TRIM: An Efficient Framework for Exact Eccentricity Computation on Large-Scale Graphs

Dian Ouyang
Guangzhou University
dian.ouyang@gzhu.edu.cn

Jiajie Lin
Guangzhou University
jiajie.lin@e.gzhu.edu.cn

Wentao Li
University of Leicester
wl226@leicester.ac.uk

Fan Zhang
Guangzhou University
zhangf@gzhu.edu.cn

Jianye Yang
Guangzhou University
jyyang@gzhu.edu.cn

Xi Luo
Guangzhou University
xluo@gzhu.edu.cn

ABSTRACT

In graph theory, the eccentricity of a vertex quantifies its centrality by measuring the maximum distance to any other vertex in the graph. This metric underpins important graph properties such as the diameter (maximum eccentricity) of the graph, which is defined by the minimum and maximum centrality values across all vertices. Due to the substantial time overhead caused by full-graph BFS traversals, researchers have focused on incorporating bounding techniques to accelerate algorithm execution. However, the state-of-the-art approach is unable to identify useless vertices and fails to terminate during the search since its bound update relies on complete traversals. In this paper, we propose a novel framework that uses vertex dominance to identify redundant vertices and introduces a new rule to ensure correct termination after skipping a vertex. In addition, we adopt a merging strategy to reduce the number of traversals. Our method achieves up to two orders of magnitude speedup in runtime compared to the state-of-the-art approach. while efficiently handling graph data at the 100-million scale.

PVLDB Reference Format:
Dian Ouyang, Jiajie Lin, Wentao Li, Fan Zhang, Jianye Yang, and Xi Luo. TRIM: An Efficient Framework for Exact Eccentricity Computation on Large-Scale Graphs. PVLDB, 19(4): 644 - 656, 2025.
doi:10.14778/3785297.3785306

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Jiajie-Lin23/TRIM.

1 INTRODUCTION

Graphs $G(V, E)$ are mathematical structures employed to model interconnected systems, where vertices (V) represent entities and edges (E) denote relationships among them. Researchers frequently focus on deriving metrics that effectively capture the structural properties of graphs. A key metric in this context is the eccentricity of a vertex v, defined as $\mathrm{ecc}(v) = \max_{u \in V} \mathrm{dist}(v, u)$, which measures the largest shortest-path distance from v to any other vertex u in the graph. This metric reflects both local centrality and global structural properties, and it captures two fundamental characteristics of the graph: diameter and radius. The radius (minimum eccentricity) and diameter (maximum eccentricity) quantify a graph's compactness and scale. Furthermore, the eccentricity distribution—the collection of eccentricities of all vertices—serves as a pivotal metric for characterizing networks [26, 40].

The concept of eccentricity is fundamental in real-world networks such as social, biological, internet, and collaboration graphs, supporting tasks like identifying peripheral vertices to estimate the worst-case response time and analyzing eccentricity distributions to evaluate synthetic models against real networks [4, 15, 27, 28, 32, 39]. Unlike average distance, the maximum shortest-path distance captures the tail behavior of networks by directly quantifying the worst-case latency, travel time, or propagation delay. Its practical value is illustrated through two primary application domains.

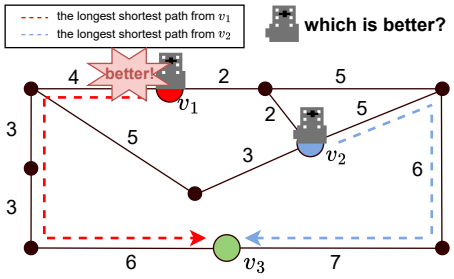


Figure 1: The real-world example of graph G^*

(1) Facility placement problem. In network infrastructure optimization, the placement of critical facilities such as hospitals, delivery centers, fire stations, or mirror servers is of central importance [3, 9, 18]. For example, suppose we aim to determine the optimal location for a new hospital. As a simplified version of the optimal location query (OLQ) problem [22], this task reduces to selecting the site that minimizes the worst-case emergency response time — i.e., the maximum shortest-path travel time from the hospital to any point in the network. As shown in Figure 1, the graph models a road network, where vertices represent candidate building sites, and edges with their weights represent roads and their associated travel time. Consider two candidate sites v_1 and v_2. To determine which offers better emergency access, we compute their eccentricities—the maximum shortest-path distances to all other vertices.

Fan Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785306

Among these vertices, v_3 is the farthest from both sites, yielding eccentricities of 16 for v_1 and 18 for v_2. Since v_1 has a smaller eccentricity, indicating a shorter worst-case response time, it is the more suitable location for building a new hospital.

(2) Matrix reordering for sparse computation. Eccentricity is widely adopted in practical graph analysis workflows, and many graph analysis libraries (e.g., NetworkX [29], JGraphT [1]) directly support computing eccentricity via built-in APIs. Beyond serving as a structural descriptor, eccentricity also plays a central role in downstream system optimizations. In particular, it is common to model a graph as a sparse matrix to solve graph-related problems using linear algebra techniques. Yet, the performance of many sparse operations critically depends on how closely the non-zero entries cluster around the main diagonal. The classical Reverse Cuthill–McKee (RCM) algorithm addresses this by renumbering vertices to reduce matrix bandwidth and improve structural compactness. Furthermore, eccentricity can guide vertex selection in RCM-based reorderings [10, 13], which serves as a core primitive in sparse solvers, graph systems, and database engines, highlighting its importance for real-world system optimization.

The previous strategies for computing eccentricity mainly fall into two categories: the BFS framework and the reference vertex framework.

The BFS framework derives from an intuitive principle: by performing BFS traversals from all vertices in the graph and calculating the maximum shortest-path distance for each vertex, the graph's eccentricity values can be systematically computed. Many related works have focused on this topic [21, 37, 38]. The core idea of this method is to initialize upper and lower bounds for each vertex and iteratively update the bounds of the remaining vertices to compute the vertex eccentricity. In each step, the algorithm selects a vertex for traversal based on specific rules and updates the bounds of other vertices using the triangle inequality. By dynamically applying pruning strategies and prioritizing vertices, the algorithm effectively eliminates redundant traversals and reduces computational overhead.

The reference vertex framework is proposed by Li [20, 21], focusing on calculating eccentricity values for all vertices. This method begins by selecting a set of high-degree vertices as reference vertices to initialize upper and lower bounds for all vertices. During traversal, it prioritizes vertices that are farthest from the reference vertices and iteratively optimizes vertex bounds using a new update formula. Notably, their study [21] introduces the PLL index to pre-store shortest-path distances between vertices, significantly boosting algorithmic efficiency. In their later study [20], they modify the update strategies to circumvent the high storage overhead of the PLL index and extend the methodology to approximate eccentricity calculations with theoretical guarantees.

Motivation. Existing approaches encounter a major bottleneck during the traversal phase, where a large number of redundant vertices fail to contribute meaningful information to the computation of vertex eccentricity. Due to the inadequate analysis of the vertex relationships, they cannot identify redundant vertices and must perform a full traversal to ensure correct algorithm termination.

Our idea. In order to address the problem of redundant vertex traversals, we propose a novel framework to reduce unnecessary computations and accelerate eccentricity computation. First, we propose a vertex dominance strategy to identify relationships among vertices and introduce a novel rule to determine termination without requiring the traversal of the skipped vertex. Moreover, we design a vertex merging strategy that replaces expensive single-vertex traversals with more efficient chain-based traversals.

Contribution. The main contributions of this paper are summarized as follows:

(1) We analyze the shortcomings of existing works and investigate their underlying causes. To address the challenge of low computational efficiency, we propose a novel framework called TRIM that skips redundant vertices while ensuring the correctness of the eccentricity computation.
(2) We analyze vertex relationships and design the largest pruning distance strategy and the core-tree strategy to identify redundant vertices. Moreover, we leverage information from traversed vertices and replace single-vertex traversals with a vertex merging strategy, both of which significantly improve execution efficiency.
(3) We extensively validate the effectiveness of our proposed method and conduct comprehensive comparisons with state-of-the-art algorithms across 20 diverse datasets. Experimental results demonstrate that our algorithm achieves up to 2–3 orders of magnitude speedup in query performance on large-scale networks.

Section 2 defines the problem and introduces relevant definitions and concepts. Section 3 reviews existing studies on eccentricity and discusses their limitations. Section 4 presents our proposed TRIM framework in detail. Section 5 provides a comprehensive evaluation of the proposed algorithm through extensive experiments. Finally, Section 6 concludes the paper with a summary of our main contributions.

2 PRELIMINARY

In this section, we formally define the eccentricity computation problem and introduce the concepts used throughout the paper.

Let $G(V, E)$ be an unweighted, undirected graph with a set of $n = |V|$ vertices and a set of $m = |E|$ edges. An edge $e(u, v) \in E$ connects two vertices $u, v \in V$. For any vertex $v \in V$, its degree $deg(v)$ is the number of one-hop neighbors of v, i.e., $deg(v) = |\{u|e(u, v) \in E\}|$. A path $p(s, t)$ from vertex s to vertex t in the graph is a sequence of $e_1(s, v_1), e_2(v_1, v_2), ..., e_{l-1}(v_{l-2}, v_{l-1}), e_l(v_{l-1}, t) \in E$. The length of the path $|p(s, t)| = l$ is the number of edges on the path. The distance $dist(s, t)$ denotes the shortest path length between s and t. The distances from a vertex to all other vertices can be computed using Breadth-First Search (BFS), which has a complexity of $O(m + n)$.

Definition 1. *Given a vertex v in a graph $G(V, E)$, the eccentricity of v is $ecc(v) = \max_{u \in V} dist(v, u)$. The farthest vertex of v is $f_v = \arg\max_{u \in V} dist(v, u)$. If multiple vertices have the same maximum distance from v, f_v can represent any of these vertices. The eccentricity distribution of G, denoted as $ED(G)$, is $ED(G) = \{ecc(v)|v \in V\}$.*

Example 1. *Figure 2 shows an example graph with 14 vertices and 18 edges. The degree of v_0 is 5, and the shortest distance between v_3 and v_9 is 4. The number above each vertex denotes its eccentricity.*

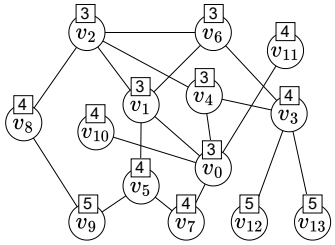


Figure 2: An example of a graph G

Problem Definition. Given a connected graph $G(V, E)$, the *exact eccentricity computation* problem aims to compute the eccentricity $ecc(v)$ for every vertex $v \in V$, where $ecc(v) = \max_{u \in V} dist(v, u)$ and $dist(v, u)$ is the shortest-path distance between v and u.

In this paper, we use tree decomposition to identify redundant vertices during the computation of exact eccentricity. We first give the definition of the tree decomposition technology, which will be used in the following section.

Definition 2. (TREE DECOMPOSITION [11]). *Given a graph $G = (V, E)$, a tree decomposition T is constructed by iteratively removing the vertices $v \in V$ from the current graph and creating tree nodes $\{X_1, X_2, \ldots, X_t\}$, where each tree node X_i is a subset of V, and the decomposition satisfies the following properties:*

(1) ***Vertex Coverage:*** $\bigcup_{X \in V(T)} X = V$;
(2) ***Edge Coverage:*** $\forall (u, v) \in E, \exists X_v \in T$ *such that both* $u \in X_v$ *and* $v \in X_v$;
(3) ***Running Intersection:*** *For any $w \in V$, the subgraph of T induced by tree vertices containing w is connected.*

Minimum-Degree Elimination Tree Decomposition[5]. We perform a series of edge addition and vertex deletion operations on the graph to construct a tree decomposition, where G_i denotes the graph structure at the $i-th$ iteration. During each iteration, we select the vertex u with the minimum degree, remove it from G_i, and update the connectivity between its neighbors: for each pair $v, v' \in N(u)$, we update the edge weight as $\min(w(v, v'), w(v, u)+w(u, v'))$ if an edge $(v, v') \in E$ exists; otherwise, we insert a new edge (v, v') with weight $w(v, u) + w(u, v')$, ensuring shortest-path consistency among all neighbors of u. This elimination loop continues until all vertices are removed. After the decomposition is complete, for each tree node X_u, let v be the vertex in $\{X_u \setminus u\}$ with the smallest value. Then, set the parent of X_u to be X_v in the tree decomposition.

Next, we introduce the cutset property of tree decomposition, which helps us to identify all cut sets within the graph structure.

Lemma 1. (Cut Set Property [34]). *Given two nodes X_i and X_j that are adjacent on the tree decomposition. A cut set is a set of vertices in a graph, and if these vertices are removed, the graph is split into two separate components. Let C_i denote the union of the trees connected to X_i, and let C_j be the union of the tree nodes connected to X_j. $X_i \cap X_j$ is a separator for all s and t pairs with $s \in C_i \setminus (X_i \cap X_j)$ and $t \in C_j \setminus (X_i \cap X_j)$.*

However, constructing a full tree decomposition for large-scale networks is computationally expensive. To this end, we introduce

Table 1: Summary of Notations

Notation	Description
$G(V, E)$	A graph with $\|V\|$ vertices and $\|E\|$ edges
$dist(s, t)$	Shortest distance between vertices s and t
$ecc(v)$	The exact eccentricity of vertex v
f_v	The farthest vertex to v
$\overline{ecc}(v), \underline{ecc}(v)$	The upper and lower bounds of $ecc(v)$
L^z	Reference queue of vertex z, sorted by descending shortest-path distance
T_G	Core-tree decomposition of graph G
X_v	The set containing vertex v and its neighboring vertices in the tree decomposition
I_{X_i}	The index vertex of a tree node X_i
LPD	The maximum pruning distance used to identify redundant vertices
C_v	The chain containing v
v_l, v_r	The left and right endpoints of the chain

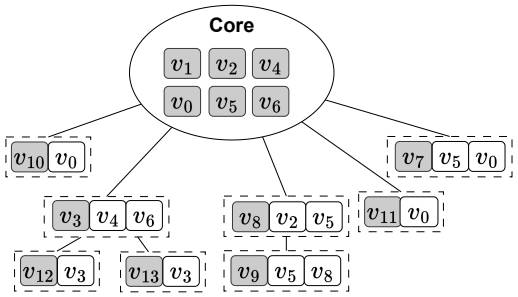


Figure 3: Core-tree structure of graph G with $d = 3$

core-tree decomposition [23], which performs decomposition on boundary vertices and significantly reduces the time overhead.

Definition 3. (CORE-TREE DECOMPOSITION)[23]. *Given an MDE-based core-tree decomposition T_G with tree nodes $\{X_1, X_2, ..., X_t\}$ and tree width d. Each tree node X_i is associated with a unique index vertex, denoted as I_{X_i}. The core-tree decomposition divides the graph into a core B^c and a forest $\mathcal{F}$.*

The elimination process is similar to tree decomposition but differs as the core-tree decomposition partitions the graph into two components based on the bound d, rather than maintaining the graph as a connected structure. It terminates when a tree node X_v satisfies $|X_v| \geq d + 1$. Each node X_v in the forest maintains a root node X_{r_v}, defined as its highest ancestor that is not in the core. The root node X_{r_v} contains an index vertex r_v, which is a member of the node itself. Based on this structure, we define the neighborhood set for each tree node X_r as $N_r = \{X_{r_v} \setminus r_v\}$.

Example 2. *Figure 3 shows the core-tree decomposition of the graph in Figure 2 with tree width $d = 3$. Each tree node $X_v \in \mathcal{F}$ has fewer than 4 vertices. Vertices 3 and 7–12 form a forest, while vertices 0, 1, 2, 4, 5, and 6 constitute the core. Each tree node X_i has a neighborhood N_r and an index vertex I_{X_i}. For example, $N_r(v_3) = N_r(v_{12}) = N_r(v_{13}) = \{v_4, v_6\}$ and the index vertex I_{X_3} is vertex 3.*

Similar to tree decomposition, the core-tree also has the cut-set property. Specifically, the neighborhood set N_r serves as the cut set that separates the vertices in a tree $T \in \mathcal{F}$ from the vertices in the set $(V \setminus (T \cup N_r))$. The following theorem formalizes this property.

Corollary 1. (Cut Set of Core-Tree). *Given a graph $G(V, E)$ and a tree $T \in \mathcal{F}$ in which every tree node $X_v \in T$ shares a common root node X_r. The neighborhood N_r of r forms the cut-set between any vertex $u \in T$ and any vertex $v \in (V \setminus (T \cup N_r))$. In other words, the shortest path from a vertex $s \in T$ to any vertex $t \in (V \setminus T)$ must contain at least one vertex in the cut-set.*

Proof. Let $s \in T$ and $t \in (V \setminus (T \cup N_r))$. We define $X_{f(r)}$ as the parent of the ancestor node X_r in the tree containing s. Consider the edge from X_r to its parent $X_{f(r)}$. If $t \notin N_r$, then according to Lemma 1, $N_r = (X_r \cap X_{f(r)})$ forms a cut-set between s and t. □

In this paper, we assume that G is a connected graph; however, this approach can be extended to a disconnected graph. Furthermore, our method applies only to undirected graphs, as it relies on the triangle-inequality property described in the IFECC [20]. This property do not generally hold in directed graphs, so the approach cannot be directly extended to directed graph networks. For the sake of clarity, the commonly used symbols are listed in Table 1.

3 PROBLEM ANALYSIS

This section abstracts the BFS-based framework from recent studies and introduces the state-of-the-art approach for the eccentricity distribution problem.

3.1 BFS-Framework with Bound Update

The eccentricity of a vertex can be computed using BFS. This approach has a total time complexity of $O(n(n+m))$, since it performs a BFS from each vertex, making it inefficient for large-scale graphs. To improve efficiency, many existing methods leverage upper and lower bounds of vertex eccentricity. Specifically, each vertex $v \in V$ is assigned a lower bound $\underline{ecc}(v)$ and an upper bound $\overline{ecc}(v)$ on its eccentricity. When a BFS is performed from a source vertex u, the exact eccentricity of u is determined, and the bounds of other vertices can be updated using Lemma 2. This helps reduce the number of BFS operations needed throughout the calculation.

Lemma 2. (Triangle Inequality Bound Update [38]). *Given a vertex u, once a BFS from vertex u is complete, we can use the following inequalities to update the bounds of other vertices $v \in V$.*

$$ecc(v) \leq dist(u, v) + ecc(u) \tag{1}$$

$$ecc(v) \geq \max\{dist(u, v), ecc(u) - dist(u, v)\} \tag{2}$$

Upon executing a BFS from a vertex $u \neq v$, the upper bound $\overline{ecc}(v)$ can be further updated to $dist(u, v) + ecc(u)$, while the lower bound $\underline{ecc}(v)$ is updated to $\max\{\text{dist}(u, v), \text{ecc}(u) - \text{dist}(u, v)\}$. When the lower and upper bounds of v are equal, $ecc(v)$ is determined without performing a BFS from v.

BFS-framework. Previous works based on the BFS framework [2, 8, 12, 14, 35–38] generally follow a sequence of key steps summarized as follows:

(1) For each vertex v in the graph, initialize the eccentricity upper bound $\overline{ecc}(v) = +\infty$ and the lower bound $\underline{ecc}(v) = 0$;
(2) Select a source vertex set $S \subseteq V$. The selection of S can be performed based on a certain criterion.
(3) Select a vertex $s \in S$, perform a BFS traversal rooted at s, and update the eccentricity bounds of all reachable vertices using Lemma 2.
(4) If the upper and lower bounds of every vertex $v \in V$ are equal, the algorithm terminates; if not, go to Step 3.

3.2 Reference-Framework with Bound Update

To reduce the number of vertex traversals and update operations, Li et al. [20, 21] proposed a traversal strategy guided by reference vertices.

The state-of-the-art method IFECC [20] builds upon PLLECC [21]. They share a core idea: constructing a Farthest-First Vertex Order (FFO) queue using selected landmarks to guide vertex traversal, and iteratively updating bounds for all vertices. However, IFECC identifies that a significant portion of the distance information stored in PLLECC's PLL index remains unused during query execution, resulting in considerable space overhead for large-scale graphs. To address this, IFECC adopts a traversal-based strategy during computation, avoiding full index construction and improving efficiency. In the following, we introduce this core idea in more detail.

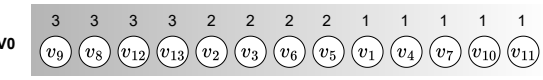


Figure 4: Farthest-First Vertex Order

Farthest-First vertex Order (FFO). IFECC adopts a reference queue called the Farthest-First Vertex Order (FFO), introduced in PLLECC [21], which prioritizes vertex traversal by sorting vertices in decreasing order of their distances from a reference vertex.

(1) Select a subset of vertices with the highest degree as reference vertices and add them to the set Z.
(2) For each reference vertex $z \in Z$, perform a BFS traversal starting from z, and construct a queue L^z, ordered by decreasing distance to z: $L^z = \langle V_1^z, V_2^z, \ldots, V_n^z \rangle$.

Example 3. *Given the reference vertex v_0 from the Figure 2, the vertex order based on its shortest distances to v_0 is used to construct the queue $L_z = \{v_9^z, v_8^z, \ldots, v_{11}^z\}$, as shown in Figure 4, where vertices are sorted in descending order of their shortest distances to v_0. The number above each vertex represents its shortest distance from v_0.*

Lemma 3. (Update Bound [21]). *Given a reference vertex z, consider the case when r is the farthest unvisited vertex in the FFO of z. The bounds of other vertices $v \in \{V \setminus z\}$ can be updated as follows:*

$$\underline{ecc}(v) = \max\left\{\underline{ecc}(v), dist(v, r)\right\} \tag{3}$$

$$\overline{ecc}(v) = \min\left\{\overline{ecc}(v), \max\left\{\underline{ecc}(v), dist(v, z) + dist(z, r)\right\}\right\} \tag{4}$$

According to the proof of IFECC, we select one reference vertex and then compute the exact eccentricities of all vertices as follows:

(1) Select one reference vertex z with the highest degree and perform BFS from z.

(2) Construct the queue L^z, ordered in decreasing shortest distance from z as: $L^z = \langle V_1^z, V_2^z, ..., V_n^z \rangle$. Compute the eccentricity $ecc(z)$ of z.
(3) Use the equations in Lemma 2 to update the upper and lower bounds of each vertex $v \in \{V \setminus \{z\}\}$.
(4) Select a vertex r and perform a BFS following the order of L^z. Use the distance results to compute $\text{dist}(v, r)$ and $\text{dist}(v, z) + \text{dist}(z, r)$ according to Lemma 3, which are used to update the lower and upper bounds of other vertices.
(5) If all vertices $v \in V$ have $\underline{ecc}(v) = \overline{ecc}(v)$, terminate the algorithm; otherwise, goto step 4.

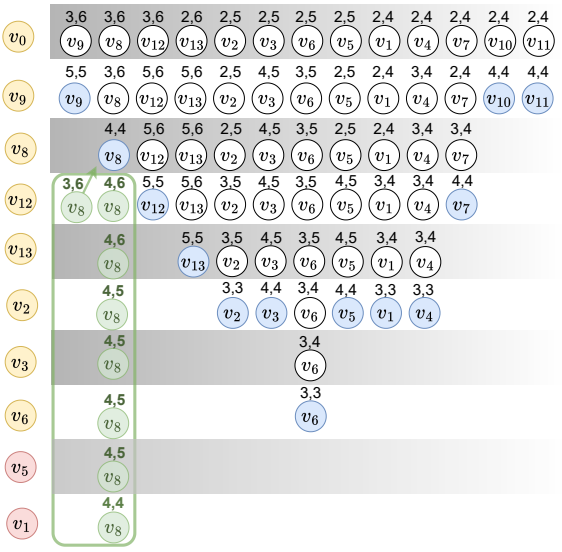


Figure 5: The illustration of IFECC

Example 4. *Figure 5 illustrates the process on the graph from Figure 2, where the yellow vertices are used to perform the standard IFECC algorithm, and the green vertices are reserved for subsequent explanation. IFECC first selects v_0 as the initial reference vertex and uses Lemma 2 to initialize the bounds of other vertices. After the BFS from v_9, the lower bound of v_{10} is updated to* $\max\{\underline{ecc}(v_{10}), dist(v_9, v_{10})\} = 4$, *and the upper bound is updated to* $\min\{\overline{ecc}(v_{10}), \max\{\underline{ecc}(v_{10}), dist(v_{10}, v_0) + dist(v_0, v_9)\}\} = 4$. *This process continues until vertex v_6, where all vertices satisfy $\overline{ecc}(v) = \underline{ecc}(v)$. The total number of traversals is 8.*

Limitation. The IFECC framework suffers from two limitations: (1) it lacks a redundancy identification strategy to prune redundant vertices; (2) each vertex must perform a complete traversal to ensure the correct termination of the algorithm. In other words, the framework lacks a strategy to identify redundant vertices. Here, redundant vertices refer to those whose traversals can be substituted by the traversals of other vertices. Nevertheless, skipping any redundant vertex is not permitted, since the termination condition $\overline{ecc}(v) = \underline{ecc}(v)$ for all vertices $v \in V$ relies on complete bound updates, which depend on complete traversal.

Example 5. *Figure 5 illustrates the limitations of IFECC. We focus on the changes of vertex v_8. The green vertices indicate the situation when IFECC skips the traversal of v_8, showing its lower and upper bounds. If we skip the traversal of vertex v_8, its bounds will not be updated. Therefore, two additional red vertices $v_5, v_1 \in L^z$ have been traversed, which increases the total number of traversals from 8 to 9.*

Our solution. To address the above limitation, our core idea is to use a vertex dominance strategy to identify redundant vertices and perform traversals and update operations on the useful vertices. Additionally, for each traversal, we apply a rule that ensures correctness and allows for efficient checking of whether all vertices have been correctly computed, even if some vertices are skipped. Furthermore, we design a vertex merging strategy that uses two Dijkstra traversals to avoid traversing the majority of the vertices. Together, these methods effectively eliminate redundant computations. The implementation details are presented in the later sections.

3.3 APSP Problem

The all-pairs shortest paths (APSP) problem is a classical task in graph theory, which aims to compute the shortest path distance between every pair of vertices in a graph. The most fundamental solution is the Floyd–Warshall algorithm, which runs in $O(n^3)$ time and requires $O(n^2)$ space. To improve efficiency, Wei Peng [33] proposed a fast APSP algorithm for complex networks. The key idea is to prioritize high-degree vertices as hubs, compute their shortest paths first, and use these results as upper bounds when processing other sources, thereby achieving an empirical complexity of $O(n^{2.4})$. Pereira Junior [16] proposed the FB-APSP algorithm, which employs a bidirectional scanning strategy to improve efficiency. FB-APSP uses both outgoing and incoming edges to reduce redundant edge exploration.

After the all-pairs shortest paths (APSP) computation is completed, the eccentricity of any vertex can be obtained by performing an $O(n)$ scan over the distance matrix. Therefore, the eccentricity problem is essentially a subproblem of APSP. However, APSP computes far more distance information than eccentricity requires. Moreover, it requires $O(n^2)$ storage, which becomes infeasible for large-scale graphs with millions of vertices. As a result, we exclude APSP-based methods from our experimental evaluation.

3.4 Other Centrality Variants

Recent studies have proposed several extended centrality metrics. Steiner-k-eccentricity measures the maximum size of a minimal Steiner tree spanning k terminals, including a given vertex [24]. Resistance eccentricity captures the maximum effective resistance distance between vertices [25]. Other centrality metrics include closeness centrality [31], which reflects a vertex's average distance to all others, and betweenness centrality [30] measures how often a vertex lies on shortest paths.

4 A NOVEL FRAMEWORK FOR ED PROBLEM

In this section, we introduce a novel algorithm called Traversal with Redundant Vertex Identification and Vertex Merging Strategy (TRIM). This framework builds on the bound update strategy of IFECC and extends it with our pruning and early termination techniques. As implied by the name, the key idea is to accelerate computation by eliminating traversals through pruning strategies.

4.1 Framework

To ensure the algorithm can terminate correctly while skipping redundant vertices, we propose a novel termination rule. To this end, we analyze how bound updates impact the computation results.

Upper Bound Analysis. Lemma 3 indicates that (1) the upper bound $\overline{ecc}(v) = dist(v,z) + dist(z,r)$ decreases as $\text{dist}(z,r)$ decreases, since $\text{dist}(v,z)$ is fixed; and (2) the lower bound converges to its exact value no later than the upper bound. The following lemma formalizes the second observation.

Lemma 4. *For each vertex $v \in V$, the lower bound converges to its exact value no later than the upper bound according to Lemma 3.*

Proof. We prove this lemma by contradiction. Assume that the lower bound converges to its exact value after the upper bound. Then, for a vertex $v \in V$, there exist two vertices $r, r' \in L^z$ such that vertex r satisfies $dist(v,r) < ecc(v)$ and $dist(v,z)+dist(z,r) \leq ecc(v)$, while vertex r' satisfies $dist(v,r') = ecc(v)$, $dist(z,r) \geq dist(z,r')$, and r' is traversed after r. We have $dist(v,z)+dist(z,r) \geq dist(v,z)+dist(z,r') \geq dist(v,r') = ecc(v)$. In this case, $dist(v,z)+dist(z,r) = ecc(v)$ must hold. However, if this equality holds, r must be the farthest vertex from v, implying $dist(v,r) = ecc(v)$, contradicting our assumption $dist(v,r) < ecc(v)$. This proves the lemma. □

Based on the above analysis, we conclude that if a vertex $v \in V$ fails to update the lower bound of any other vertex, there exists another vertex $v' \in V$ that can substitute v to update the lower bound correctly. Since the correct update occurs at v' rather than v and, by Lemma 4, the traversal of v can be skipped, and v is regarded as redundant. We now formally define the redundant vertex.

Definition 4 (Redundant Vertex). *A vertex $v \in L^z$ is redundant if the inequality $\underline{ecc}(u) \geq dist(v,u)$ holds for every vertex $u \in V$ during the update operation performed by vertex v. In this case, processing v cannot increase any lower bound $\underline{ecc}(\cdot)$.*

However, skipping a redundant vertex will leave its upper bound outdated, potentially preventing the algorithm from terminating even when all lower bounds are accurate. To address this, we propose a rule to determine the termination condition of the algorithm.

Lemma 5. *During the execution of bounds update, vertices $v \in V$ are processed sequentially according to the fixed FFO queue. Suppose the currently selected vertex v satisfies:*

$$v = \arg\max_{u \in V} \left\{ \overline{ecc}(u) - \underline{ecc}(u) \right\}.$$

Let $\underline{ecc}(v)$ denote the current lower bound of $v \in V$. When processing next vertex $r \in L^z$, if $\max\{dist(v,z) + dist(z,r) - \underline{ecc}(v)\} = 0$, then all bounds have converged, and the algorithm can terminate.

Proof. Let v be the vertex with the largest bound gap after an update operation from vertex $r' \in L^z$. When the next vertex $r \in L^z$ is selected and processed from the FFO queue, the value $\text{dist}(v,z) + \text{dist}(z,r)$ gives the updated upper bound of v. The condition $\max\{\text{dist}(v,z) + \text{dist}(z,r) - \underline{ecc}(v)\} = 0$ implies that v's bounds have converged. Since all vertices undergo the same decrement in their upper bounds and v has the largest gap, all bounds have converged, and the algorithm can terminate with a correct result. □

Algorithm 1: TRIM FRAMEWORK (G)

Input: Graph $G(V,E)$
Output: Eccentricity distribution $ED(G)$

1 $L^z, vis_t, vis_e, H, \overline{ecc}, \underline{ecc}, T_G(B^c, \mathcal{F}) \leftarrow$ Initialization (G); //Alg.2
2 $\mathcal{M} \leftarrow$ VertexMerging(G); //Alg.4
3 **foreach** *vertex* $v \in L^z$ **do**
4 $\mathcal{A} \leftarrow$ ComputeDominance($G, \mathcal{F}, vis_t, \underline{ecc}, v$); //Alg.3
5 **if** *LPDJudge*($G, \mathcal{A}, \mathcal{M}, vis_e, H, v, \underline{ecc}, \overline{ecc}$) = *true* **then**
6 **return** $\underline{ecc}$;
7 **return** ecc;

Algorithm 2: Initialization (G)

Input: Graph $G(V,E)$
Output: $L^z, vis_t, vis_e, H, \overline{ecc}, \underline{ecc}, T_G(B^c, \mathcal{F})$

1 Reference vertex $z \leftarrow$ the vertex with the highest degree;
2 $vis_e(v) \leftarrow false$ for each vertex $v \in V$;
3 $vis_t(T) \leftarrow false$ for each tree $T \in \mathcal{F}$;
4 Compute core-tree decomposition [23] $T_G(B^c, \mathcal{F})$;
5 Compute $\text{ecc}(z)$ and the FFO $L^z = \langle v_1^z, v_2^z, \ldots, v_n^z \rangle$ of z;
6 **foreach** *vertex* $v \in L^z$ **do**
7 $\underline{\text{ecc}}(v) \leftarrow \max\{\text{ecc}(v), \text{dist}(v,z), \text{ecc}(z) - \text{dist}(v,z)\}$;
8 $\overline{\text{ecc}}(v) \leftarrow \min\{\text{dist}(v,z) + \text{ecc}(z)\}$;
9 $H(v) \leftarrow \overline{\text{ecc}}(v) - \underline{\text{ecc}}(v)$ for each vertex $v \in V$;
10 **return** $L^z, vis_t, vis_e, H, \overline{ecc}, \underline{ecc}, T_G$;

If we adopt IFECC and skip multiple redundant vertices, the upper bounds for all vertices may remain outdated. With Lemma 5, we can precisely determine the termination condition during traversal.

Lower Bound Analysis. For any vertex $v \in V$, Lemma 3 indicates that the lower bound $\underline{ecc}(v)$ is determined by the maximum shortest distance from visited vertices to v. Therefore, a BFS traversal with update operations should be valid if the current vertex can successfully update the lower bounds of other vertices.

Based on the bound analysis, we present a framework outlining its key components. We compute the exact eccentricities of all vertices using a boundary-based strategy and prune redundant vertices (as defined in Definition 4) to accelerate computation. To enable pruning, we first compute the core-tree decomposition (Algorithm 2), which captures structural relationships among vertices and is used in Algorithm 3 to determine their vertex relationships. We further apply a novel strategy (Algorithm 4) to reduce the number of vertex traversals. Finally, the termination condition is defined by Lemma 5 based on the analysis of Lemma 4.

Algorithm. Algorithm 1 begins by constructing the search queue L^z, initializing the related data structures (vis_t, vis_e, H, $\overline{ecc}$, and $\underline{ecc}$), and building the core-tree structure (Line 1). Next, a vertex merging mechanism is applied to merge traversals of multiple vertices (Line 2), followed by an update operation from each vertex $v \in L^z$ (Line 3). Then, we compute the dominance relationship to identify redundant vertices (Line 4), and we apply a novel update rule to determine the termination condition (Lines 5–6). Finally, the algorithm returns the eccentricity array, which contains the exact eccentricity of each vertex in the graph (Line 7).

4.2 Initialization

Since our main contribution lies in traversal pruning strategies, we adopt a similar initialization strategy used in IFECC, selecting the highest-degree vertex as the reference and constructing an FFO queue. In contrast, we construct a core-tree structure to identify redundant vertices. The details are presented in Algorithm 2.

Algorithm. Algorithm 2 first selects a vertex with the highest degree as the reference vertex, sets all entries in vis_e,vis_t to false, and constructs a core-tree structure [23] (Lines 1-4). Then, we perform BFS from z, compute $ecc(z)$, and construct the FFO of z according to the shortest distance to z for all vertices $v \in V$ (Line 5). Furthermore, we update the bounds for all vertices in L^z using Lemma 2 and initialize the array H (Lines 6-9). Finally, we return all the arrays as well as the tree structure T_G (Line 10).

4.3 Vertex Dominance Strategy

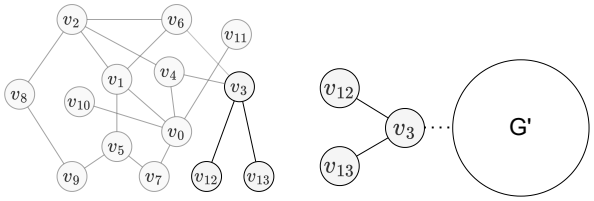
After constructing the search queue L^z, we obtain a traversal order of the vertices. Then, we can perform BFS and use Lemma 3 to update the bounds of each vertex $v \in L^z$ to get their exact eccentricity. However, strictly following the queue L^z leads to redundant computations. The inherent reason lies in the dominance relationships among the vertices $v \in L^z$. We now introduce this relationship and give a rule on how to use it to identify the redundant vertices.

Definition 5. (Vertex Dominance Strategy). *Given a graph G and a vertex set S of size t, if there exist vertices $r, r' \notin S$ such that:*

$$(dist(r, v_1) \geq dist(r', v_1)) \wedge ... \wedge ... \wedge (dist(r, v_t) \geq dist(r', v_t))$$

Then we say that r dominates r' in S, denoted as $r \succ r'$.

Vertex Domination Pruning Rule: Given a graph $G(V, E)$, if there exist vertices $u, v \in L^z$ such that $u \succ v$ holds in S, then after using u to update the bounds of vertices in $\{V \setminus S\}$, v is regarded as a redundant vertex and can be safely pruned according to Lemma 4 and Definition 4, since v cannot update any vertex lower bound.



(a) A Graph G **(b) A simplified graph G' and three vertices v_3, v_{12}, v_{13}**

Figure 6: The example of vertex dominance.

Example 6. *Figure 6 (a) shows the original graph G, and Figure 6 (b) shows the simplified graph G', where $G' = \{G \setminus \{v_3, v_{12}, v_{13}\}\}$. Strictly following the IFECC approach requires visiting v_{12}, v_{13} and v_3 sequentially, which requires 3 full BFS traversals. However, since v_3 forms the cut set between v_{12}, v_{13} and other vertices $v \in G'$, each vertex $v \in G'$ has the same shortest distance to v_{12} and v_{13}, which is strictly greater than its distance to v_3. Therefore $v_{12} \succ \{v_3, v_{13}\}$ in the set G'. With minimal cost, we update the bounds of $\{v_3, v_{12}, v_{13}\}$ from v_3 and v_{13}, after which v_3, v_{13} can be safely pruned. The BFS traversals can be reduced from 3 to 1 by skipping two redundant vertices.*

We employ two strategies to handle vertex dominance. The first strategy, performed before traversing $r \in L^z$, uses core-tree decomposition to identify dominance relationships among vertices in the same tree and prune redundant vertices. The intuition behind the first strategy is that there exists a cut set composed of the parent vertex and its neighbors, which enables distance comparisons among the vertices within the same subtree. The second is an adaptive dominance judgment strategy applied during traversal, which dynamically determines whether the current vertex is dominated by previously visited vertices since they have no impact on lower bound updates. We introduce these two strategies in detail below.

4.4 Core-Tree and Vertex Dominance Strategy

The details of the core-tree decomposition are illustrated in Section 2. Given a tree T, the shortest distances from vertex $v \in T$ to vertex $u \in \{V \setminus T\}$ are comparable according to Corollary 1. According to Definition 2, given two vertices v, v' that belong to the same tree T, the set $S = \{V \setminus T\}$ is determined. If $dist(v, u) > dist(v', u)$ holds for all $u \in \{V \setminus T\}$, indicating that v dominates v' in the set $S = \{V \setminus T\}$. Since the dominance relation holds in $\{V \setminus T\}$, we need to update the bounds of the vertices in the tree T.

Lemma 6. (Core-Tree-Based Dominance Pruning Strategy). *Given a graph $G(V, E)$ and a tree T. If two vertices $u, v \in T$ satisfy $u \succ v$ in the set $\{V \setminus T\}$, then v can be pruned after computing its distances to all vertices in the tree T via a Dijkstra starting from v, which only involves vertices in the cut set N_r and the tree. The resulting distances are used to update the lower bounds of the vertices in T.*

Proof. Let $T \subseteq \mathcal{F}$ be a subtree of the core-tree. Corollary 1 implies that the neighbors N_r of the root vertex r of the root node X_r form a cut set C that separates any vertex $s \in T$ from any vertex $t \in (V \setminus (T \cup N_r))$. Consequently, the shortest path from s to t must pass through a vertex in N_r, and it satisfies $\text{dist}(s, t) = \min_{r \in N_r} \{\text{dist}(s, r) + \text{dist}(r, t)\}$. For two vertices $u, v \in T$ where $u \succ v$, the shortest path segment $dist(r, t)$ is the same, and it holds that $dist(u, r) \geq dist(v, r)$. Therefore, $dist(u, t) \geq dist(v, t)$, implying that u can substitute v as farthest vertex for vertex $t \in \{V \setminus T\}$. □

If the current vertex satisfies $u \succ v$ in tree T, then v cannot provide effective lower bound updates to vertices $x \in \{V \setminus T\}$. Therefore, after updating the lower bounds of all vertices in T, v can be safely pruned according to Lemma 6.

Algorithm. Algorithm 3 outlines the Vertex Dominance Computation. If tree T has been visited before vertex v, it is not recomputed (Lines 1-3). Initially, all entries of the array $\mathcal{A}$ are set to true (Line 4). Each vertex $x \in \{I_{X_i} \mid X_i \in T\}$ computes its shortest distances to the cut set N_r (Line 5). For any $v_i, v_j \in \{I_{X_i} \mid X_i \in T\}$, if the shortest-path distance from v_i to every vertex in N_r is greater than or equal to that from v_j, we set $\mathcal{A}(v_j)$ as false, indicating that $v_i \succ v_j$, and thus the bounds updated from v_j are computed using a small-range Dijkstra (Lines 6-9). The final active array $\mathcal{A}$ serves as the dominance indicator for vertices in tree T (Line 10).

Theorem 2. *The time complexity of Algorithm 3 is $O(d \cdot n'^2 + n' \log n')$, which is bounded by $O(d \cdot n^2 + n \log n)$ since $n' \leq n$.*

Proof. Let d be the tree width, n' the number of vertices, and k the average cost of computing the distance from a vertex to the

Algorithm 3: ComputeDominance $(G, \mathcal{F}, vis_t, \underline{ecc}, v)$

Input: Graph G, a forest $\mathcal{F}$, an array vis_t, an array $\underline{ecc}$, and a vertex v
Output: Domination array $\mathcal{A}$

```
T ← the tree contains vertex v;
if vis_t(T) = true then return A;
vis_t(T) ← true;
A(v) ← true for each index vertex v of X_v ∈ T;
Compute the shortest distances from vertex x ∈ {I_{X_i} | X_i ∈ T} to
  the set N_r;
for v_i, v_j ∈ {I_{X_i} | X_i ∈ T} do
    if ∀u ∈ N_r, s.t.dist(v_i, u) > dist(v_j, u)∨
     (dist(v_i, u) = dist(v_j, u) ∧ (i < j)) then
        Update the lower bounds ecc for x ∈ {I_{X_i} | X_i ∈ T};
        A(v_j) ← false;
return A;
```

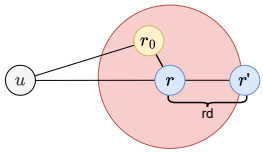


Figure 7: The illustration of *LPD*

cut set N_r. Computing distances for all n' vertices costs $O(n'k)$, which is negligible compared to $O(dn'^2)$ since $k \ll dn'$. Sorting distance lists costs $O(n' \log n')$, and pairwise dominance comparisons in k dimensions cost $O(dn'^2)$. Thus, the total time complexity of Algorithm 3 is $O(dn'^2 + n' \log n')$. □

Example 7. *Figure 3 illustrates that vertices v_{12} and v_3 belong to the same tree rooted at v_3 (from the graph in Figure 2). The neighborhood of v_3 forms the cut set $N_r = \{v_4, v_6\}$. Since v_{12} has a greater shortest path distance to N_r than v_3, we conclude that $v_{12} \succ v_3$, so v_3 can be safely pruned during traversal.*

4.5 Largest Pruning Distance

We observe that each visited vertex $r' \in L^z$ stores its shortest-path distances to all other vertices. When selecting the current vertex $r \in L^z$, its distances to other vertices remain unknown, making it impossible to form the comparison set S in Definition 5. To address this, we propose an adaptive dominance judgment strategy that dynamically determines dominance relationships during traversal without requiring an explicit definition of S.

We solve the vertex dominance relationship based on an intuitive idea: evaluate the effectiveness of the current vertex $r_0 \in L^z$ using distance information from visited vertices $r \in L^z$ during its traversal. Suppose a BFS is performed from a vertex $r \in L^z$. If we cannot update the lower bound of a vertex v, then there exists another vertex r' satisfying $dist(r', v) \geq dist(r, v)$. Without loss of generality, assume that two vertices r and r', where $dist(u, r') \geq dist(u, r)$.

Example 8. *Figure 7 shows the lower bound update for a vertex $u \in V$. Let the red circle be centered at r with radius $rd = dist(u, r') - dist(u, r)$. To update $\underline{ecc}(u)$ from $r_0 \in L^z$, we check if $dist(r, r_0) \leq rd$. By the triangle inequality among vertices $\{u, r, r_0\}$, this gives: $dist(u, r_0) \leq dist(r, r_0) + dist(u, r) \leq dist(u, r')$. It implies that $dist(u, r_0)$ is not greater than the lower bound $\underline{ecc}(v)$. Therefore, the vertex r_0 cannot update the lower bound of the vertex u.*

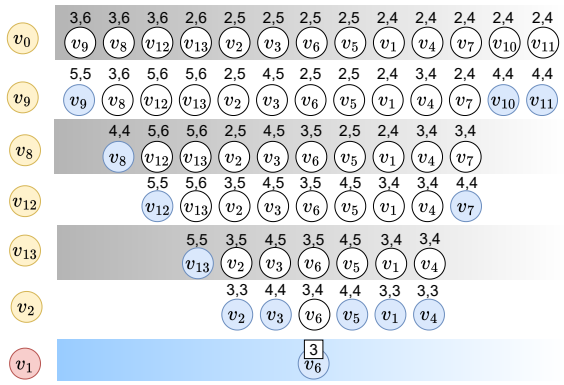


Figure 8: The example of *LPD*

Based on the above observation, we provide a formal definition to determine the pruning distance.

Definition 6 (Largest Pruning distance). *Given a vertex $r \in L^z$, the largest pruning distance (denoted as LPD) is defined as:*

$$LPD(r) = \min\{\underline{ecc}(v) - dist(r, v) \mid v \in V\}$$

where $\underline{ecc}(v)$ denotes the lower bound of vertex v, and $dist(r, v)$ denotes the shortest path distance between vertex r and vertex v.

A zero value of *LPD* implies that some vertices regard the current vertex $r \in L^z$ as the farthest vertex, while a non-zero LPD indicates that all such vertices select a visited vertex $r' \in L^z$ instead.

Lemma 7. *We can safely prune the current vertex $x \in L^z$ during BFS upon encountering a vertex $r \in L^z$ that has completed a global BFS and calculated its LPD, satisfying the following condition:*

$$LPD(r) \geq dist(x, r).$$

Proof. We prove this by contradiction. Let r be the encountered vertex during BFS from x. We assume that vertex x cannot be pruned by r under the condition $LPD(r) \geq \text{dist}(x, r)$. It implies that x can update at least one lower bound for a vertex $v \in V$. According to Definition 6, for every vertex $v \in V$, there exists a vertex r' such that: $dist(v, r') - dist(r, v) \geq LPD(r)$. By assumption, $LPD(r) \geq dist(x, r)$, which leads to the inequality $dist(v, r') \geq dist(x, r) + dist(r, v) \geq dist(v, x)$. This implies that for every vertex $v \in V$ there exists a vertex r' whose shortest distance to v is greater than or equal to the distance from x to v. Therefore, vertex x cannot update any lower bound, which contradicts our initial assumption. □

Example 9. *Figure 8 demonstrates the pruning rule from Lemma 7 and the update operation from Lemma 3. After v_2 performs BFS, we get $LPD(v_2) = 2$. When the next vertex v_3 performs BFS and reaches v_2, since $dist(v_2, v_3) = LPD(v_2)$, traversal from v_3 can be pruned. After pruning v_1, the termination condition $dist(v, z) + dist(z, r) - \underline{ecc}(v) = 0$ is met, so the algorithm terminates, taking each vertex's lower bound as its exact eccentricity. In total, 7 BFS traversals are performed.*

Algorithm 4: VertexMerging (G)

Input: Graph $G = (V, E)$
Output: Vertex Merging structure

1 $visited(v) \leftarrow$ false for every vertex v;
2 Set $\mathcal{M} \leftarrow \emptyset$;
3 **foreach** $v \in V$ **do**
4 **if** $\deg(v) \neq 2$ ***and*** $visited(v) = false$ **then**
5 $visited(v) \leftarrow true$;
6 **if** $\forall\, u \in N(v) :\ \deg(u) > 2$ **then**
7 Construct the chain C_v including vertex v;
8 $\mathcal{M} \leftarrow \mathcal{M} \cup C_v$;
9 **foreach** $x \in N_v$ **do**
10 Construct the chain C_x from x;
11 $\mathcal{M} \leftarrow \mathcal{M} \cup C_x$;
12 $\forall\, w \in C_x,\ visited(w) \leftarrow$ true;
13 **return** $\mathcal{M}$;

4.6 Vertex Merging Strategy

The original traversal strategy relies on a single-vertex BFS, with each iteration updating bounds solely from the perspective of one reference vertex. This raises the question: Can one traversal cover multiple vertices? We observe that vertices in real-world graphs often reside in chain-like structures. Motivated by this, we propose a vertex merging strategy to eliminate redundant traversals by treating such structures as unified units for update.

Definition 7. (Chain Structure). *Given a vertex v with degree 2. The chain with the length len containing v : $Chain_v = \{v_1, v_2, ..., v_{len}\}$ satisfies the following conditions:*

- *The degree of the left and right endpoints of the chain must be either 1 or greater than 2.*
- *The degree of all other vertices in the chain equals 2.*

Algorithm. The process of merging vertices is shown in Algorithm 4. We initialize the array $visited$ and $\mathcal{M}$ (Lines 1-2). For each unvisited vertex v, if $\deg(v) \neq 2$, we mark v as a potential endpoint (Lines 4-5). If all neighbors $u \in N(v)$ satisfy $\deg(u) > 2$, then v is recorded as a single-vertex chain (Lines 6-8); otherwise, we traverse its neighbors to construct the corresponding chain (Lines 9–12).

Theorem 3. *The time complexity of the Vertex Merging construction is $O(n + m)$.*

Proof. Algorithm 4 constructs the chain structure in $(n + m)$ time. It iterates over all n vertices in $O(n)$ time (Line 3), and each edge is scanned once. For each unvisited vertex with degree $\neq 2$, we either insert it as a singleton chain (Lines 6–7), or construct a chain starting from its neighbors with degree $\neq 2$ and add it to $\mathcal{M}$ (Lines 8–11). The total time complexity is therefore $O(n + m)$. □

Lemma 8. *Given a vertex u and a chain with length len, after completing BFS of left endpoint v_l and right endpoint v_r, the farthest vertex v can be determined by $\frac{dist(u,v_l)+dist(u,v_r)+len}{2}$. The farthest distance can be calculated using the following equation:*

- *If u, v are **on the same chain**, the farthest distance is:*

$$\max\{dist(u, v_l),\ dist(u, v_r)\}$$

Algorithm 5: LPDJudge ($G, \mathcal{A}, \mathcal{M}, vis_e, H, v, \underline{ecc}, \overline{ecc}$)

Input: Graph G, array $\mathcal{A}$, merging structure $\mathcal{M}$, array vis_e, array H, vertex v, arrays $\underline{ecc}$ and $\overline{ecc}$ (both updated by reference)
Output: Termination flag: true if bounds have converged

1 $C_v \leftarrow$ the chain that contains v in $\mathcal{M}$;
2 $v_l, v_r \leftarrow$ left and right endpoints of C_v;
3 Queue $q \leftarrow \emptyset$;
4 Sets $S_{v_l}, S_{v_r} \leftarrow \emptyset$;
5 **if** $\mathcal{A}(v) = true$ **then**
6 **if** $|C_v| = 1$ *and* $vis_e(v) = false$ **then**
7 $q \leftarrow q \cup \{v, v, 0\}$;
8 **else**
9 **foreach** $u \in \{v_l, v_r\}$ *where* $vis_e(u) = false$ **do**
10 $q \leftarrow q \cup \{u, u, 0\}$;
11 **while** $q \neq \emptyset$ **do**
12 $\langle u, \text{source}, d\rangle \leftarrow q$.extractMin();
13 **if** $u \in S_{source}$ **then** continue ;
14 $S_{source} \leftarrow S_{source} \cup u$;
15 $H(u) \leftarrow H(u) - \max(0, d - \underline{ecc}(u))$;
16 $\underline{ecc}(u) \leftarrow \max\{\underline{ecc}(u), d\}$;
17 $LPD(v_{source}) \leftarrow \min\{LPD(v_{source}), \underline{ecc}(u) - d\}$;
18 **if** $|C_v| = 1$ **then**
19 **if** $vis_e(u) = true$ *and* $d \leq LPD(u)$ **then** break ;
20 **else**
21 **if** $vis_e(u) = true$ *and* $u \in S_{v_l}, S_{v_r}$ **then**
22 $\mathcal{D} \leftarrow$ the farthest distance from u to vertex on C_v;
23 **if** $\mathcal{D} \leq LPD(u)$ **then** break ;
24 **for** *each neighbor* $x \in N_u$ **do**
25 $dist\{source, x\} \leftarrow d + 1$;
26 $q \leftarrow q \cup \{x,\ \text{source},\ dist\{source, x\}\}$;
27 **if** $q = \emptyset$ **then** $vis_e(v_l), vis_e(v_r) \leftarrow true$;
28 **while** $H \neq \emptyset$ **do**
29 $x \leftarrow argmin(H(x))$;
30 $H(x) \leftarrow H(x) - (dist(v', z) - dist(v, z))$ // v' is the last reference vertex;
31 **if** $H(x) \leq 0$ **then**
32 $H \leftarrow H \setminus x$;
33 **else return** *false*;
34 **return** *true*

- *If u, v are **not on the same chain**, the farthest distance is:*

$$\min\{dist(u, v_l) + dist(v_l, v),\ dist(u, v_r) + dist(v_r, v)\}$$

Proof. Let v be the farthest vertex from u on the chain.

If u and v lie on the same chain, then v must be either v_l or v_r, since no internal vertex can be further than endpoints.

If u, v are not on the same chain, any path connecting an external vertex u to the chain must traverse either v_l or v_r (as these form the chain's cut set). By the maximality of $dist(u, v)$, the distances via both endpoints must satisfy:

$$\text{dist}(u, v_l) + \text{dist}(v_l, v) = \text{dist}(u, v_r) + \text{dist}(v_r, v). \qquad \text{(a)}$$

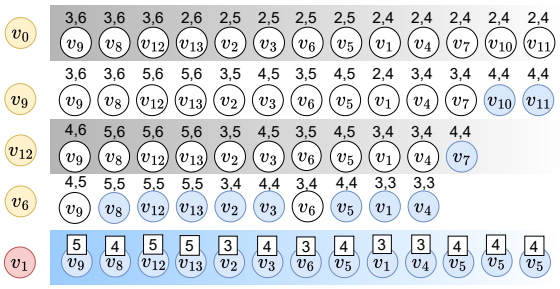


Figure 9: The illustration of TRIM

Since the chain's length is $len = dist(v_l, v_r)$, we have:

$$\text{dist}(v_l, v) + \text{dist}(v_r, v) = \text{len}. \tag{b}$$

Substituting $\text{dist}(v_r, v) = \text{len} - \text{dist}(v_l, v)$ into (a):

$$\text{dist}(v_l, v) = (\text{dist}(u, v_r) + \text{len} - \text{dist}(u, v_l)) \ / \ 2. \tag{c}$$

Substituting (c) into $\text{dist}(u, v) = \text{dist}(u, v_l) + \text{dist}(v_l, v)$:

$$\text{dist}(u, v) = (\text{dist}(u, v_r) + \text{dist}(u, v_l) + \text{len}) \ / \ 2. \tag{d}$$

The farthest vertex on the chain is determined by its distance from v_l. To find the farthest vertex from an external vertex u, we compute the critical position at distance $\frac{\text{dist}(u,v_r)+\text{dist}(u,v_l)+\text{len}}{2}$ from v_l, and compare adjacent vertices to identify the farthest vertex from u. □

We design an LPD-based pruning strategy with vertex merging strategy to reduce redundant traversals. Each vertex $v \in L^z$ is regarded as part of a chain, and two independent Dijkstra searches are initiated from the chain's endpoints. If a vertex u that has completed a full traversal is encountered during traversal, we compute its farthest shortest distance d to the chain using Lemma 8 and compare it with $\text{LPD}(u)$ to decide whether the traversal can be pruned. Algorithm 5 outlines the exploration and termination process.

Algorithm. After initializing (Lines 1-4), if $\mathcal{A}(v) \neq true$, we prune vertex v (Line 5); otherwise, we enqueue u with distance 0 for a single-vertex chain. For multi-vertex chains, only unvisited endpoints v_l or v_r are enqueued, while visited vertices reuse recorded distances (Lines 6-10). The main loop extracts the vertex with the smallest distance from the queue and checks whether it has already been dequeued (Lines 12-14). Then, $H(u)$ is updated using the lower-bound difference, while $\underline{ecc}(u)$ and $LPD(v_{source})$ are updated simultaneously (Lines 15-17). For single-vertex chains, the algorithm checks if the current vertex has performed BFS and satisfies $d \leq \text{LPD}(u)$. For chains with more than one vertex, if the current vertex has performed BFS and both endpoints are visited, we check whether the distance $\mathcal{D}$ satisfies $\mathcal{D} \leq \text{LPD}(u)$ (Lines 18-23). The neighbors of the current vertex are then enqueued with updated distances. If q becomes empty, $vis_e(v_l)$ and $vis_e(v_r)$ are set to true (Lines 24-27). Then, we repeatedly extract the minimum value from H and update the values in the set H. If all extracted values are zero, the algorithm can terminate according to Lemma 5 and return false. Otherwise, the algorithm returns true (Lines 28–34).

Theorem 4. *The time complexity of Algorithm 5 is $O(n^* + m^*)$. n^* and m^* are the number of vertices and edges where Algorithm 5 traverses by, which is bounded by $O(n+m)$ since $n^* \leq n$ and $m^* \leq m$.*

Proof. Algorithm 5 performs BFS from the endpoints of the merging structure and may terminate early if $d \geq \text{LPD}(v)$. It costs $O(n^* + m^*)$. On line 29, we have to find the minimum vertex in a set. In an unweighted graph, the upper and lower bound is bounded by the eccentricity value. We can use a method like bucket sorting to store the gap between the upper and lower bounds. We can use $O(1)$ time to find x. Hence, the total time complexity of Algorithm 5 is $O(n^* + m^*)$. □

Example 10. *Figure 9 illustrates the process of the TRIM. The total number of full BFS traversals is reduced to 6. Specifically, vertex v_9 lies in a chain of length 4 and requires only two traversals.*

To demonstrate the extensibility, we adapt TRIM to weighted graphs. We can replace each BFS with Dijkstra. Specifically, Algorithms 1, 3 and 4 remain unchanged from the unweighted case. In Algorithm 2 (line 5), $ecc(z)$ is computed by Dijkstra instead of BFS. In Algorithm 5 (line 25), update rule is changed to $dist\{source, x\} \leftarrow d + w[u][x]$, where $w[u][x]$. With a priority queue, the time complexity of Algorithm 5 becomes $(n^* + m^*) \log n$.

4.7 Overall Time Complexity

The time complexity of Algorithm 1 is summarized as follows:

Theorem 5. *The overall time complexity of our proposed Algorithm 1 is $O(d \cdot n'^2 + (n^* + m^*) \cdot n^{\#} + n + m)$. Since $n^{\#} \leq n' \leq n$, $n^* \leq n$ and $m^* \leq m$, the overall time complexity is bounded by $O(d \cdot n^2 + (n+m) \cdot n) = O(d \cdot n^2 + nm)$.*

Proof. Recall that d is the tree width, n' is the number of vertices in the decomposition tree, $n^{\#}$ is the number of non-dominated vertices, and n^*, m^* are the vertices and edges processed during traversals. The time complexity includes $O(n)$ for initialization (Algorithm 2) and $O(n+m)$ for merging structures (Algorithm 4). For each $v \in L^z$, vertex dominance (Algorithm 3) costs $O(d \cdot n'^2 + n' \log n')$, and Line 5 costs $O(n^* + m^*) \cdot n^{\#}$. The total complexity is $O(2n + m + z \cdot (2d \cdot n'^2 + n' \log n' + 2n + m))$. □

5 EXPERIMENT

In this section, we first introduce the experimental setup and provide an overview of the algorithms evaluated. Then, we present a runtime comparison with the state-of-the-art IFECC algorithm, followed by an analysis of the tree width d and ablation studies.

Algorithms. We compare the following approaches:

- IFECC: The state-of-the-art algorithm for computing exact eccentricity with one reference vertex.
- TRIM-LTM: Our full framework with three strategies enabled: the Largest Pruning Distance (L), the Core-Tree Pruning (T), and the Vertex Merging (M).
- TRIM-LT, TRIM-LM, TRIM-TM: Variants with two strategies enabled, one removed.
- TRIM-L, TRIM-T, TRIM-M: Variants with only a single strategy enabled.

All algorithms are implemented in C++ and compiled with GNU GCC 9.4.0 with -O3 optimization. All experiments are conducted on a machine with two Intel Xeon Platinum 8124M CPUs (@3.00GHz,

Table 2: Dataset Description and Comparison of IFECC and TRIM

	n	m	Type	Eccentricity	Average_ecc	IFECC(sec)	TRIM(sec)	Speedup rate	Traversals (IFECC)	Traversals (TRIM)
NY	264,346	733,846	Road	1,652,818	1,149,594	2390.0	32.3	74.07	57,611	109
DBLP	317,080	1,049,866	Social	23	15	17.2	4.0	4.29	854	220
COL	435,666	1,057,066	Road	9,350,254	6,719,744	25604.7	254.9	100.44	314,384	526
FLA	1,070,376	2,712,798	Road	11,993,938	8,388,811	-	164.4	-	-	121
Youtube	1,134,890	2,987,624	Social	24	15	2.9	1.9	1.56	39	7
ITALY	6,686,493	7,013,978	Infrastructure	13,108	8,745	84465.1	995.8	84.82	147,975	55
E-USA	3,598,623	8,778,114	Road	20,568,329	13,508,712	-	262.7	-	-	79
WIKI	2,388,953	9,313,364	Social	11	8	9122.4	1699.0	5.37	84,915	20,408
Skitter	1,694,616	11,094,209	Internet	31	21	8.0	4.6	1.74	46	14
W-USA	6,262,104	15,248,146	Road	36,108,898	26,603,177	-	18852.3	-	-	3,731
FLIC	1,624,992	15,476,835	Social	24	15	6.3	3.3	1.94	31	6
Patents	3,764,117	16,511,740	Citation	26	18	27197.2	9733.5	2.79	45,971	13,768
BAIDU	2,107,689	16,996,139	Web	20	12	18.5	9.7	1.90	60	18
Hollywood09	1,069,126	56,306,653	Collaboration	12	8	1466.1	666.5	2.20	4,464	1,680
Sinaweibo	58,655,849	261,321,018	Social	8	6	43615.4	2609.4	16.71	5,513	371
ARBA	22,634,275	552,231,867	Web	47	29	1122.0	343.3	3.27	814	167
IT	41,290,577	1,027,474,895	Web	45	26	3231.5	910.0	3.55	704	163
Twitter (MPI)	52,515,193	1,614,062,827	Social	18	13	2312.4	1592.6	1.45	69	44
FRIE	65,608,366	1,806,067,135	Social	37	22	4237.3	1397.2	3.03	89	18
UK07	104,288,749	3,293,805,080	Web	112	76	517.4	178.1	2.91	39	2

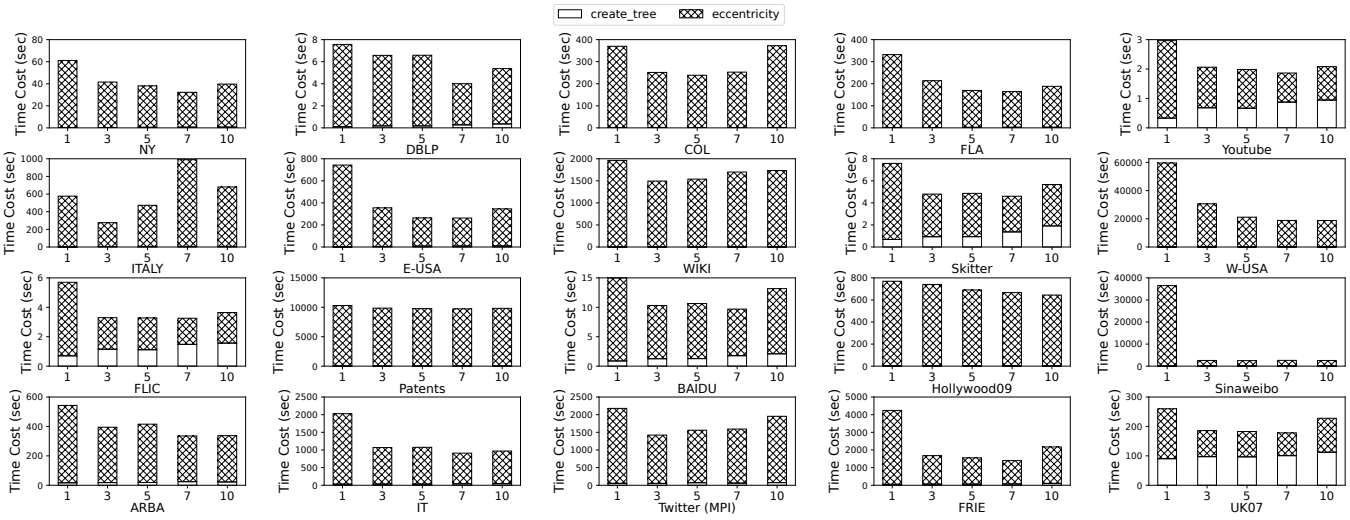


Figure 10: The effect of tree width d

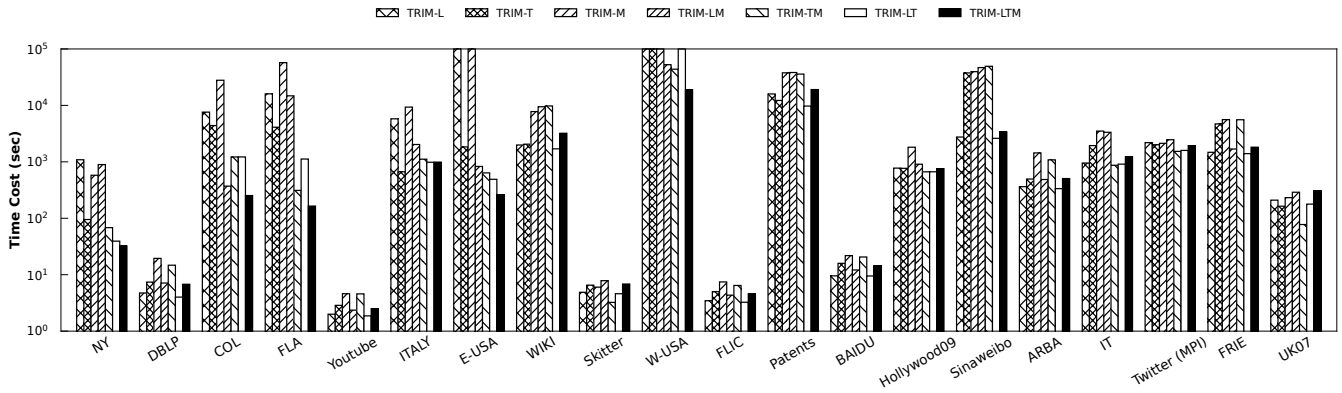


Figure 11: Runtime Comparison across different strategies

18 cores each, 72 threads total) and 512GB main memory, running Linux (Ubuntu 20.04.6 LTS, Kernel 6.11.0-26-generic, 64-bit).

We employ TRIM-LT for non-road networks and TRIM-LTM for road networks, with a detailed ablation study in the subsequent sections. The IFECC column shows the runtime of our compared method. The Speedup rate column indicates the ratio of IFECC's runtime to that of the TRIM-LTM/TRIM-LT column, thereby reflecting the performance improvement achieved by our approach.

Datasets. We use 20 publicly available, large-scale networks of various types, as shown in Table 2, including social, internet, web, road, infrastructure, citation, and collaboration networks. All datasets are publicly available from the Stanford Large Network Dataset Collection[1] [17] , the Laboratory for Web Algorithms [2] [6, 7], and the Koblenz Network Collection [3] [19].

Exp 1: Computation time. We compare the proposed approach with the state-of-the-art IFECC in terms of its time consumption. To ensure fairness, we adopt the same configuration as IFECC, which employs one reference vertex as recommended in the study [20]. We use TRIM-M on non-road networks and TRIM on road networks.

Our approach achieves an average speedup of 18.4× over IFECC, consistently outperforming the state-of-the-art method across all datasets. For non-road networks, it achieves up to 85.4× on Italy and 16.7× on Sinaweibo. For road networks, it achieves up to two orders of magnitude improvement in runtime over IFECC on the COL dataset. Our approach completes all datasets in 12 hours, while IFECC fails to process some datasets within 24 hours. These demonstrate the high efficiency and strong scalability of our method.

Exp 2: Varying tree width. Figure 10 shows the time consumption on 16 datasets when selecting different tree widths d. We incorporate the time spent on tree decomposition into the overall computation. We observe a non-monotonic trend in execution time as tree width increases from 1 to 10: the runtime first decreases from 1 to 7, and then increases from 7 to 10. This is due to two considerations. On one hand, larger tree width increases decomposition overhead but enables more effective pruning by assigning more vertices to branches. On the other hand, a larger tree width increases the computational cost when identifying redundant vertices.

Specifically, the average runtimes for configurations with $d = 1, 3, 5$, and 10 are 2.38×, 1.11×, 1.06×, and 1.16× those of $d = 7$, respectively. Notably, $d = 3$ performs better on Twitter(MPI), WIKI, and ITALY. $d = 5$ performs better on COL and Sinaweibo, while $d = 10$ performs better on W-USA and Hollywood09. So, we select $d = 7$ as the default tree decomposition width in our method.

Exp 3: Ablation Study. Figure 11 presents a complete ablation study covering all seven TRIM variants across all datasets. On road networks, TRIM-LTM achieves the highest efficiency overall. Removing any single component leads to noticeable efficiency degradation, while adding an additional strategy on top of one or two existing strategies generally accelerates the algorithm execution. On non-road networks, TRIM-LT achieves the best performance on most datasets. In these networks, removing any single strategy from TRIM-LT generally reduces efficiency on most datasets, whereas adding the vertex merging component (M) to TRIM-L, TRIM-T, or TRIM-LT often incurs additional time overhead. Therefore, the vertex merging strategy is most suitable for road networks.

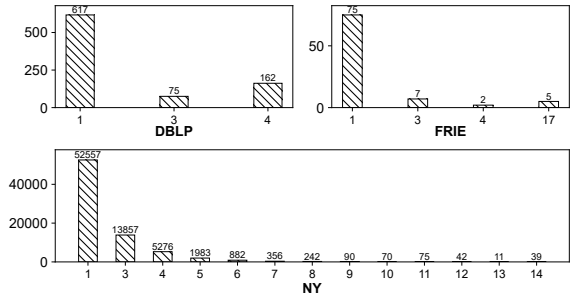


Figure 12: Chain Length Comparison of different datasets

To investigate why TRIM-LTM underperforms TRIM-LT on non-road networks, we analyze the chain lengths of the vertices that perform a traversal across three datasets: DBLP, FRIE, and NY. As shown in Figure 12, DBLP and FRIE networks involve fewer than 1,000 vertices and most vertices are located in short chains. In contrast, road networks exhibit longer chains and far more vertices that perform a traversal. For example, NY reaches chain lengths of 14, with many vertices in long chains. These longer chains enhance the benefits of vertex merging by reducing redundant traversals. In non-road networks, however, the limited chain lengths and small search space diminish pruning effectiveness. These observations are consistent with the average eccentricity and the "Traversals (IFECC)" column in Table 2, further confirming that the shorter chain lengths and smaller search space in non-road networks fundamentally limit the effectiveness of the vertex merging strategy.

6 CONCLUSION

In this paper, we study the problem of exact eccentricity computation in large-scale networks. We propose a novel TRIM framework that selectively traverses and updates only the effective vertices, significantly reducing the overhead caused by redundant vertex visits. Compared to the state-of-the-art IFECC algorithm, our method achieves substantial improvements in computational efficiency across all datasets. The experimental results demonstrate that our approach achieves up to two orders of magnitude speedup on large-scale networks. In addition, it remains effective on road networks where IFECC encounters limitations, demonstrating its scalability and practical superiority.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 62502105, No. 62472114, No. 62572137, No. 62572140, No. 62536007, No. 62302417), the National Key R&D Program of China (No. 2022YFB2702300, No. 2022YFB3104100), the Guangdong Basic and Applied Basic Research Foundation (No. 2023A1515110592, No. 2025A1515011716, No. 2024A1515011501, No. 2023A1515012603), the Major Key Project of PCL (No. PCL2024A05) and the Science and Technology Guangzhou Education Department Foundation (No. 2023KQNCX057).

[1] http://snap.stanford.edu/data/
[2] http://law.di.unimi.it
[3] http://konect.cc/networks/

REFERENCES

[1] 2025. GraphMeasurer (JGraphT) — Vertex eccentricity, radius, diameter. https://jgrapht.org/javadoc-1.2.0/org/jgrapht/alg/shortestpath/GraphMeasurer.html.

[2] Takuya Akiba, Yoichi Iwata, and Yuki Kawata. 2015. An exact algorithm for diameters of large real directed graphs. In *International Symposium on Experimental Algorithms*. Springer, 56–67.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Diameter of the world-wide web. *nature* 401, 6749 (1999), 130–131.

[4] Albert-Laszlo Barabâsi, Hawoong Jeong, Zoltan Néda, Erzsebet Ravasz, Andras Schubert, and Tamas Vicsek. 2002. Evolution of the social network of scientific collaborations. *Physica A: Statistical mechanics and its applications* 311, 3-4 (2002), 590–614.

[5] Anne Berry and Pinar Heggernes. 2003. The Minimum Degree Heuristic and the Minimal. *lecture notes in computerence* (2003).

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[8] Shiri Chechik, Daniel H Larkin, Liam Roditty, Grant Schoenebeck, Robert E Tarjan, and Virginia Vassilevska Williams. 2014. Better approximation algorithms for the graph diameter. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1041–1052.

[9] Ertugrul N Ciftcioglu, Kevin S Chan, Ananthram Swami, Derya H Cansever, and Prithwish Basu. 2015. Topology control for time-varying contested environments. In *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE, 1397–1402.

[10] SL GONZAGA DE OLIVEIRA and ABREU AAAM. 2019. An Experimental Analysis of Three Pseudo-peripheral Vertex Finders in conjunction with the Reverse Cuthill-McKee Method for Bandwidth Reduction. *TEMA (São Carlos)* 20, 3 (2019), 497–507.

[11] Rudolf Halin. 1976. S-functions for graphs. *Journal of geometry* 8 (1976), 171–186.

[12] Keith Henderson. 2011. *Opex: Optimized eccentricity computation in graphs*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[13] JiaJun Hou, HongJie Liu, and ShengXin Zhu. 2024. RCM++: Reverse Cuthill-McKee ordering with Bi-Criteria Node Finder. *arXiv preprint arXiv:2409.04171* (2024).

[14] Keita Iwabuchi, Geoffrey Sanders, Keith Henderson, and Roger Pearce. 2018. Computing exact vertex eccentricity on massive-scale distributed graphs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 257–267.

[15] Shudong Jin and Azer Bestavros. 2006. Small-world characteristics of internet topologies and implications on multicast scaling. *Computer Networks* 50, 5 (2006), 648–666.

[16] Dyson Pereira Junior and Emilio Carlos Gomes Wille. 2018. FB-APSP: A new efficient algorithm for computing all-pairs shortest-paths. *Journal of Network and Computer Applications* 121 (2018), 33–43.

[17] Leskovec Jure. 2014. SNAP Datasets: Stanford large network dataset collection. *Retrieved December 2021 from http://snap. stanford. edu/data* (2014).

[18] Matjaž Krnc, Jean-Sébastien Sereni, Riste Škrekovski, and Zelealem B Yilma. 2020. Eccentricity of networks with structural constraints. *Discussiones Mathematicae Graph Theory* 40, 4 (2020), 1141–1162.

[19] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[20] Wentao Li, Miao Qiao, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2022. On scalable computation of graph eccentricities. In *Proceedings of the 2022 International Conference on Management of Data*. 904–916.

[21] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2018. Exacting eccentricity for small-world networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 785–796.

[22] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Eccentricities on small-world networks. *The VLDB Journal* 28, 5 (2019), 765–792.

[23] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[24] Xingfu Li, Guihai Yu, Sandi Klavžar, Jie Hu, and Bo Li. 2021. The Steiner k-eccentricity on trees. *Theoretical Computer Science* 889 (2021), 182–188.

[25] Zenan Lu, Xiaotian Zhou, Ahad N Zehmakan, and Zhongzhi Zhang. 2024. Resistance Eccentricity in Graphs: Distribution, Computation and Optimization. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4113–4126.

[26] Clémence Magnien, Matthieu Latapy, and Michel Habib. 2009. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics (JEA)* 13 (2009), 1–10.

[27] Damien Magoni and Jean Jacques Pansiot. 2001. Analysis of the autonomous system network topology. *ACM SIGCOMM Computer Communication Review* 31, 3 (2001), 26–37.

[28] Damien Magoni and Jean-Jacques Pansiot. 2002. Analysis and comparison of Internet topology generators. In *International Conference on Research in Networking*. Springer, 364–375.

[29] NetworkX Developers. [n.d.]. eccentricity — NetworkX Documentation. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.distance_measures.eccentricity.html. Accessed: YYYY-MM-DD.

[30] Mark EJ Newman. 2005. A measure of betweenness centrality based on random walks. *Social networks* 27, 1 (2005), 39–54.

[31] Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. 2008. Ranking of closeness centrality for large-scale social networks. In *International workshop on frontiers in algorithmics*. Springer, 186–195.

[32] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using graph theory to analyze biological networks. *BioData mining* 4 (2011), 1–27.

[33] Wei Peng, Xiaofeng Hu, Feng Zhao, and Jinshu Su. 2012. A fast algorithm to find all-pairs shortest paths in complex networks. *Procedia Computer Science* 9 (2012), 557–566.

[34] Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms* (1986).

[35] Liam Roditty and Virginia Vassilevska Williams. 2013. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 515–524.

[36] Julian Shun. 2015. An evaluation of parallel eccentricity estimation algorithms on undirected real-world graphs. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1095–1104.

[37] Frank W Takes and Walter A Kosters. 2011. Determining the diameter of small world networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. 1191–1196.

[38] Frank W Takes and Walter A Kosters. 2013. Computing the eccentricity distribution of large graphs. *Algorithms* 6 (2013), 100–118.

[39] Stanley Wasserman and Katherine Faust. 1994. Social network analysis: Methods and applications. (1994).

[40] Douglas Brent West et al. 2001. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River.