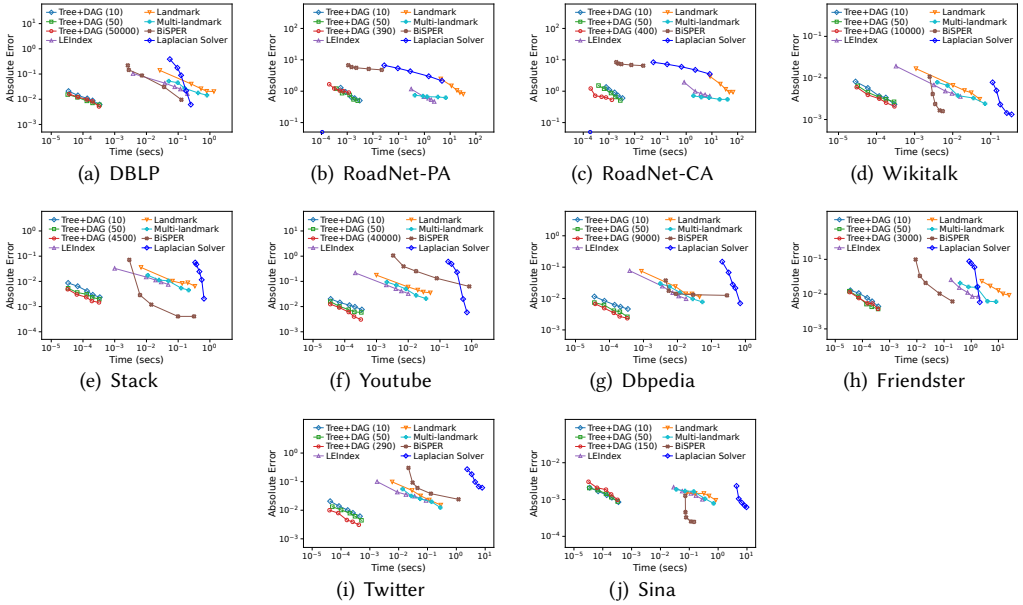


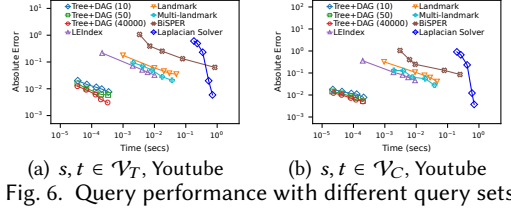
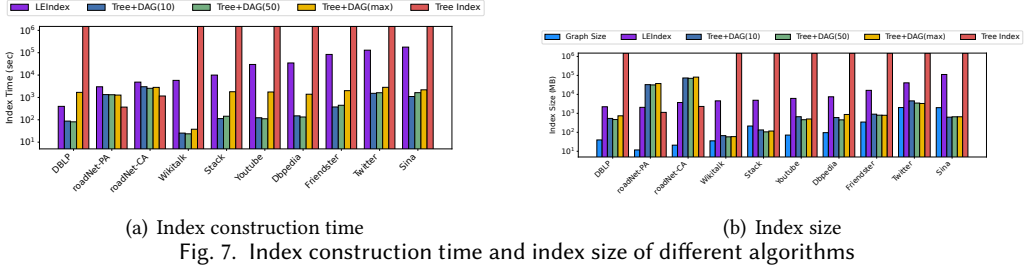
Fig. 5. Query performance of different algorithms

of Tree+DAG obtained when setting  $d = \infty$ . Although deterministic with very small absolute error, in our experiments it could build an index on only 2 road networks. LEIndex [39] uses spanning forests with root  $\mathcal{V}_l$  sampled by LERW to approximate the Schur complement. In our comparison, we follow the released implementation and compute the landmark-related index with a Laplacian solver [20], which improves the accuracy while keeping the construction cost small because the landmark set is tiny. Following [39], we select the 10 highest-degree nodes to build the LEIndex indices. To audit this solver-aided implementation, we also include the original Multi-landmark implementation from [39]; following its recommended setting, we use degree+ landmark selection, with  $|\mathcal{V}_l| = 10$  on non-road graphs and  $|\mathcal{V}_l| = 100$  on road networks. We also include two SOTA online methods, Landmark [35] and BiSPER [15], in our comparisons. We further include a direct Laplacian solver that solves a large Laplacian linear system independently for each query. For all approximation methods, there is a tolerance parameter  $\epsilon$  to control accuracy. Unless otherwise noted, we set  $\epsilon = 0.5$ , following [15, 35, 39].

**Experimental environment.** All experiments are conducted on a Linux server with Intel Xeon E5-2680 v4 CPU and 512GB memory. The algorithms are implemented in C++ and compiled with GCC 7.5.0. The experiments are conducted in a single thread.

## 5.2 Performance evaluation

**Exp-I: Query performance.** We first compare the query performance of the algorithms. For Tree+DAG, we vary the sample size  $\omega$  from 100 to 500. For the other approximate methods, we vary the tolerance parameter  $\epsilon$  from 0.5 down to 0.1, and report query time versus absolute error. Because TREEINDEX is exact, larger  $d$  yields higher accuracy for our approach. Results are shown in Fig. 5. Across all datasets, Tree+DAG is at least an order of magnitude faster than LEIndex, followed by the two online baselines BiSPER and Landmark. The original Multi-landmark implementation is generally no better than our solver-aided LEIndex baseline, and Tree+DAG still gives a better accuracy–efficiency trade-off. The direct Laplacian solver is exact but must solve one large linear system per query, making it slower than specialized single-pair methods on most datasets. Varying



$d$  yields similar query times, while larger  $d$  provides slightly lower error. For example, on Twitter, Tree+DAG achieves an absolute error of  $10^{-2}$  in  $8 \times 10^{-5}$  seconds, whereas the fastest existing method, LEIndex, requires over 0.7 seconds—more than 3 orders of magnitude slower. TreeIndex cannot construct an index for most graphs; it succeeds only on two road networks, which corresponds to the special case of Tree+DAG with  $d = \infty$ . The exact TreeIndex query times on these two road-network datasets are  $1.07 \times 10^{-4}$  seconds on RoadNet-PA and  $1.86 \times 10^{-4}$  seconds on RoadNet-CA; since TreeIndex is exact, the corresponding markers are placed at the lower edge of the log-scale error axis only to indicate query time. In contrast, Tree+DAG handles all remaining graphs in our experiments and delivers substantial speedup.

**Exp-II: Query performance with different query sets.** We further evaluate Tree+DAG's query performance on Youtube across different query sets; results on the other datasets are similar. We sample 1,000 node pairs whose endpoints are both in  $\mathcal{V}_C$  or both in  $\mathcal{V}_T$ , and compare query time and absolute error against competing methods. As shown in Fig. 6, Tree+DAG remains competitive when the query nodes lie in either the core or the periphery, indicating that it handles diverse query sets effectively.

**Exp-III: Index construction time.** We then evaluate index construction time across algorithms. The results can be found in Fig. 7(a). As  $d$  increases, build time follows a U-shape—initially decreasing and then increasing. Overall, construction is fast; for example, on Twitter the index builds in just 18 minutes. Tree+DAG is substantially faster than LEIndex, since LEIndex must solve large Laplacian systems. TreeIndex fails to finish on all datasets except road networks; on those, however, Tree+DAG is slower than TreeIndex because LERW is slow on such graphs.

**Exp-IV: Index size.** We next compare the index sizes of the index-based methods (Fig. 7(b)). Tree+DAG's index is much smaller than TreeIndex's and scales to graphs with millions of nodes and billions of edges. On Twitter, for example, Tree+DAG's index is just 3.1GB—about  $1.5 \times$  the graph size. By contrast, LEIndex has a compact footprint of  $O(n \times |\mathcal{V}_I|)$ , as it stores only Schur-complement values for the top 10 high-degree nodes.

**Exp-V: Statistical analysis of TREEINDEX and COREINDEX.** We compute the quantities defined in Section 4 to assess overall complexity. The resulting statistics for TREEINDEX and COREINDEX are summarized in Table 2. Both the average tree height and the average number of cycles per node are small—for example, on Twitter with  $d = 290$ , the average height is 3.3 and the average cycles per node is 0.040—suggesting fast queries. The last column further reports the empirical

Table 2. Statistics of the Tree+DAG index.  $\bar{h}_{\mathcal{V}_T}$  is the average path length from each node in  $\mathcal{V}_T$  to its root in the hierarchical tree structure.  $\mathcal{T}_{cons} = \sum_{v_i \in \mathcal{V}_T, u \in N(v_i) \cap \mathcal{U}_i; w \in \mathcal{P}_{u \rightarrow v_i}} |\text{SubTree}(w)|$  is the cost of index construction of TREEINDEX.  $\sum_{C \in \mathcal{C}^{\mathcal{V}_C}} |C|$  is the sum of the size of the cycles in  $\mathcal{C}^{\mathcal{V}_C}$ .  $\sum_{C \in \mathcal{C}^{\mathcal{V}_C}_{walk}} |C|$  is the sum of the size of the cycles in  $\mathcal{C}^{\mathcal{V}_C}_{walk}$ .  $|\mathcal{V}^{\mathcal{V}_C}_{walk}|$  is the expected number of nodes passed while index construction.  $\overline{\mathcal{C}^{\mathcal{V}_C}[\cdot]}$  is the average number of cycles in  $\mathcal{C}^{\mathcal{V}_C}$  that involves a node  $u \in \mathcal{V}$ .  $\hat{\Sigma}_{st}$  is the average cycle-count variance over 500 random query pairs.

| Dataset    | $d$         | $\frac{ \mathcal{V}_T }{ \mathcal{V} }$ | TREEINDEX                 |                                                 |                      |                                                | COREINDEX                                                            |                                        |                                  |                     |  |
|------------|-------------|-----------------------------------------|---------------------------|-------------------------------------------------|----------------------|------------------------------------------------|----------------------------------------------------------------------|----------------------------------------|----------------------------------|---------------------|--|
|            |             |                                         | $\bar{h}_{\mathcal{V}_T}$ | $\bar{h}_{\mathcal{V}_T} \cdot  \mathcal{V}_T $ | $\mathcal{T}_{cons}$ | $\sum_{C \in \mathcal{C}^{\mathcal{V}_C}}  C $ | $\sum_{C \in \mathcal{C}^{\mathcal{V}_C}} \frac{\mathcal{V}_C}{ C }$ | $ \mathcal{V}^{\mathcal{V}_C}_{walk} $ | $\mathcal{C}^{\mathcal{V}_C}[-]$ | $\hat{\Sigma}_{st}$ |  |
| DBLP       | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 601,359                                        | 601,359                                                              | 1,167,956                              | 0.515                            | 3.574               |  |
|            | $d = 10$    | 0.808                                   | 2.569                     | 2,422,844                                       | 6,681,417            | 179,071                                        | 273,524                                                              | 393,517                                | 0.153                            | 0.648               |  |
|            | $d = 50$    | 0.892                                   | 4.074                     | 4,242,454                                       | 29,554,417           | 126,192                                        | 230,553                                                              | 305,899                                | 0.108                            | 0.418               |  |
|            | $d = 50000$ | 0.908                                   | 40.339                    | 42,783,884                                      | 6,347,015,307        | 112,011                                        | 217,990                                                              | 284,441                                | 0.096                            | 0.363               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 17,286,200                                     | 17,286,200                                                           | 1,087,562                              | 15.895                           | 610.546             |  |
| RoadNet-PA | $d = 10$    | 0.904                                   | 6.407                     | 6,300,538                                       | 18,018,780           | 15,342,800                                     | 17,066,000                                                           | 1,005,890                              | 14.108                           | 566.040             |  |
|            | $d = 50$    | 0.981                                   | 28.158                    | 30,034,125                                      | 550,129,351          | 13,300,950                                     | 16,424,900                                                           | 996,653                                | 12.230                           | 436.220             |  |
|            | $d = 390$   | 0.999                                   | 729,341                   | 792,924,480                                     | 324,141,342,082      | 7,083,020                                      | 16,358,700                                                           | 995,990                                | 6.513                            | 369.923             |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 38,860,400                                     | 38,860,400                                                           | 1,957,027                              | 19.857                           | 1,038.158           |  |
| RoadNet-CA | $d = 10$    | 0.911                                   | 7.064                     | 12,598,048                                      | 39,392,878           | 34,659,200                                     | 38,355,200                                                           | 1,837,960                              | 17.710                           | 846.429             |  |
|            | $d = 50$    | 0.983                                   | 31.664                    | 60,921,537                                      | 1,207,813,689        | 29,865,300                                     | 36,564,100                                                           | 1,812,830                              | 15.261                           | 749.948             |  |
|            | $d = 400$   | 1.000                                   | 650.033                   | 1,271,698,560                                   | 450,095,152,581      | 17,230,400                                     | 37,547,000                                                           | 1,813,610                              | 8.804                            | 707.641             |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 473,716                                        | 473,716                                                              | 2,388,953                              | 0.020                            | 0.040               |  |
| Wikitalk   | $d = 10$    | 0.979                                   | 1.019                     | 2,382,828                                       | 46,587               | 24,818.7                                       | 25,497.2                                                             | 68,367.4                               | 0.010                            | 0.016               |  |
|            | $d = 50$    | 0.992                                   | 1.052                     | 2,493,027                                       | 140,422              | 19,505.2                                       | 20,346.2                                                             | 35,983.1                               | 0.008                            | 0.014               |  |
|            | $d = 10000$ | 0.995                                   | 1.535                     | 3,648,212                                       | 4,596,400            | 15,417.5                                       | 16,493.8                                                             | 27,181.6                               | 0.006                            | 0.013               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 1,379,630                                      | 1,379,630                                                            | 2,572,345                              | 0.054                            | 0.155               |  |
| Stack      | $d = 10$    | 0.737                                   | 1.102                     | 2,088,544                                       | 195,928              | 57,755.7                                       | 61,585.3                                                             | 745,889                                | 0.022                            | 0.051               |  |
|            | $d = 50$    | 0.858                                   | 1.324                     | 2,921,364                                       | 791,381              | 38,673.6                                       | 43,719.3                                                             | 449,887                                | 0.015                            | 0.033               |  |
|            | $d = 4500$  | 0.896                                   | 2.722                     | 6,275,434                                       | 6,343,227            | 30,809.3                                       | 36,285                                                               | 354,288                                | 0.012                            | 0.025               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 2,026,430                                      | 2,026,430                                                            | 3,216,075                              | 0.630                            | 39.954              |  |
| Youtube    | $d = 10$    | 0.912                                   | 1.748                     | 5,124,994                                       | 3,181,949            | 189,795                                        | 327,836                                                              | 451,783                                | 0.059                            | 0.909               |  |
|            | $d = 50$    | 0.955                                   | 2.214                     | 6,800,417                                       | 6,845,852            | 108,537                                        | 215,151                                                              | 278,414                                | 0.034                            | 0.192               |  |
|            | $d = 40000$ | 0.967                                   | 7.273                     | 22,611,572                                      | 419,242,427          | 82,239.5                                       | 174,241                                                              | 225,360                                | 0.026                            | 0.142               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 9,585,690                                      | 9,585,690                                                            | 3,915,921                              | 0.245                            | 0.885               |  |
| Dbpedia    | $d = 10$    | 0.910                                   | 1.764                     | 6,287,548                                       | 16,892,658           | 208,969                                        | 289,145                                                              | 588,032                                | 0.053                            | 0.169               |  |
|            | $d = 50$    | 0.961                                   | 2.586                     | 9,730,933                                       | 56,203,696           | 119,674                                        | 199,238                                                              | 350,442                                | 0.031                            | 0.101               |  |
|            | $d = 9000$  | 0.971                                   | 19.188                    | 72,981,887                                      | 94,225,188,122       | 90,679.5                                       | 165,305                                                              | 287,084                                | 0.023                            | 0.082               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 8,232,540                                      | 8,232,540                                                            | 8,556,203                              | 0.096                            | 0.468               |  |
| Friendster | $d = 10$    | 0.829                                   | 1.128                     | 8,001,050                                       | 1,553,447            | 377,718                                        | 442,582                                                              | 1,752,200                              | 0.044                            | 0.157               |  |
|            | $d = 50$    | 0.918                                   | 1.300                     | 10,211,240                                      | 6,089,193            | 308,758                                        | 379,080                                                              | 1,092,430                              | 0.036                            | 0.097               |  |
|            | $d = 3000$  | 0.936                                   | 2.146                     | 17,193,600                                      | 82,071,070           | 272,506                                        | 346,544                                                              | 962,731                                | 0.032                            | 0.093               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 68,696,700                                     | 68,696,700                                                           | 21,297,772                             | 0.323                            | 1.939               |  |
| Twitter    | $d = 10$    | 0.766                                   | 1.605                     | 26,173,646                                      | 17,858,600           | 1,615,300                                      | 2,279,360                                                            | 6,553,680                              | 0.076                            | 0.336               |  |
|            | $d = 50$    | 0.852                                   | 2.275                     | 41,269,141                                      | 95,505,827           | 1,015,120                                      | 1,664,990                                                            | 4,746,760                              | 0.048                            | 0.170               |  |
|            | $d = 290$   | 0.876                                   | 3.297                     | 61,516,349                                      | 363,486,873          | 860,573                                        | 1,493,150                                                            | 4,219,410                              | 0.040                            | 0.159               |  |
|            | $d = 0$     | 0.000                                   | -                         | -                                               | -                    | 115,170                                        | 115,170                                                              | 58,655,820                             | 0.002                            | 0.0031              |  |
| Sina       | $d = 10$    | 0.920                                   | 1.000                     | 53,949,409                                      | 4,275                | 114,335                                        | 114,390                                                              | 4,818,540                              | 0.002                            | 0.0026              |  |
|            | $d = 50$    | 0.989                                   | 1.001                     | 58,045,293                                      | 52,615               | 112,772                                        | 112,874                                                              | 836,287                                | 0.002                            | 0.0024              |  |
|            | $d = 150$   | 0.996                                   | 1.002                     | 58,487,957                                      | 144,456              | 111,822                                        | 111,932                                                              | 477,650                                | 0.002                            | 0.0024              |  |

query-local variance  $\hat{\Sigma}_{st}$  used in Theorem 4.7. It is also small on the large non-road core-periphery graphs after decomposition; for example,  $\hat{\Sigma}_{st} = 0.159$  on Twitter, 0.082 on Dbpedia, and 0.0024 on Sina under the default largest- $d$  settings. We also observe that the construction cost  $\mathcal{T}_{con}$  can grow rapidly; however, with timely truncation the total build time remains near-linear. For instance, on Twitter with  $\omega = 100$ , the construction time is approximately  $O(0.51 \times n)$ . When  $\omega = 500$ , it is  $O(1.64 \times n)$ . Overall, these results indicate near-linear build time and index size, with fast query latency in practice on the evaluated graphs.

**Exp-VI: Performance when varying  $d$ .** In this experiment, we vary the fraction of nodes assigned to TREEINDEX,  $c \in [0, 1]$ , and compare query time, query error, index build time, and index size. When  $c = 0$ , we store only the DAG structure; when  $c = 1$ , the structure reduces to a pure TREEINDEX. Results on Youtube are shown in Fig. 8 (other datasets exhibit the same trends). As  $c$  increases, both query time and query error decrease monotonically, because TREEINDEX is exact and a larger tree portion yields faster, more accurate queries. In contrast, index size and build time follow a non-monotone (first-down-then-up) curve: when  $c$  is small, COREINDEX dominates and shrinking the core lowers cost; as  $c$  grows further, TREEINDEX becomes dominant and the cost rises again. This creates an optimal core-size setting—denote it by  $d$  (equivalently, an optimal  $c$ )—near the point where the TREEINDEX and COREINDEX contributions are balanced; the optimal  $d$  shifts with the accuracy target, moving higher when tighter error is required.

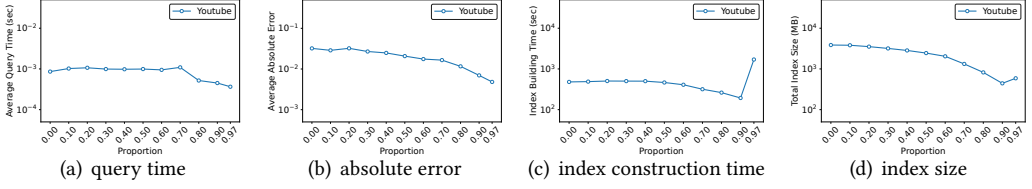
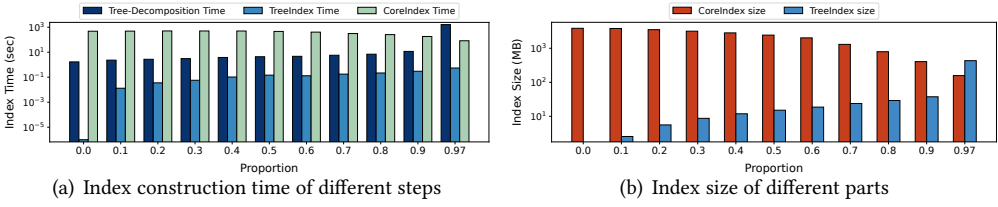
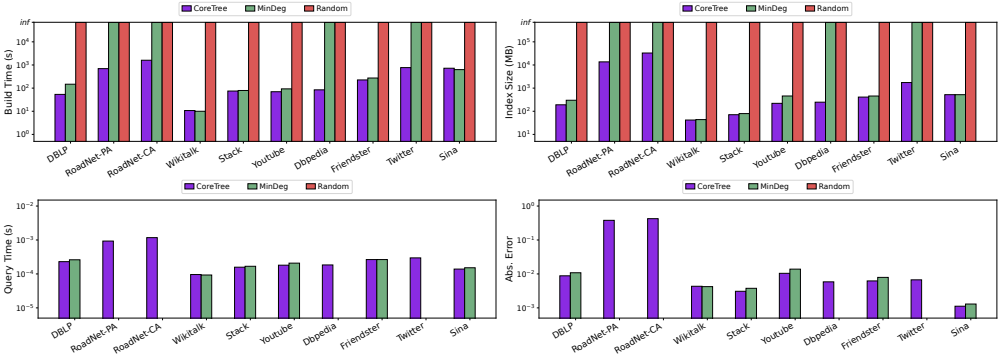
Fig. 8. Performance when varying  $d$  for YoutubeFig. 9. Index analysis when varying  $d$  for Youtube

Fig. 10. Ablation study of different node orderings across all 10 datasets

To make this decomposition explicit, Fig.9(a) breaks the build time into three stages: (i) core-tree decomposition, (ii) TREEINDEX construction, and (iii) COREINDEX construction. Early on, the core stage dominates; as  $d$  (or  $c$ ) increases, the tree stage takes over. Fig.9(b) shows an analogous split for index size, exhibiting the same transition from core-dominated to tree-dominated.

**Exp-VII: Ablation study of different node ordering.** Node ordering is a key design choice in Tree+DAG because it affects both components of the index: it controls the sparsity of Cholesky columns stored by TreeIndex and determines the layer assignment of erased cycles in CoreIndex. We compare three choices across all 10 datasets: CoreTree, the default ordering induced by the core-tree decomposition; MinDeg, a degree-ascending ordering; and Random, a uniformly random ordering. Fig. 10 reports index construction time, index size, query time, and absolute error. CoreTree is consistently more efficient than MinDeg, while Random is not scalable and fails to complete index construction on large datasets. This confirms that the core-tree ordering is important for both sparse tree labels and compact core-cycle storage; changing it can noticeably degrade construction cost, index size, query time, or accuracy.

**Exp-VIII: Accuracy analysis.** We analyze the accuracy of DAG-based COREINDEX. We set  $d = 0$ , so that no TREEINDEX is built. We vary the sample size  $\omega$  from 100 to 500 and compare the query error and variance with  $v$ -absorbed walk sampling (Landmark). Notice that other methods are not included in this experiment because they include deterministic computation, as TREEINDEX does,

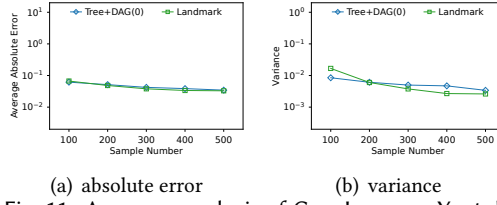


Fig. 11. Accuracy analysis of COREINDEX on Youtube

Table 3. System-level breakdown of online single-pair queries on four large non-road graphs with  $\omega = 500$ . Tree/List/Arith report fractions of one query's measured processing time; LLC miss and memory stall report hardware-counter measurements on the same online query workload.

| Dataset | $d$  | Tree (%) | List (%) | Arith (%) | LLC miss (%) | Mem. stall (%) |
|---------|------|----------|----------|-----------|--------------|----------------|
| Stack   | 4500 | 0.99     | 98.70    | 0.20      | 84.78        | 30.29          |
| Dbpedia | 9000 | 3.29     | 96.44    | 0.18      | 71.73        | 34.40          |
| Twitter | 290  | 0.55     | 99.30    | 0.10      | 73.35        | 32.28          |
| Sina    | 150  | 0.62     | 99.09    | 0.20      | 78.32        | 32.04          |

making it difficult to isolate the contribution of LERW sampling. For example, BiSPER precomputes  $P^k$  for small  $k$  to improve accuracy, thus it performs poorly on road networks which require a large  $k$ , while LEIndex precomputes  $\mathcal{S}[:, v_i]$  for large  $v_i$  to improve accuracy, but the number of  $v_i$  is very small (i.e., 10). From this perspective, Tree+DAG also precomputes  $\mathcal{S}[:, v_i]$  for the smallest  $v_i$  to improve accuracy; the nodes  $\mathcal{V}_T$  occupy a large proportion of the graph. The results are shown in Fig. 11. It can be observed that when  $\omega$  increases, the query error decreases. Tree+DAG achieves a competitive accuracy with Landmark, while being much faster. This validates that our estimator is effective via Rao–Blackwellization.

**Exp-IX: System profiling.** We further profile the online query process on four large non-road graphs with  $\omega = 500$ . The goal is to validate the query-local access pattern in Lemma 4.5: after preprocessing, a query should read only the two rootward paths in TREEINDEX and the endpoint-incident cycle lists in COREINDEX. The average sizes of these query-local structures are already reported in Table 2. Table 3 therefore focuses on system behavior: the time share of each query component and hardware-counter measurements for cache and memory stalls. Here, LLC miss denotes the last-level-cache load miss rate, and memory stall denotes the fraction of execution cycles stalled by the memory hierarchy. The breakdown separates the tree contribution from the core contribution. The tree contribution traverses the two rootward paths in TreeIndex, while the core-list phase reads the incident cycle/layer lists of the two query endpoints in CoreIndex. These lists are stored directly with the endpoints, so query processing performs array-based endpoint-list reads rather than a traversal over the full cycle DAG. The profile shows that the core-list phase dominates the online query cost, accounting for 96.4%–99.3% of the measured query-processing time, while the final arithmetic aggregation is below  $0.5 \mu s$  on all four datasets. The LLC miss and memory stall counters show that this dominant endpoint-list access is indeed memory-sensitive, with 71.73%–84.78% last-level-cache load misses and 30.29%–34.40% cycles stalled by memory. At the same time, Table 2 shows that the endpoint lists are very short after decomposition. Thus the bottleneck is a small number of endpoint-local list reads, not a large traversal of the cycle DAG, which supports the intended query-local behavior of Tree+DAG.

## 6 Case Studies

We present two case studies to illustrate why practical applications on core–periphery graphs often require many fast *single-pair* resistance-distance queries. Core–periphery structure has been observed and exploited in large graphs for personalized PageRank and shortest-path queries [32, 42].

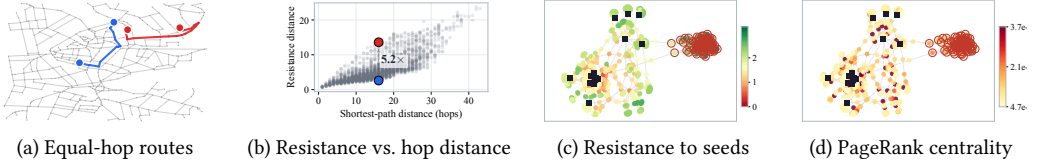


Fig. 12. Case studies of single-pair resistance-distance queries on transportation and social networks

We use two representative settings: robust planning on a road network and trust-region scoring on a social network.

**Case Study 1: Robust routing on transportation networks.** On the Oldenburg road network [7] (496 nodes, 622 edges), a planner may repeatedly ask whether a specific origin–destination pair is robust to local road closures. We sample 2,500 origin–destination pairs and compare their shortest-path distance with their resistance distance. As shown in Figure 12(a)–(b), two pairs sharing the same shortest-path distance (16 hops) can differ by a factor of  $5.2\times$  in resistance distance. The pair with low resistance ( $r = 2.6$ ) lies in an area with many alternative parallel routes, whereas the high-resistance pair ( $r = 13.6$ ) traverses a bridge-like bottleneck. Thus resistance distance gives a simple point-to-point robustness signal—many alternative routes versus a potential single point of failure—that shortest-path distance alone misses [48].

**Case Study 2: Sybil detection on social networks.** On a social trust network, a system may repeatedly score candidate accounts by their structural distance to a trusted region. We evaluate this use case on the Epinions trust network (75,879 users, 405,740 undirected trust edges) [47]. Following the semi-synthetic evaluation protocol of SybilRank [9], we extract a sparse peripheral subgraph (588 nodes, 943 edges; maximum degree  $\leq 50$ ) and inject 80 Sybil accounts connected internally by a sparse Erdős–Rényi random graph ( $p = 0.02$ ) plus a ring for connectivity, with 2 attack edges linking Sybils to peripheral honest users. We select 15 trusted seed nodes and compare four unsupervised scores: mean resistance distance from seeds, node degree, mean shortest-path distance from seeds, and PageRank centrality. In Figure 12(c)–(d), black squares mark trusted seeds, red circles mark injected Sybil accounts, and node colors encode the corresponding score. Resistance distance gives a clear trusted-region signal in this illustrative setting: the mean resistance from trusted seeds is 2.72 for Sybil accounts and 1.75 for honest accounts. If nodes are ranked by suspiciousness, the resistance score has a pairwise ranking score of 0.97, meaning that a randomly chosen Sybil account is ranked as more suspicious than a randomly chosen honest account in 97% of Sybil–honest comparisons. PageRank centrality gives a weaker node-level signal, with a pairwise ranking score of 0.78, because it does not explicitly measure structural separation from the trusted region. The advantage arises because resistance distance captures the bottleneck created by the few attack edges and the lack of route redundancy between Sybil accounts and trusted seeds. Shortest-path distance is also strong in this controlled injection because the Sybil cluster is separated by only a few attack edges, but it measures only hop-count separation and does not capture route redundancy. Together, the two case studies illustrate the same computational need: applications repeatedly evaluate many selected pairs online, from road origin–destination pairs to candidate account–trusted seed pairs.

## 7 Related work

**Computation of resistance distance.** Resistance distance computation is a fundamental problem in graph data management. A major theoretical line of work designs efficient Laplacian solvers [11, 13, 16, 20, 21, 23, 25, 29, 49] with near-linear time in the number of edges. However, these methods are largely of theoretical interest: in practice they often fail to scale truly linearly, and for very large graphs we would like to answer point-to-point queries without traversing the entire

graph. To this end, several algorithms [3, 8, 15, 17, 33, 35, 37, 39, 45, 61] compute resistance distance locally. As discussed in Section 2.1, random walk-based methods [15, 33, 35, 39, 45, 61] work well on fast-mixing graphs but struggle on small treewidth graphs, whereas Cholesky decomposition-based approaches [22, 37] exhibit the opposite behavior. By exploiting the core-periphery structure of large real-world graphs, our approach scales effectively across diverse graph families.

**Computation of other related quantities.** Beyond resistance distance, several related quantities admit similar computational ideas, including personalized PageRank (PPR) [14, 24, 34, 36, 41, 53–55, 58], heat-kernel methods [26, 62], and Kemeny-constant estimation [31, 52, 59, 60]. Some of these methods use random-walk, Monte Carlo, or random-forest sampling, but none establishes the layer-wise LERW–Cholesky connection studied here. While [42] uses core-tree decomposition to construct a preconditioner for PPR, the techniques are entirely different and do not support rapid per-query answers. To our knowledge, ours is the first approach that leverages core-tree decomposition to answer a random walk-based quantity with endpoint-local index queries in practice, with potential applicability to other such quantities. Shortest-path distance is also well studied in graph data management [1, 18, 19, 27]. Tree decompositions [10, 12, 44, 56] and core-tree decompositions [32] have been proposed to accelerate shortest-path queries. Resistance distance, however, is fundamentally different, and the requisite techniques differ accordingly.

## 8 Conclusion

In this paper, we study the problem of resistance distance computation. While existing random walk-based and Cholesky decomposition-based methods each excel on different graph structures, we propose Tree+DAG, a novel index structure that scales resistance distance queries across diverse large graphs. To bridge these two paradigms, we establish new theoretical insights linking loop-erased random walks with the Cholesky decomposition of the inverse Laplacian matrix—showing that the layer-wise cycle counts provide unbiased estimators of normalized squared Cholesky contributions. Based on this connection, we decompose the graph into a sparse, tree-like periphery and a dense core. The tree periphery is easy to separate by small vertex cuts, which motivates building a hierarchical tree index to deterministically store Cholesky factors. In contrast, the dense core mixes rapidly under random walks, so we design a compact index that stores only the few cycles produced by loop-erased walks. This hybrid design yields near-linear index size and construction time, and low endpoint-local query latency in practice. Extensive experiments on 10 large real-life datasets show that Tree+DAG achieves at least an order of magnitude faster query performance than existing methods while maintaining comparable accuracy.

## Acknowledgments

This work was partially supported by the NSFC Grants U24A20255, U2241211, and 62427808. Rong-Hua Li is the corresponding author of this paper.

## References

- [1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. 349–360.
- [2] Vedat Levi Alev, Nima Anari, Lap Chi Lau, and Shayan Oveis Gharan. 2018. Graph Clustering using Effective Resistance. In *ITCS*, Vol. 94. 41:1–41:16.
- [3] Eugenio Angriman, Maria Predari, Alexander van der Grinten, and Henning Meyerhenke. 2020. Approximation of the Diagonal of a Laplacian’s Pseudoinverse for Complex Network Analysis. In *ESA*, Vol. 173. 6:1–6:24.
- [4] Ravindra B. Bapat. 2010. *Graphs and Matrices*. Springer London.
- [5] Mitchell Black, Zhengchao Wan, Amir Nayyeri, and Yusu Wang. 2023. Understanding Oversquashing in GNNs through the Lens of Effective Resistance. In *ICML*. 2528–2547.

- [6] Francesco Bonchi, Claudio Gentile, Francesco Paolo Nerini, André Panisson, and Fabio Vitale. 2025. Fast and Effective GNN Training through Sequences of Random Path Graphs. In *KDD*. 49–60.
- [7] Thomas Brinkhoff. 2002. A Framework for Generating Network-Based Moving Objects. *GeoInformatica* 6, 2 (2002), 153–180.
- [8] Dongrun Cai, Xue Chen, and Pan Peng. 2023. Effective Resistances in Non-Expander Graphs. In *ESA*, Vol. 274. 29:1–29:18.
- [9] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. 2012. Aiding the Detection of Fake Accounts in Large Scale Social Online Services. In *Proc. NSDI*. 197–210.
- [10] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *VLDB J.* 21, 6 (2012), 869–888.
- [11] Chao Chen, Tianyu Liang, and George Biros. 2021. RCHOL: Randomized Cholesky Factorization for Solving SDD Linear Systems. *SIAM J. Sci. Comput.* 43, 6 (2021), C411–C438.
- [12] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient Distance Querying on Road Networks by Projected Vertex Separators. In *SIGMOD*. 313–325.
- [13] Paul F. Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. 2011. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *STOC*. 273–282.
- [14] Mustafa Coskun, Ananth Grama, and Mehmet Koyutürk. 2016. Efficient Processing of Network Proximity Queries via Chebyshev Acceleration. In *KDD*. 1515–1524.
- [15] Guanyu Cui, Hanzhi Wang, and Zhewei Wei. 2025. Mixing Time Matters: Accelerating Effective Resistance Estimation via Bidirectional Method. In *KDD*. 177–188.
- [16] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. 2019. Fully dynamic spectral vertex sparsifiers and applications. In *STOC*. 914–925.
- [17] Rajat Vadiraj Dwaraknath, Ishani Karmarkar, and Aaron Sidford. 2023. Towards Optimal Effective Resistance Estimation. In *NIPS*. 59034–59046.
- [18] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling - Scaling Up Distance Queries on Road Networks. *Proc. ACM Manag. Data* 1, 4 (2023), 244:1–244:25.
- [19] Muhammad Farhan, Henning Koehler, and Qing Wang. 2025. Dual-Hierarchy Labelling: Scaling Up Distance Queries on Dynamic Road Networks. *Proc. ACM Manag. Data* 3, 1 (2025), 35:1–35:25.
- [20] Yuan Gao, Rasmus Kyng, and Daniel A. Spielman. 2026. AC(k): Robust Solution of Laplacian Equations by Randomized Approximate Cholesky Factorization. *SIAM J. Sci. Comput.* 48, 3 (2026), A1643–A1684.
- [21] Yu Gao, Yang P. Liu, and Richard Peng. 2021. Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao. In *FOCS*. 516–527.
- [22] Craig Gotsman and Kai Hormann. 2023. Efficient Point-to-Point Resistance Distance Queries in Large Graphs. *J. Graph Algorithms Appl.* 27, 1 (2023), 35–44.
- [23] Monika Henzinger, Billy Jin, Richard Peng, and David P. Williamson. 2023. A Combinatorial Cut-Toggling Algorithm for Solving Laplacian Linear Systems. *Algorithmica* 85, 12 (2023), 3680–3716.
- [24] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibao Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proc. ACM Manag. Data* 1, 1 (2023), 25:1–25:26.
- [25] Arun Jambulapati and Aaron Sidford. 2021. Ultrasparse Ultrasparsifiers and Faster Laplacian System Solvers. In *SODA*. 540–559.
- [26] Kyle Kloster and David F. Gleich. 2014. Heat kernel based community detection. In *KDD*. 1386–1395.
- [27] Henning Koehler, Muhammad Farhan, and Qing Wang. 2025. Stable Tree Labelling for Accelerating Distance Queries on Dynamic Road Networks. In *EDBT*. 477–489.
- [28] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW (Companion Volume)*. 1343–1350.
- [29] Rasmus Kyng and Sushant Sachdeva. 2016. Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple. In *FOCS*. 573–582.
- [30] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <https://snap.stanford.edu/data>.
- [31] 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.
- [32] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up Distance Labeling on Graphs with Core-Periphery Properties. In *SIGMOD*. 1367–1381.
- [33] Meihao Liao, Cheng Li, Rong-Hua Li, and Guoren Wang. 2025. Efficient Index Maintenance for Effective Resistance Computation on Evolving Graphs. *Proc. ACM Manag. Data* 3, 1 (2025), 36:1–36:27.
- [34] 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. *Proc. ACM Manag. Data* 1, 2 (2023), 160:1–160:26.

- [35] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, Hongchao Qin, and Guoren Wang. 2023. Efficient Resistance Distance Computation: The Power of Landmark-based Approaches. *Proc. ACM Manag. Data* 1, 1 (2023), 68:1–68:27.
- [36] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, and Guoren Wang. 2022. Efficient Personalized PageRank Computation: A Spanning Forests Sampling Based Approach. In *SIGMOD*. 2048–2061.
- [37] Meihao Liao, Yueyang Pan, Rong-Hua Li, and Guoren Wang. 2025. Efficient Exact Resistance Distance Computation on Small-Treewidth Graphs: A Labelling Approach. *Proc. ACM Manag. Data* 3, 6 (2025), 323:1–323:27.
- [38] Meihao Liao, Bao Xing, Rong-Hua Li, and Guoren Wang. 2026. Tree+DAG: An Efficient Index for Resistance Distance Computation on Core-Periphery Graphs. Full version. <https://github.com/mhliao0516/CoreTree>
- [39] Meihao Liao, Junjie Zhou, Rong-Hua Li, Qiangqiang Dai, Hongyang Chen, and Guoren Wang. 2024. Efficient and Provable Effective Resistance Computation on Large Graphs: An Index-based Approach. *Proc. ACM Manag. Data* 2, 3 (2024), 133:1–133:27.
- [40] Yang Liu, Chuan Zhou, Shirui Pan, Jia Wu, Zhao Li, Hongyang Chen, and Peng Zhang. 2023. CurvDrop: A Ricci Curvature Based Approach to Prevent Graph Neural Networks from Over-Smoothing and Over-Squashing. In *WWW*. 221–230.
- [41] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized PageRank Estimation and Search: A Bidirectional Approach. In *WSDM*. 163–172.
- [42] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing Personalized PageRank Quickly by Exploiting Graph Structures. *Proc. VLDB Endow* 7, 12 (2014), 1023–1034.
- [43] Fragkiskos D. Malliaros, Christos Giatsidis, Apostolos N. Papadopoulos, and Michalis Vazirgiannis. 2020. The Core Decomposition of Networks: Theory, Algorithms and Applications. *VLDB J.* 29, 1 (2020), 61–92.
- [44] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *SIGMOD*. 709–724.
- [45] Pan Peng, Daniel Lopatta, Yuichi Yoshida, and Gramoz Goranci. 2021. Local Algorithms for Estimating Effective Resistance. In *KDD*. 1329–1338.
- [46] Yi Qi, Wanyue Xu, Liwang Zhu, and Zhongzhi Zhang. 2021. Real-World Networks Are Not Always Fast Mixing. *Comput. J.* 64, 2 (2021), 236–244.
- [47] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust Management for the Semantic Web. In *Proc. ISWC*. 351–368.
- [48] Ali Kemal Sinop, Lisa Fawcett, Sreenivas Gollapudi, and Kostas Kollias. 2021. Robust Routing Using Electrical Flows. In *SIGSPATIAL*. 282–292.
- [49] Daniel A. Spielman and Nikhil Srivastava. 2008. Graph sparsification by effective resistances. In *STOC*. 563–568.
- [50] Prasad Tetali. 1991. Random walks and the effective resistance of networks. *Journal of Theoretical Probability* 4, 1 (1991), 101–109.
- [51] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. 2022. Understanding over-squashing and bottlenecks on graphs via curvature. In *ICLR*.
- [52] Piet Van Mieghem, Karel Devriendt, and H Cetinay. 2017. Pseudoinverse of the Laplacian and best spreader node in a network. *Physical Review E* 96, 3 (2017), 032311.
- [53] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibowang, and Zengfeng Huang. 2020. Personalized PageRank to a Target Node, Revisited. In *KDD*. 657–667.
- [54] Sibowang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. HubPPR: Effective Indexing for Approximate Personalized PageRank. *Proc. VLDB Endow* 10, 3 (2016), 205–216.
- [55] Sibowang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *KDD*. 505–514.
- [56] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *SIGMOD*. 99–110.
- [57] David Bruce Wilson. 1996. Generating Random Spanning Trees More Quickly than the Cover Time. In *STOC*. 296–303.
- [58] Hao Wu, Junhao Gan, Zhewei Wei, and Rui Zhang. 2021. Unifying the Global and Local Approaches: An Efficient Power Iteration with Forward Push. In *SIGMOD*. 1996–2008.
- [59] Haisong Xia and Zhongzhi Zhang. 2024. Efficient Approximation of Kemeny’s Constant for Large Graphs. *Proc. ACM Manag. Data* 2, 3 (2024), 134:1–134:26.
- [60] 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.
- [61] Renchi Yang and Jing Tang. 2023. Efficient Estimation of Pairwise Effective Resistance. *Proc. ACM Manag. Data* 1, 1 (2023), 16:1–16:27.
- [62] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S. Bhowmick, Jun Zhao, and Rong-Hua Li. 2019. Efficient Estimation of Heat Kernel PageRank for Local Clustering. In *SIGMOD*. 1339–1356.

- [63] Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. 2023. Rethinking the Expressive Power of GNNs via Graph Biconnectivity. In *ICLR*.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026