The VLDB Journal (2019) 28:765–792
https://doi.org/10.1007/s00778-019-00566-9

REGULAR PAPER

Eccentricities on small-world networks

Wentao Li[1] · **Miao Qiao**[2] · **Lu Qin**[1] · **Ying Zhang**[1] · **Lijun Chang**[3] · **Xuemin Lin**[4]

Received: 13 September 2018 / Revised: 7 August 2019 / Accepted: 17 August 2019 / Published online: 28 August 2019

Abstract

This paper focuses on the efficiency issue of computing and maintaining the eccentricity distribution on a large and perhaps dynamic small-world network. Eccentricity distribution evaluates the importance of each node in a graph, providing a node ranking for graph analytics; moreover, it is the key to the computation of two fundamental graph measurements, diameter, and radius. Existing eccentricity computation algorithms are not scalable enough to handle real large networks unless approximation is introduced. Such an approximation, however, leads to a prominent relative error on small-world networks whose diameters are notably short. Our solution optimizes existing eccentricity computation algorithms on their bottlenecks—one-node eccentricity computation and the upper/lower bounds update—based on a line of original insights; it also provides the first algorithm on maintaining the eccentricities of a dynamic graph without recomputing the eccentricity distribution upon each edge update. On real large small-world networks, our approach outperforms the state-of-the-art eccentricity computation approach by up to three orders of magnitude and our maintenance algorithm outperforms the recomputation baseline (recompute using our superior eccentricity computation approach) by up to two orders of magnitude, as demonstrated by our extensive evaluation.

Keywords Eccentricity · Small-world networks · Centrality measures · Dynamic graphs

1 Introduction

Shortest distances characterize the pair-wise relationships among nodes in a graph. Given a graph with a vertex set and an edge set, the shortest distance $dist(u, v)$ between two nodes u, v is defined as the minimized length of a path from u to v. The largest– shortest distance from one node u to another node of the graph defines u's **eccentricity**. The two extremes—the maximization and the minimization—of eccentricities over all nodes in a graph are, respectively, the values of the two fundamental [31] features, diameter, and radius, of the entire graph. The computation and maintenance of these two features of a large and perhaps dynamic graph can be inevitable for some graph analytical tasks. Besides, the eccentricity also measures the centrality of a node in the graph. The eccentricity distribution, the eccentricities of all the nodes in a graph, thus helps in identifying important nodes in a graph, which could be influential people in a social network, critical nodes in an epidemic contact network, or important sites in a Web graph. With all the applications listed above, an efficient approach for computing the eccentricity distribution of a graph is highly demanded.

Unfortunately, the eccentricity distribution can be expensive to compute especially on big graphs. The state-of-the-art algorithm for computing solely the diameter requires $\Omega\left(\frac{n^2}{\log n}\right)$ time (see [7,32] and the reference therein) where n denotes the total number of nodes in the graph (n is 1.32

✉ Wentao Li
wentao.li@student.uts.edu.au

Miao Qiao
miao.qiao@auckland.ac.nz

Lu Qin
lu.qin@uts.edu.au

Ying Zhang
ying.zhang@uts.edu.au

Lijun Chang
lijun.chang@sydney.edu.au

Xuemin Lin
lxue@cse.unsw.edu.au

1 CAI, FEIT, University of Technology Sydney, Sydney, Australia

2 University of Auckland, Auckland, New Zealand

3 The University of Sydney, Sydney, Australia

4 The University of New South Wales, Sydney, Australia

billion on Facebook[1]). An immense n renders significant efficiency issue on diameter computation as well as eccentricity computation while introducing approximation is a natural solution.

To reduce the computation cost, approximate algorithms for estimating the eccentricity $\widetilde{ecc}(v)$ of each node v with an error bound [8,23] have been proposed. Their efficiency gain, however, can hardly be extended to error-intolerant small-world networks.

Small-world networks, a term first proposed by Watts and Strogatz [30], describe a group of graphs that feature a highly clustered topology and short path length. The phenomenon of short path length has been observed much earlier in the book of "Six Degrees of Separation" [12] and has been confirmed later on recent data of biological networks, neural networks, collaboration networks, communication networks, and social networks. For example,[2] Slashdot, a social network, has a diameter of 11, while wiki-talk, a communication network, has a diameter of 9. On unweighted networks, any additive positive error $\delta = |\widetilde{ecc}(\cdot) - ecc(\cdot)|$ will have $\delta \geq 1$. Note that $\delta = 1$ is already a high error compared to the short radius/diameter of a small-world network: if the radius $r = 5$, then $\frac{1}{r} = 20\%$. In other words, small-world networks are intolerant to errors on the eccentricity distribution.

Though on small-world networks, a high empirical accuracy can be achieved by the approximate algorithms without error bounds [26,27], an efficient exact algorithm is still highly demanded. For the eccentricity of a node, the estimation with a chance of a high relative error (due to the small graph diameter and a lack of error bound) can hardly be trusted and doubtlessly engaged. Furthermore, on billion-scale graphs, it would be even challenging to evaluate the accuracy of an approximate algorithm without the exact eccentricities.

The state-of-the-art exact eccentricity computation (see [27] and the references therein) follows the same paradigm, which i) associates each node with eccentricity bounds, an upper bound and a lower bound on its eccentricity; ii) for each node v, if the upper and lower bounds of v do not meet, compute the eccentricity $ecc(v)$ using a Breadth-First Search (BFS) and then *update* the bounds *globally* for all other nodes. The performance is, therefore, largely dependent on the node order of v traversed in step ii). Our solution significantly improves the efficiency by revising the paradigm based on a spectrum of insights. We identify the bottleneck of the existing approaches—the exhaustive BFS and global update—and then speed up the convergence of the upper and lower eccentricity bounds.

When it comes to real graphs with dynamic nature, a graph becomes a sequence of snapshots evolving over time. To detect anomaly in the graph, tracking diameter and eccentricities as prominent features of the graph structure is desirable [11,17]. Recomputing the eccentricity upon each update can be obviously exhaustive. We find that only a portion of nodes may have their eccentricities affected by an edge update. Furthermore, we propose an algorithm that can identify these affected nodes and then adjust their eccentricities efficiently.

Our contributions are summarized below.

– We facilitate an early termination in computing the eccentricity of a node v by i) non-trivially reversing (approximately) the node visiting order of BFS at a low cost and ii) optimizing the computation of $ecc(v)$ based on the eccentricity bounds of v instead of using BFS to compute $ecc(v)$ in $\Omega(m)$ time.
– We show that, in updating the eccentricity bounds, it suffices for our algorithm to update only a connected area of amortized $O(d)$ nodes while achieving the same effect as performing a global update. Here, d denotes the graph diameter—a small integer for a small-world network.
– We reduce the cost of maintaining the eccentricities upon an edge insertion/deletion by first scoping the possibly affected nodes and then facilitating a more targeted eccentricity recomputation—the eccentricities of the other affected nodes can be determined directly based on the rules that we identified.
– Empirical studies show that our approach outperforms the state of the art by up to three orders of magnitude on eccentricity computation. In particular, our approach is the only one that completed the computation within 8 h on all graphs. Besides, for eccentricity maintenance on dynamic graphs, our approach gains up to two orders of magnitude speedup over the baseline approach.

The paper is organized as follows. Section 2 formally introduces the eccentricity computation and maintenance problems. Section 3 describes our algorithm in computing the eccentricity. Section 4 devises an efficient eccentricity maintenance algorithm. Section 5 summarizes the related work. Section 6 demonstrates the experimental results, while Sect. 7 concludes the paper.

2 Preliminaries

Section 2.1 defines the problems of eccentricity computation and eccentricity maintenance which Sects. 3 and 4 shall study. Section 2.2 introduces the state-of-the-art solution to pair-wise shortest distance queries, the building block of eccentricity computation.

[1] https://newsroom.fb.com/company-info/.

[2] https://snap.stanford.edu/data/index.html.

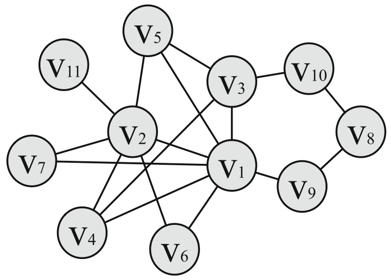

Fig. 1 Example graph G

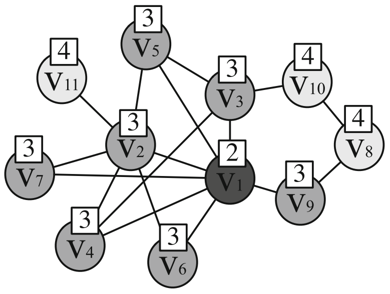

Fig. 2 Eccentricity of nodes in G

2.1 Problem definition

This paper considers the eccentricity on an unweighted and undirected graph. Let $G(V, E)$ be a graph with a vertex set V and an edge set E. An edge $e(u, v)$ in E connects two nodes u, v in V. Denote by $n = |V|$ the total number of nodes and $m = |E|$ the total number of edges in G. The degree $deg(v)$ for a node v is the number of neighbors adjacent to v, that is, $deg(v) = |\{u, e(u, v) \in E\}|$.

Given two nodes s and t in V, a *path* $p(s, t)$ from s to t is a sequence of distinct nodes $\langle u_0 = s, u_1, \dots, u_k = t\rangle$ with neighboring nodes connected by edges, that is, $(u_{i-1}, u_i) \in E$, for $\forall i \in [1, k]$. The *length* $|p(s, t)|$ of a path is the total number of edges on the path. The *shortest distance* $dist(s, t)$ between s and t is the length of the shortest path from s to t. The shortest distances on V hold the *triangle inequality*, that is, for three nodes $s, u, t \in V, dist(s, t) \leq dist(s, u) + dist(u, t)$.

Example 1 Figure 1 shows a running example of graph G with 11 nodes and 17 edges. The shortest path from v_7 to v_8 is $\langle v_7, v_1, v_9, v_8\rangle$ with $dist(v_7, v_8) = 3$. For nodes v_7, v_8, v_3, the triangle inequality $dist(v_7, v_8) \leq dist(v_3, v_7) + dist(v_3, v_8)$ holds since $dist(v_7, v_3) = 2$ and $dist(v_3, v_8) = 2$.

Definition 1 (*Eccentricity*) Given a node u of a graph $G(V, E)$, the eccentricity of u is defined as

$$ecc(u) = \max_{v\in V} dist(u, v).$$

Among all the eccentricities of nodes in G, the maximum eccentricity is defined as the diameter d, that is, $d = \max_{v\in V} ecc(v)$; the minimum eccentricity is defined as the radius, that is, $r = \min_{v\in V} ecc(v)$.

Example 2 Figure 2 labels, on the graph G of the running example, the eccentricity of each node. The color grayscale indicates, for each node, the eccentricity value: a node with a darker color means that it has a smaller eccentricity. For example, the eccentricity of node v_1 is calculated as $ecc(v_1) = \max_{v\in V} dist(v_1, v) = 2$, and $ecc(v_{11}) = \max_{v\in V} dist(v_{11}, v) = 4$. Intuitively, a node at the center has a smaller eccentricity than the node at the border.

Problem 1 (*Eccentricity computation*) Given a graph $G(V, E)$, compute the eccentricity distribution, namely the eccentricity $ecc(u)$ for all the nodes $u \in V$.

If a graph is disconnected, that is, there exist two nodes u, v such that there is no path from u to v, then the eccentricities of all the nodes in V become $+\infty$ which is trivial. Therefore, we assume that $G(V, E)$ is connected throughout the paper. In particular, this assumption also applies to the dynamic scenario of eccentricity maintenance defined as below.

Problem 2 (*Eccentricity maintenance*) Let $G(V, E)$ be a graph with the eccentricity distribution given; namely, the eccentricity $ecc(u)$ on G for each node $u \in V$ is given. Let e be an edge on V, that is, $e \in V \times V$. Denote by G' the graph of G after a graph update—an edge insertion with $G' = (V, E\cup\{e\})$ or an edge deletion with $G' = (V, E\backslash\{e\})$. Denote by $ecc'(v)$ the eccentricity of node v in G', for $\forall v \in V$. The aim is to compute $ecc'(v)$, for $\forall v \in V$ whose $ecc'(v) \neq ecc(u)$.

In the eccentricity maintenance, we focus on the edge insertions that do not change the vertex set V and the edge deletions that do not disconnect the graph. In other words, the eccentricity of a node will not increase after an edge addition and will not decrease after an edge deletion. If an update goes beyond these cases, we simply recompute all the eccentricities.

The building block of efficient eccentricity computation is pair-wise shortest distance computation. The state-of-the-art solution is introduced in Sect. 2.2.

2.2 Pair-wise shortest distance

On unweighted graphs, the pair-wise shortest distance problem (PWSD) reports, given two nodes u and v, the distance between u and v. It can be trivially resolved by performing a Breadth-First Search (BFS) from node u online. The linear online computation can take seconds for answering a single

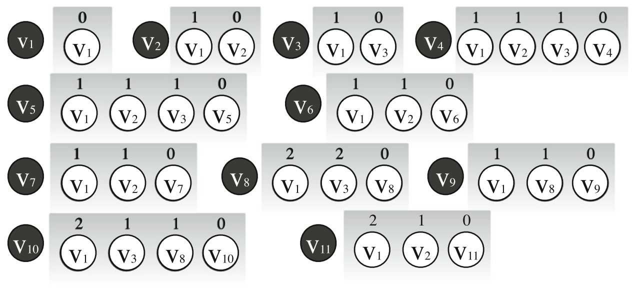

Fig. 3 Pruned landmark labeling for all nodes in G

Algorithm 1: BoundEcc

```
Input: Graph $G(V, E)$
Output: $ecc(u)$ for each $u \in V$
1  W ← a priority queue of $V$;
2  Initialize: $\overline{ecc}(u) \leftarrow +\infty$, $\underline{ecc}(u) \leftarrow 0$, for $\forall u \in V$;
3  while $W$ is not empty do
4      $u \leftarrow W.pop()$;
5      Compute $dist(u, v)$, for $\forall v \in V$, by calling a BFS;
6      $ecc(u) \leftarrow \max_{v \in V} dist(u, v)$;
7      for each node $w \in W$ do
8          $\overline{ecc}(w) \leftarrow \min\{\overline{ecc}(w), ecc(u) + d(u, w)\}$;
9          $\underline{ecc}(w) \leftarrow \max\{\underline{ecc}(w), d(u, w), ecc(u) - d(u, w)\}$;
10         if $\overline{ecc}(w) = \underline{ecc}(w)$ then remove $w$ from $W$;
11 return $ecc(u)$, $\forall u \in V$
```

query on large graphs; this motivated the indexing technique of 2-hop labeling. 2-hop labeling methods label each node w in V with the distances from w to every node in a set $S(w) \subseteq V$. The set $S(\cdot)$ is selected for each node such that for any two nodes u, v in V, $S(u) \cap S(v)$ contains at least one node on a shortest path from u to v. In such a way, the shortest distance can be computed with triangle inequalities:

$$dist(u, v) = \min_{x \in S(u) \cap S(v)} dist(u, x) + dist(x, v).$$

Pruned landmark labeling 2-hop labeling methods for general graphs suffer from a large label set. For the PWSD problem on a special type of graphs—social networks — a *2-hop labeling* method called Pruned landmark labeling (PLL) approach [3] has been proposed: a PWSD query can be answered in *1 microsecond* even for a large social network. The PLL approach is explained in detail in “Appendix I”.

Example 3 Figure 3 shows the pruned landmark labeling for all nodes in graph G (Fig. 1 of the running example). For example, the label of v_5 is $S(v_5) = \{v_1 : 1, v_2 : 1, v_3 : 1, v_5 : 0\}$ and the label of v_8 is $S(v_8) = \{v_1 : 2, v_3 : 2, v_8 : 0\}$. We have $S(v_5) \cap S(v_8) = \{v_1, v_3\}$. Therefore, we can calculate $dist(v_5, v_8) = \min\{dist(v_5, v_1) + dist(v_1, v_8), dist(v_5, v_3) + dist(v_3, v_8)\} = 3$.

Average label length We use PLL as a black box for answering pair-wise shortest distance queries. To quantify the query time of PLL, we introduce the parameter of average label length which is defined as below.

Definition 2 (*Average label length*) Given a graph G, for a node u of G, denote by $S(u)$ the set of nodes selected by the pruned landmark labeling (PLL) approach. The average label length is defined as $b = average_{u \in V}|S(u)| = \Sigma_{u \in V} \frac{|S(u)|}{n}$.

The parameter of average label length cannot be replaced by an existing graph parameter of average degree. As observed in the PLL (Fig. 3) of graph G (Fig. 1), the label length of a node is not proportional to its degree.

Lemma 1 *The shortest distance between two nodes can be computed in $O(b)$ time in expectation by leveraging* PLL.

Proof Computing $\min_{w \in S(u) \cap S(v)} dist(u, w) + dist(w, v)$, for $\forall u, v \in V$, costs $O(|S(u)| + |S(v)|)$. $Exp_{u,v \in V}(|S(u)| + |S(v)|) = 2Exp_{u \in V}|S(u)| = 2b$. □

3 Eccentricity computation

Section 3.1 depicts the state-of-the-art approach, BoundEcc, for computing eccentricities. Section 3.2 identifies and analyzes the key subproblem of BoundEcc—one-node eccentricity computation and global update—to which Sects. 3.3 and 3.4 jointly provide an improved solution.

3.1 The state of the art

Algorithm 1 depicts how BoundEcc [27] computes the eccentricity for each node in V. Following the general framework of computing eccentricity, radius, and diameter, BoundEcc associates each node $u \in V$ with an upper bound $\overline{ecc}(u)$ of the eccentricity $ecc(u)$ and a lower bound $\underline{ecc}(u)$ (Line 2). These bounds are updated (Lines 7–9) until either the bounds meet with each other (Line 11) or the exact $ecc(u)$ is determined by a BFS computation (Lines 5–6). The upper and lower bounds are generally updated with triangle inequalities (Lines 8–9). Lemma 2 provides the rules in updating the bounds. For the priority queue W, the selection of the priority function on V does not affect the correctness of the algorithm but may affect the running time. In [27], the suggested priority queue alternatively pops i) the node with the maximum eccentricity upper bound and ii) the node with the minimum eccentricity lower bound. The underlying priority function is an implicitly generated order on V.

Lemma 2 (Update bounds [27]) *Let u be a node of graph $G(V, E)$ with eccentricity $ecc(u)$. Given a node v and its distance $dist(v, u)$ to u,*

$$ecc(v) \leq ecc(u) + dist(u, v) \quad (1)$$

$$ecc(v) \geq ecc(u) - dist(u, v) \quad (2)$$

$$ecc(v) \geq dist(u, v) \quad (3)$$

Table 1 Execution of BoundEcc

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}
1	**2**	*1,3*	*1,3*	*1,3*	*1,3*	*1,3*	*1,3*	*2,4*	*1,3*	*2,4*	*2,4*
2	–	**3**	*2,3*	*2,3*	*2,3*	*2,3*	*2,3*	*3,4*	*2,3*	*3,4*	2,4
3	–	–	**3**	2,3	2,3	2,3	2,3	3,4	2,3	3,4	*3,4*
4	–	–	–	**3**	2,3	2,3	2,3	3,4	*3,3*	3,4	3,4
5	–	–	–	–	**3**	2,3	2,3	3,4	–	3,4	3,4
6	–	–	–	–	–	**3**	2,3	3,4	–	3,4	3,4
7	–	–	–	–	–	–	**3**	3,4	–	3,4	3,4
8	–	–	–	–	–	–	–	**4**	–	3,4	*4,4*
9	–	–	–	–	–	–	–	–	–	**4**	–

Bold indicates a recalculation Italic indicates an update of lower/upper bound after computing the exact eccentricity for a certain node

As long as one adopts this framework, the worst-case complexity would be *quadratic* to n. BoundEcc can tailor the searching orders in W (Line 7) to different heuristics; however, it has three drawbacks:

- When the eccentricity of a new node is determined, BoundEcc examines the eccentricity bounds of *every* node in W (Line 7), which is exhaustive and unnecessary.
- As long as the upper and lower bounds do not match, that is, $\underline{ecc}(u) < \overline{ecc}(u)$, even if the gap is only one, BoundEcc will compute $ecc(u)$ *from scratch*. The effort that has been invested in updating the upper and lower bounds is entirely wasted.
- BFS has to traverse *all* the nodes in V which leaves no chance for an early termination since $ecc(u)$ is determined by the last visited node in a BFS from u.

Example 4 The execution process of BoundEcc is shown in Table 1. The two numbers in each cell are the upper and lower bounds of eccentricity for the corresponding node. Using BoundEcc, we need to recalculate the eccentricity for 9 nodes. The bold indicates a recalculation while the italic means an update of lower/upper bound after computing the exact eccentricity for a certain node. For example, after computing $ecc(v_2) = 3$, we can update the lower/upper bounds of v_3 from 1, 3 to 2, 3. The BoundEcc algorithm involves a large number of exact shortest path calculations.

3.2 Problem analysis

In viewing the three drawbacks of BoundEcc, we focus on the following two problems to enable an efficient eccentricity computation.

Problem 3 (*Exact eccentricity computation for a node*) Given a node x in graph $G(V, E)$, associated with $\underline{ecc}(x)$ and $\overline{ecc}(x)$, determine the eccentricity $ecc(x)$ of x.

Let each node $u \in V$ in $G(V, E)$ bear an upper bound $\overline{ecc}(u)$ and a lower bound $\underline{ecc}(u)$ on the eccentricity $ecc(u)$ of u. The bounds $\{\overline{ecc}(u), \underline{ecc}(u)\}, \forall u \in V$, are called the **eccentricity bounds**.

Let x be an arbitrary node with $\overline{ecc}(x) > \underline{ecc}(x)$. Consider the process of two steps: i) compute the shortest distances from x to all the other nodes (Line 5, Algorithm 1) and ii) update the eccentricity bounds of all the nodes (Lines 6–10) based on the newly computed distances from x. We call x the **trigger node** of step ii). Instead of using $ecc(x)$ to update the bounds globally by Lemma 2 as BoundEcc, we define the problem of efficient bound update below.

Problem 4 (*Efficient bound update*) Let a state Sta_1 be the eccentricity bounds of all nodes v in V with $\underline{ecc}(v) < \overline{ecc}(v)$. Start the bound update once the trigger node x with both $\underline{ecc}(x)$ and $\overline{ecc}(x)$ in Sta_1 becomes its exact eccentricity $ecc(x)$. Consider another state Sta_2 of the eccentricity bounds of V obtained by recursively applying Lemma 2 to Sta_1 until it is stable—Lemma 2 can no longer update any bound in Sta_2. The problem of efficient bound update minimizes the computation cost in finding Sta_2 based on $ecc(x)$ and Sta_1.

The aim is to reduce the exhaustive BFS computation in Problem 3 and to avoid the update on the eccentricity bounds across the node set of V. This may not be achieved without proper indexing structures.

1. We pre-compute the distance $dist(z, u)$ from a **reference node** z (the selection of z will be explained later) to all the nodes u in V. The nodes in V are stored in a list $L_z = \{u_1, u_2, \ldots, u_n\}$ with non-descending distances associated $dist(z, u_1) \leq dist(z, u_2) \leq \ldots \leq dist(z, u_n)$.
2. The state-of-the-art 2-hop labeling method for the pairwise shortest distance (PWSD) queries. Instead of obtaining the eccentricity of $u \in V$ by performing BFS from u until it reaches the maximum shortest distance (Lines 5–6, Algorithm 1), we probe the distances from x to its potentially farthest nodes by invoking pair-wise shortest distance (PWSD) queries. Specifically, we pre-compute the **pruned landmark labeling** (PLL) structure which answers the shortest distance of any two nodes in $O(b)$ time where b denotes the average label length of PLL (Sect. 2.2).

It is worth noting that our study is independent of the technique of PLL: we consider PLL as merely an embodiment of a black box for answering PWSD queries while BoundEcc necessitates n^2 PWSD queries—even if PLL can magically answer a query in $O(1)$ time, the drawbacks of BoundEcc still retain.

Section 3.3 solves Problem 3 without BFS computation, while Sect. 3.4 exempts a solution to Problem 4 from traversing the node set.

3.3 Exact eccentricity computation for a node

Recall that Problem 3 aims at efficiently computing the eccentricity $ecc(x)$ of a node x. Section 3.3.1 introduces a solution to Problem 3 based on the list L_z if a given reference node z, and Sect. 3.3.2 discusses how to speed up the solution in Sect. 3.3.1 by allowing each node x to choose its own reference node z.

3.3.1 Computing $ecc(x)$ under a fixed reference node

The very reason that BFS has to traverse the whole graph to get the eccentricity of x is the conflict between

- the non-decreasing order of the distances from nodes to x in which BFS follows, and
- the max nature in $ecc(x)$ among the distances of all nodes to x.

If the order of BFS traversal can be reversed, that is, the nodes with longer distance from x will be visited earlier, then the first node we visit can already provide $ecc(x)$. However, unless x coincides with the reference node z whose distance information has been indexed, the BFS traversal of x can hardly be reversed.

Utilize the pruned landmark labeling (PLL) *structure* PLL efficiently answers pair-wise shortest distance queries—probing the distances from x to a small subset $V' \subseteq V$ of nodes becomes inexpensive. By aggregating the exact distances from x to nodes in V', we can partially evaluate the eccentricity of x.

Definition 3 (*Partial eccentricity*) Given a subset $V' \subseteq V$, define the partial eccentricity of x on V' as $pecc(x|V') = \max_{u \in V'} dist(x, u)$.

Lemma 3 *Let V' be a subset of V. $pecc(x|V') \le ecc(x)$, that is, partial eccentricity provides a lower bound for the eccentricity. Besides, $pecc(x|V) = ecc(x)$.*

Utilize the reference node z To transfer the knowledge gained on the reference node z to the computation of $ecc(v)$, we first introduce the definition of a bounded set with bounded eccentricities.

Definition 4 (*Bounded set*) Given $\lambda \ge 0$, the bounded set

$$V^z_{\le\lambda} = \{u \in V | dist(u, z) \le \lambda\}.$$

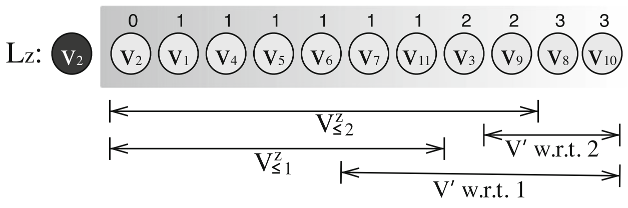

Fig. 4 Illustration of bounded set and partial set ($z = v_2$)

Lemma 4 (Bounded eccentricity) *For $\lambda \ge 0$, the partial eccentricity of a bounded set of λ is also bounded:*

$$pecc(x|V^z_{\le\lambda}) \le dist(x, z) + \lambda.$$

Proof Please see "Appendix A". □

Definition 5 (*λ-partial set*) Given $\lambda \ge 0$, a set V' is called a λ-partial set, if $V' \cup V^z_{\le\lambda} = V$; namely, V' is a super set of $V \setminus V^z_{\le\lambda}$.

Example 5 Figure 4 shows L_z for $z = v_2$ of the running graph in Fig. 1. Obviously, if $x = z$, we can determine $ecc(x) = 3$ directly from L_z. We also show $V^z_{\le 1}$ and $V^z_{\le 2}$ where $V^z_{\le 1}$ contains the first 7 nodes in L_z, while $V^z_{\le 2}$ contains the first 9 nodes in L_z. The λ-partial set V' with respect to $\lambda = 2$ can be any subset of V that contains $\{v_8, v_{10}\}$; V' w.r.t. $\lambda = 1$ can be any subset containing $\{v_3, v_9, v_8, v_{10}\}$. For $x = v_4$ and $\lambda = 2$, $pecc(x|V^z_{\le\lambda}) = \max_{v \in V^z_{\le\lambda}} dist(x, v) = 2$. Obviously, we have $pecc(x|V^z_{\le\lambda}) \le dist(x, z) + \lambda = dist(v_4, v_2) + 2 = 3$.

Utilize both the reference node z and the PLL Given λ, we combine the information we get from the reference node z and the partial eccentricity on a λ-partial set.

Lemma 5 *Given a λ-partial set V', the eccentricity*

$$ecc(x) = \begin{cases} pecc(x|V'), & \text{if } pecc(x|V^z_{\le\lambda}) \le pecc(x|V') \\ pecc(x|V^z_{\le\lambda}), & otherwise \end{cases}$$

Proof Please see "Appendix B". □

Assume that at a time, for a given λ and a corresponding λ-partial set V', $pecc(x|V')$ is obtained using PLL while $pecc(x|V^z_{\le\lambda})$ is bounded by Lemma 4. Is it possible that we can determine the eccentricity $ecc(x)$? The following theorem provides a positive answer.

Theorem 1 *Given a λ-partial set V', if $pecc(x|V') \ge dist(x, z) + \lambda$, then $ecc(x) = pecc(x|V')$.*

Proof From Lemma 4, $pecc(x|V^z_{\le\lambda}) \le dist(x, z) + \lambda$. If $pecc(x|V') \ge dist(x, z) + \lambda$, then $pecc(x|V') \ge pecc(x|V^z_{\le\lambda})$. Lemma 5 leads to $ecc(x) = pecc(x|V')$. □

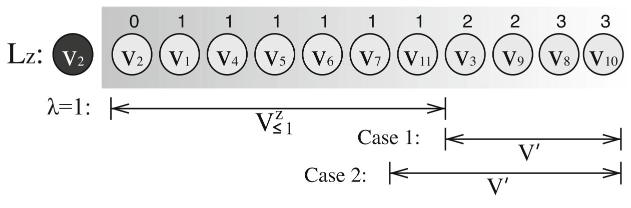

Fig. 5 Illustration of $V^z_{\le\lambda}$ and V' for $\lambda = 1$ $(z = v_2)$

Algorithm 2: EccentricityOneNode

```
    Input: Node $x$, $z$, $\overline{ecc}(x)$, $\underline{ecc}(x)$, $L_z$, the PLL structure
    // $L_z = \{u_1, u_2, \ldots, u_n\}$.
    Output: $ecc(x)$
1   $p \leftarrow 0$;                       // $p = pecc(x|V')$, $V'$ is initially $\emptyset$
2   for $i$ takes from $n$ down to 1 do
3       if $i = 1$ then $\lambda \leftarrow 0$; else $\lambda \leftarrow dist(z, u_{i-1})$;
4       Obtain $dist(x, u_i)$ by querying PLL;
5       $p \leftarrow \max\{p, dist(x, u_i)\}$; // By absorbing $u_i$, $V'$ remains a
        $\lambda$-partial set $p = pecc(x|V')$.
6       $\underline{ecc}(x) \leftarrow \max\{\underline{ecc}(x), p\}$;
7       $\overline{ecc}(x) \leftarrow \min\{\overline{ecc}(x), \max\{p, dist(x, z) + \lambda\}\}$;
8       if $\underline{ecc}(x) = \overline{ecc}(x)$ then return $\underline{ecc}(x)$;
9   return $\underline{ecc}(x)$
```

Example 6 Suppose we want to calculate $ecc(v_9)$. Given $\lambda = 1$ and $z = v_2$, $pecc(v_9|V^z_{\le\lambda}) = 3$. Figure 5 shows two cases to select the partial set V'. In Case 1, we have $pecc(v_9|V^z_{\le\lambda}) > pecc(v_9|V') = 2$. We can thus compute $ecc(v_9) = pecc(v_9|V^z_{\le\lambda})$. In Case 2, we have $pecc(v_9|V^z_{\le\lambda}) \le pecc(v_9|V') = 3$. We can thus compute $ecc(v_9) = pecc(v_9|V')$; we also have $pecc(v_9|V') = 3 \ge dist(v_9, z) + \lambda = 3$. Thus, we can compute $ecc(v_9) = pecc(v_9|V')$ without knowing $pecc(v_9|V^z_{\le\lambda})$.

Theorem 1 leads to the bounds of $ecc(x)$.

Lemma 6 *Given a* λ*-partial set* V', $pecc(x|V')$ *is a lower bound on* $ecc(x)$ *and* $\max\{pecc(x|V'), dist(x, z) + \lambda\}$ *is an upper bound on* $ecc(x)$.

When the upper bound meets the lower bound, $ecc(u)$ can be identified directly. This allows us to leverage the original upper and lower bounds on u to narrow the gap. Now, we are ready to introduce our approach.

Exact eccentricity computation Theorem 1 implies an approach in determining $ecc(v)$: traverse nodes of V in a non-increasing order of their distances to the reference node z; in this process, update the upper bound and lower bound using Lemma 6; terminate once the upper and lower bounds meet. Algorithm 2 is such an approach.

Algorithm 2 conceptually develops a λ-partial set V' along a decreasing variable λ. In other words, when λ decreases from $dist(z, u_n)$ to 0, V' expands from $\emptyset$ to V accordingly. Initially, $pecc(v|V') = 0$ since $\lambda = dist(z, u_n)$, and thus, $V' = \emptyset$ is a λ-partial set (Line 1). Then, nodes in V are examined in a reverse order of L_z, the rearrangement of all the nodes in V by the non-descending distances with z, as formally defined in Sect. 3.2 (Line 2). For each node u_i with $i > 1$, λ is set to be the distance from the reference node z to u_{i-1}, otherwise $z = u_i$, and thus, let λ be 0 (Line 3). Obviously, $\{u_1, u_2, \ldots, u_{i-1}\}$ is a subset of $V^z_{\le\lambda}$. V' is augmented with u_i such that it remains a λ-partial set of the newly updated λ. The partial eccentricity $p = pecc(x|V')$ on V' is updated accordingly (Lines 4–5). The upper bound and lower bound of $ecc(x)$ are then updated (Lines 6–7). The loop will be terminated immediately when the gap between the two bounds becomes 0 (Line 8). The entire loop transforms $\underline{ecc}(x)$ to $ecc(x)$ (Line 9).

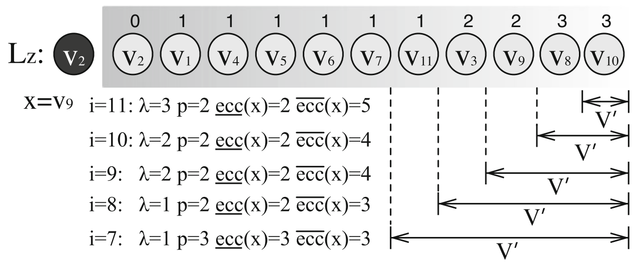
Lz:
V2
0 1 1 1 1 1 1 2 2 3 3
V2 V1 V4 V5 V6 V7 V11 V3 V9 V8 V10
x=v9
i=11: λ=3 p=2 ecc(x)=2 ecc(x)=5
i=10: λ=2 p=2 ecc(x)=2 ecc(x)=4
i=9: λ=2 p=2 ecc(x)=2 ecc(x)=4
i=8: λ=1 p=2 ecc(x)=2 ecc(x)=3
i=7: λ=1 p=3 ecc(x)=3 ecc(x)=3
V′
V′
V′
V′
V′

Fig. 6 Process to compute $ecc(v_9)$ $(z = v_2)$

Theorem 2 *Algorithm 2 reports the eccentricity of* x.

Proof Lemma 6 ensures that the upper and lower bounds of the eccentricity of x are correctly updated (Lines 6–7). Therefore, if the two bounds match, they match on $ecc(x)$ (Line 8). If the two bounds have not agreed by the end of the loop when $V' = V$, then $\underline{ecc}(x) = pecc(x|V') = pecc(x|V) = ecc(x)$ can be safely reported (Line 9) due to Lemma 3. □

Example 7 Figure 6 illustrates the process to compute $ecc(x)$ for $x = v_9$ for the running graph shown in Fig. 1. Suppose $z = v_2$, for $i = 11$ $(\lambda = 3)$, the algorithm first computes $dist(x, v_{10}) = 2$ by querying PLL, updates p to be 2, and updates $\underline{ecc}(x)$ and $\overline{ecc}(x)$ to be 2 and 5, respectively. Then, for $i = 10$ $(\lambda = 2)$, the algorithm computes $dist(x, v_8) = 1$ and updates $\overline{ecc}(x)$ to be $dist(x, z) + \lambda = 4$. The process continues until $i = 7$ $(\lambda = 1)$, where the algorithm computes $dist(x, v_{11}) = 3$ and updates $p = 3$ and $\underline{ecc}(x) = 3$. At this time, we have $\underline{ecc}(x) = \overline{ecc}(x)$, and therefore, the algorithm terminates by returning 3 as $ecc(x)$.

Theorem 3 *In the worst case, Algorithm 2 reports the eccentricity of* v *with* n PLL *queries.*

Proof For node u_i, $i \in [1, n]$, Algorithm 2 computes the exact distances from x to u_i using the structure of PLL in $O(b)$ time. In the worst case, Algorithm 2 scan the whole nodes in G, which incurs n PLL queries. □

Example 8 Figure 7 demonstrates the process to compute the eccentricity for all nodes in the running graph in Fig. 1 by

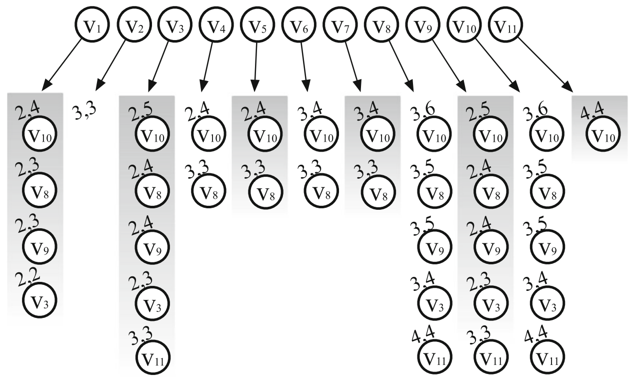
V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11
2,4 V10 2,3 V8 2,3 V9 2,2 V3
3,3
2,5 V10 2,4 V8 2,4 V9 2,3 V3 3,3 V11
2,4 V10 3,3 V8
2,4 V10 3,3 V8
3,4 V10 3,3 V8
3,4 V10 3,3 V8
3,6 V10 3,5 V8 3,5 V9 3,4 V3 4,4 V11
2,5 V10 2,4 V8 2,4 V9 2,3 V3 3,3 V11
3,6 V10 3,5 V8 3,5 V9 3,4 V3 4,4 V11
4,4 V10

Fig. 7 Computing eccentricity for all nodes ($z = v_2$)

setting the reference node as $z = v_2$. For example, for node v_1, we need to query PLL for 4 times. The $\underline{ecc}$ and $\overline{ecc}$ values after each PLL query are also labeled beside each node. In total, we have 33 PLL queries.

Remarks 1 In finding the eccentricity of a node, Algorithm 2 is superior to BFS in practice:

1. Algorithm 2 boosts the computation of $ecc(x)$ by leveraging the previously computed upper and lower bounds on $ecc(x)$. For example, if $\underline{ecc}(x)$ is smaller than $\overline{ecc}(x)$ by a tiny margin (e.g., 1) before Algorithm 2 starts, then one effective update on either the upper or the lower bound (in Lines 6–7) can terminate Algorithm 2, as opposed to searching the whole graph in $O(m)$ time using a BFS.
2. Algorithm 2 traverses the nodes in the reverse order of their distances to the reference node z. A reference node z that is close to x can terminate the algorithm at an early stage since the farthest node to x will be far from z as well.

3.3.2 Reference-node pool

As observed from Sect. 3.3.1 (Remarks 2) and Lemma 2, a reference node z in the vicinity of x can terminate Algorithm 2 at an early stage. Specifically, consider the case when $dist(x, z) = 1$. From Lemma 2, $ecc(z) - 1 \leq ecc(x) \leq 1 + ecc(z)$. Besides, by the triangle inequality, $ecc(z) - 1 \leq dist(x, u_n) \leq ecc(z) + 1$. Here, u_n denotes, in the list of L_z, the last element whose distance $dist(z, u_n)$ to z is the eccentricity $ecc(z)$ of z.

– If $dist(x, u_n) = dist(z, u_n) + 1$, we can immediately terminate Algorithm 2. Note that, the chance of this case is considerable since $dist(z, u_n) + 1$ is one of the three values that $dist(x, u_n)$ can possibly take.
– Otherwise, $dist(x, u_n)$ provides a strong lower bound for $ecc(x)$: $ecc(x) \geq dist(x, u_n) \geq ecc(z) - 1$. Let u be the node with $dist(u, x) = ecc(x)$. Then, $dist(u, z) \geq dist(u, x) - dist(x, z) \geq ecc(x) - 1 \geq ecc(z) - 2$. That is, once the $\lambda = dist(u_{i-1}, z)$ in Line 3, Algorithm 2, drops below $ecc(z) - 2$, we can safely terminate the algorithm.

Algorithm 3: RefPool

```
  Input: Graph $G(V, E)$, $k$
  Output: $pool$; $L_z, \forall z \in pool$; $\overline{ecc}(u)$ and $\underline{ecc}(u)$, $\forall u \in V$
1 Let $pool$ be the $k$ nodes in $G$ with the highest degrees;
2 for each node $z \in pool$ do
3     $L_z \leftarrow$ a list of nodes in $V$ in non-decreasing order of their distances to $z$;
4     for each node $u \in V$ do
5         update $\overline{ecc}(u)$ and $\underline{ecc}(u)$ using Lemma 2;
6 return as required.
```

Example 9 Suppose we would like to compute $ecc(x)$ for $x = v_4$ with reference node $z = v_2$. Note that $dist(x, z) = 1$. Therefore, in the worst case, we need to visit those nodes y with $dist(y, z) \geq ecc(z) - 2 = 1$, which are all nodes except $z = v_2$. As shown in Fig. 7, we visit the node set $\{v_{10}, v_8\}$ to compute $ecc(v_4)$, which is a subset of the needed nodes.

The theorem below generalizes the analysis on the case of $dist(x, z) = 1$.

Theorem 4 *Denote by λ_0 the distance $dist(x, z)$ between x and z. Let y be the farthest node to x, that is, $ecc(x) = dist(y, x)$. Then,*

$$dist(y, z) \geq ecc(z) - 2\lambda_0.$$

Therefore, it suffices for Algorithm 2 to visit all nodes in $\{v \in V | dist(v, z) \geq ecc(z) - 2\lambda_0\}$ to determine $ecc(x)$.

Proof From Lemma 2, $ecc(z) - \lambda_0 \leq ecc(x) = dist(y, x) \leq dist(y, z) + dist(x, z) = dist(y, z) + \lambda_0$ □

Theorem 4 implies that to determine $ecc(x)$, it suffices to scan only the nodes whose distances fall into a range of length $2\lambda_0$. The smaller the λ_0 is, the earlier the search of $ecc(x)$ may stop. Therefore, it boils down to select a reference node that is as close as possible to x.

Let k be a small integer. Algorithm 3 selects k highest degree nodes in $G(V, E)$ as the pool of reference nodes (Line 1). Each node x, as can be seen in Algorithm 5 (Line 4), chooses the reference node in the pool that is the closest to x. As verified by the experiments (*Exp-2*, Sect. 6), setting k as 16 is sufficient to make an excellent performance for eccentricity computation.

The adoption of a reference node pool is cost-effective. The pre-computation for the pool of reference nodes incurs only a small overhead, but it sets up the initial upper and lower bounds for the eccentricity of each node (Lines 4–5). More importantly, the pool of reference nodes scatters in the vicinity of most nodes in V; in other words, the averaged

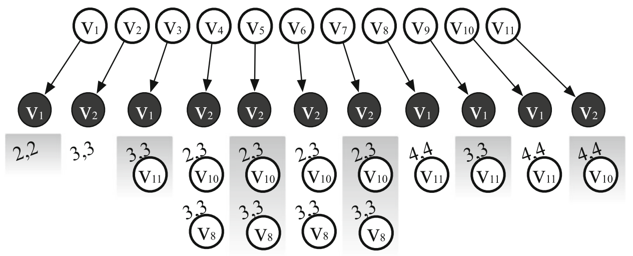

Fig. 8 Eccentricity computation ($pool = \{v_1, v_2\}$)

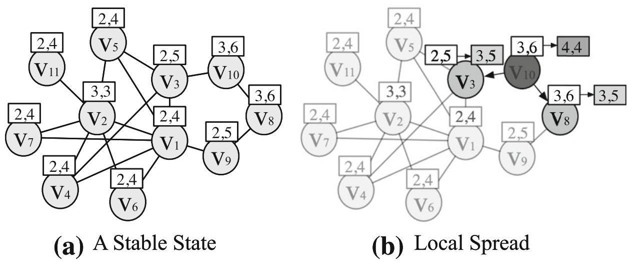

Fig. 9 Local spread: before and after updating the bounds of v_{10}

distance $d(x, z)$ from a node x to its closest reference node z is relatively small, as has been verified experimentally (*Exp-5*, Sect. 6). This is not surprising since, in a small-world network, a few high-degree nodes have significantly higher connections than the other nodes; these nodes can connect the majority of the graph in merely two to three hops.

Example 10 In the running example, graph G is shown in Fig. 1. Let $k = 2$. Select 2 nodes with the highest degree in G as the reference-node pool: $pool = \{v_1, v_2\}$. The process of computing the eccentricity for all nodes in G is then shown in Fig. 8. The reference node selected for each node is colored in black. For example, for node $x = v_3$, v_1 instead of v_2 is selected as the reference node since $dist(v_2, v_3) > dist(v_1, v_3)$. Using v_1 as the reference node, one only needs to obtain $dist(v_3, v_{11}) = 3$ to terminate the algorithm with $ecc(v_3) = 3$ by querying PLL. Compared to Example 8, using the reference-node pool reduces the number of PLL queries from 33 to 13.

3.4 Bound update optimization

With the eccentricity $ecc(x)$ of the trigger node x newly identified in Sect. 3.3, it remains to update the eccentricity bounds of other nodes (Problem 4 in Sect. 3.2). Globally traversing over all nodes in V is exhaustive. This section turns the global update to a much cheaper yet effectively the same local update.

It can be derived from Lemma 2 that the eccentricities on adjacent nodes differ by at most one.

Lemma 7 *For each undirected edge $(u, v) \in E$,*

$$ecc(u) - 1 \le ecc(v) \le ecc(u) + 1.$$

When applying Lemma 7 to the eccentricity bounds, we derive the notion of a stable state.

Definition 6 (*Stable state*) The eccentricity bounds are **stable**, if for each edge $(u, v) \in E$:

$$\overline{ecc}(u) \le \overline{ecc}(v) + 1, \ \textit{and} \tag{4}$$

$$\underline{ecc}(u) \ge \underline{ecc}(v) - 1. \tag{5}$$

Example 11 The eccentricity bounds shown in Fig. 9a are stable. For example, for adjacent nodes v_3 and v_{10}, $\overline{ecc}(v_{10}) \le \overline{ecc}(v_3) + 1$ and $\underline{ecc}(v_3) \ge \underline{ecc}(v_{10}) - 1$.

Section 3.4.1 will show that the global update under the first two rules of Lemma 2 can be materialized by iteratively applying Lemma 7 in a local spread update.

3.4.1 Iterative update

This subsection describes an iterative update process of the eccentricity bounds from one stable state to another stable state in accordance with the update of the eccentricity bounds of the trigger node x.

Let x be the trigger node. Consider a stable state of eccentricity bounds (Definition 6) before the update. To compare the eccentricity bounds before and after the update, we conceptually capture the eccentricity bounds before the update in a snapshot of $\{\overline{ecc}_{old}(u), \underline{ecc}_{old}(u)\}$, for $\forall u \in V$.

Denote by ub_x and lb_x, respectively, the new upper and lower eccentricity bounds for the trigger node x—if $ecc(x)$ is available, then $ub_x = lb_x = ecc(x)$. The update process has two steps.

1. Update the eccentricity bounds of x:
   (a) $\overline{ecc}(x) \leftarrow \min\{\overline{ecc}_{old}(x), ub_x\}$
   (b) $\underline{ecc}(x) \leftarrow \max\{\underline{ecc}_{old}(x), lb_x\}$
2. Iteratively update the eccentricity bounds based on Lemma 7 and terminate when eccentricity bounds become stable. Specifically, update the eccentricity bounds of l from that of r across edge $(l, r) \in E$:
   (a) $\overline{ecc}(l) \leftarrow \min\{\overline{ecc}(l), \overline{ecc}(r) + 1\}$
   (b) $\underline{ecc}(l) \leftarrow \max\{\underline{ecc}(l), \underline{ecc}(r) - 1\}$

Take a snapshot of the eccentricity bounds after an iterative update: $\{\overline{ecc}_{new}(u), \underline{ecc}_{new}(u)\}, \forall u \in V$.

3.4.2 Local spread

The iterative update, as we shall prove in the following theorem, narrows the eccentricity bounds over a connected subgraph of G "centered" at the trigger node x. This property of iterative update enables an optimization which effectively reduces the overhead.

Theorem 5 (Update locality) *A node v is dirty if its eccentricity bounds are updated with $\overline{ecc}_{old}(y) > \overline{ecc}_{new}(y)$ or $\underline{ecc}_{old}(y) < \underline{ecc}_{new}(y)$ — otherwise v is clean. Let V' be the set of dirty vertices, i.e., $V' = \{y \in V | \overline{ecc}_{old}(y) > \overline{ecc}_{new}(y)$ or $\underline{ecc}_{old}(y) < \underline{ecc}_{new}(y)\}$. If V' is not empty, then graph $G'(V', E')$ defined with $E' = \{(u, v) \in E | u, v \in V'\}$ is connected, that is, for any two nodes a, b in V', there is a path in G' from a to b. Besides, for each node y in V',*

- $\overline{ecc}_{new}(y) = \min\{\overline{ecc}_{old}(y), ub_x + dist(x, y)\}$
- $\underline{ecc}_{new}(y) = \max\{\underline{ecc}_{old}(y), lb_x - dist(x, y)\}$.

Theorem 5 enables us to design a local spread update approach in Algorithm 4. To update the eccentricity bounds, we visit the nodes in V in the style of Breadth-First Search (Lines 2,5) starting from the trigger node x. During the traversal, the nodes that were visited first would be popped out first in the queue Q. If one node u remains "clean" after the visit, we halt the node expansion from u (Lines 3,6,10). In this sense, the BFS is kept in a local area. All the distances (Lines 8–9) are obtained in the local BFS rather than pair-wise distance queries. Checking whether a node is "dirty" (Lines 3,7,10,11) takes constant time via tagging. The search terminates when all node expansions have been completed (Line 4).

Lemma 8 (Complexity) *Let set $V' = \{y \in V | \overline{ecc}_{old}(y) > \overline{ecc}_{new}(y)$ or $\underline{ecc}_{old}(y) < \underline{ecc}_{new}(y)\}$ be the set of nodes that has been affected by the update on the trigger node x. The complexity of Algorithm 4 is $O(\Sigma_{v \in V'} deg(v))$.*

Proof Algorithm 4 terminates when all the nodes in V' have been visited. Since each node is visited only once, the time complexity is $O(\Sigma_{v \in V'} deg(v))$. □

Example 12 Figure 9a shows a stable state for the graph in Fig. 1 of the running example. Suppose we calculate $ecc(v_{10}) = 4$, we update $\underline{ecc}(v_{10}) = \overline{ecc}(v_{10}) = 4$. Using local spread, for the neighbor v_3 of v_{10}, we update $\underline{ecc}(v_3)$ to be 3. For the neighbor v_8 of v_{10}, we update $\overline{ecc}(v_8)$ to be 5. After local spread, the state becomes stable again as shown in Fig. 9b.

Correctness of Theorem 5 Iterative update tightens the eccentricity bounds iteratively without specifying the convergence rate. To quantify the margin between the old and new snapshots, we found bounds on the margins: Lemmas 9 (weak), 10 (strong), and 11 (strong), respectively.

Algorithm 4: LocalSpread

```
Input: Node $x$, $ub_x$, $lb_x$, a stable eccentricity bounds.
Output: A stable eccentricity bounds
// All eccentricity bounds are clean. A bound $b$ gets
   dirty once it is updated, which can be
   materialized by setting a field of $b$ with $x$.
1  $\overline{ecc}(x) \leftarrow \min\{\overline{ecc}(x), ub_x\}$, $\underline{ecc}(x) \leftarrow \max\{\underline{ecc}(x), lb_x\}$;
2  $Q \leftarrow$ an empty queue;
3  if either $\overline{ecc}(x)$ or $\underline{ecc}(x)$ is dirty then add $x$ to $Q$;
4  while $Q$ is not empty do
5      $u \leftarrow Q.pop()$; mark $u$ as visited;
6      if either $\overline{ecc}(u)$ or $\underline{ecc}(u)$ is dirty then
7          for each unvisited neighbor $v$ of $u$ do
8              $\overline{ecc}(v) \leftarrow \min\{\overline{ecc}(v), ub_x + dist(v, x)\}$;
9              $\underline{ecc}(v) \leftarrow \max\{\underline{ecc}(v), lb_x - dist(v, x), dist(v, x)\}$;
10             if either $\overline{ecc}(v)$ or $\underline{ecc}(v)$ is dirty then
11                 add $v$ to $Q$ if $v$ is not in $Q$;
```

Lemma 9 (Weak bounds) *For any node $v \in V$ other than u in the graph,*

$$\overline{ecc}_{new}(u) \leq \min\{\overline{ecc}_{old}(u), \overline{ecc}_{new}(x) + dist(x, u)\} \quad (6)$$

$$\underline{ecc}_{new}(u) \geq \max\{\underline{ecc}_{old}(u), \underline{ecc}_{new}(x) - dist(x, u)\} \quad (7)$$

Proof Please see "Appendix C". □

Lemma 10 (Strong upper bounds) *For each node $y \in V$ with $\overline{ecc}_{new}(y) < \overline{ecc}_{old}(y)$, there exists a shortest path $\langle u_0, u_1, \ldots, u_k \rangle$ from x to y with $k = dist(y, x)$ such that*

1. *for each $i \in [0, k]$, $\overline{ecc}_{new}(u_i) = ub_x + dist(u_i, x)$.*
2. *for each $i \in [0, k]$, $\overline{ecc}_{new}(u_i) < \overline{ecc}_{old}(u_i)$.*
3. *$\overline{ecc}_{new}(x) = ub_x$.*

Proof Please see "Appendix D". □

All the results on the upper bounds in Lemma 10 can be symmetrically applied on lower bounds.

Lemma 11 (Strong lower bounds) *For each node $y \in V$ with $\underline{ecc}_{new}(y) > \underline{ecc}_{old}(y)$, there exists a shortest path $\langle u_0, u_1, \ldots, u_k \rangle$ from x to y with $k = dist(y, x)$ such that*

1. *for $\forall i \in [0, k]$, $\underline{ecc}_{new}(u_i) = \underline{ecc}_{new}(x) - dist(u_i, x)$.*
2. *for $\forall i \in [0, k]$, $\underline{ecc}_{new}(u_i) > \underline{ecc}_{old}(u_i)$.*
3. *$\underline{ecc}_{new} x = lb_x$.*

3.4.3 Putting all parts together

We now have all parts in the puzzle of exact eccentricity computation completed in Algorithm 5. We pre-compute an auxiliary structure PLL to efficiently answer pair-wise shortest distance queries (Line 1). Select the reference-node pool with k nodes (Line 2). For each node $v \in V$ (Line 3), we first find the reference node in the pool with the smallest distance to x (Line 4) and then use the reference node to compute the exact eccentricity of x (Line 5). After that, we use $ecc(x)$ to update the eccentricity bounds (Line 6). Finally, we are able to report the eccentricities of all nodes in V (Line 7). The correctness of Algorithms 3–5 can be easily guaranteed by the triangle inequality, Lemmas 2 and 7.

Theorem 6 *The amortized number of nodes whose eccentricities are updated by a call of* LocalSpread *is* $O(d)$.

Proof According to Theorem 5, a node $y \in V$ is updated by LocalSpread only when $\overline{ecc}(y)$ is increased to $ecc(x) + dist(x, y)$ or $\underline{ecc}(y)$ is decreased to $ecc(x) - dist(x, y)$ upon $ecc(x)$ of the corresponding trigger node x. Note that $ecc(x) + dist(x, y) \leq 2d$, here d is the diameter of the graph. Thus, after the first update on $\overline{ecc}(y)$, $\overline{ecc}(y)$ will be decreasing within the range of $[0, 2d]$. Therefore, $\overline{ecc}(y)$ will be updated at most $2d + 1$ times. We can similarly prove that $\underline{ecc}(y)$ will be updated at most $d + 1$ times. In total, LocalSpread will update the eccentricity bounds of y at most $3d + 1$ times. The total of $O((3d + 1)n)$ updates over all nodes took place in n calls of LocalSpread.The amortized number of nodes whose eccentricities are updated by a call of LocalSpread is therefore $O(d)$. □

Lemma 12 *The time complexity on updating the eccentricity bounds in Algorithm 5 in total is* $O(dm)$.

Proof According to Lemma 8, the adjacency list of a node $y \in V$ is visited by LocalSpread only when a bound of y is updated. Since each node is updated $O(d)$ times and each update reads through the corresponding node's adjacency list, the total time for LocalSpread is $O(d \cdot \sum_{v \in V} deg(v)) = O(dm)$. □

Remarks 2 The worst-case complexity of Algorithm 5 is quadratic; however,

- Algorithm 5 determines the eccentricity of a node x at an early stage by i) searching from remote nodes of x guided by a reference node that is close to x and ii) inheriting the eccentricity bounds of x from the outer loop which terminates the search whenever the bounds meet.
- The time complexity for updating the eccentricity bounds is near linear since the total cost is $O(dm)$ while a small-world network has a small d ($d < 50$ for all datasets in Table 3).

Therefore, within the loop of a node in V (Line 3, Algorithm 5), the practical cost is far less than n. In this sense, our algorithm is more efficient than its counterparts.

4 Eccentricity maintenance

Section 3 introduces an efficient algorithm for computing the eccentricity distribution of a graph. Applying this algorithm to recompute the eccentricity upon each update, when it comes to dynamic graphs, is obviously undesirable since the graph can frequently change which makes the maintenance time critical.

Algorithm 5: ECC-LS

```
Input: Graph $G(V, E)$, k
Output: $ecc(u)$ for each $u \in V$
1 PLL ← the PLL structure of $G(V, E)$;
2 $pool$, $L_z$,eccentricity bounds ← RefPool($G(V, E), k$);
3 for each node $x \in V$ with $\underline{ecc}(x) \neq \overline{ecc}(x)$ do
4     $z$ ← the node in the pool that is nearest to $x$;
5     $ecc(x)$ ← EccentricityOneNode($x, z, \overline{ecc}(x), \underline{ecc}(x), L_z$, PLL);
6     LocalSpread($x, ecc(x), ecc(x)$,eccentricity bounds);
7 return $ecc(u), \forall u \in V$
```

This section considers the eccentricity maintenance problem formulated in Problem 2 in Sect. 2.1 which aims at maintaining the eccentricity distribution upon updating an edge $e(a, b)$ in a graph.

We denote the graph, the eccentricity of a node v, and the shortest distance between two nodes (u, w) before and after the update as G and G', $ecc(v)$ and $ecc'(v)$, and $dist(u, w)$ and $dist'(u, w)$, respectively. We call a node v *maintenance dirty* if its eccentricity is affected by the update, that is, $ecc(v) \neq ecc'(v)$, and denote the set of maintenance dirty nodes as $C^{\text{m-dirty}} = \{v \in V | ecc(v) \neq ecc'(v)\}$. An efficient maintenance of the eccentricity distribution includes an efficient identification of $C^{\text{m-dirty}}$ and an efficient update of the eccentricities of nodes in $C^{\text{m-dirty}}$.

In order to identify $C^{\text{m-dirty}}$, we first inspect how pairwise shortest distances are affected by the update (Sect. 4.1). After that, we roughly scope potential maintenance dirty nodes (Sect. 4.2) and then facilitate a more targeted eccentricity recomputation by determining the eccentricities of the other maintenance dirty nodes with the rules we have found (Sect. 4.3).

4.1 Pair-wise distances affected by edge updates

We first brief the notions and findings of Yen et. al. [33] on the problem of how an edge update affects the shortest distance between two nodes. They proposed these notions and findings to maintain the closeness of nodes in a graph. The techniques on closeness maintenance, however, fail on eccentricity maintenance, as we shall see below.

Lemma 13 *When edge* $e(a, b)$ *is inserted into* G *(or deleted from* G*, resp.), for two nodes* $v, w \in V$ *with* $dist'(v, w) \neq dist(v, w)$*, each shortest path between* v *and* w *must go through* e *in* G' *(or* G*, resp.).*

Proof Please find the proof in "Appendix E". □

Definition 7 When edge $e(a, b)$ is inserted into G (or deleted from G, resp.), two critical sets C^a and C^b are derived from

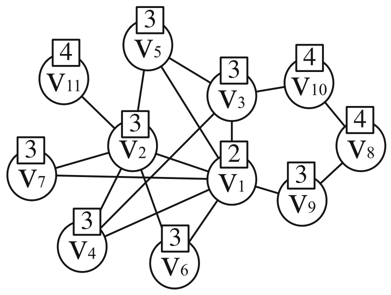

Fig. 10 Original graph G

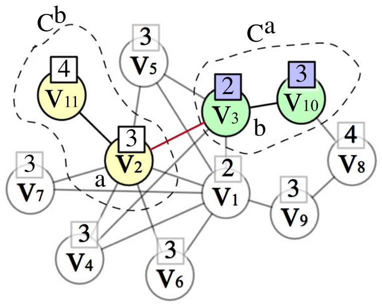

Fig. 11 Updated graph G'

$C^a = \{v \in V | dist'(v,a) \neq dist(v,a)\}$ and $C^b = \{v \in V | dist'(v,b) \neq dist(v,b)\}$.

Example 13 Consider inserting an edge $(a,b) = (v_2, v_3)$ to the graph G in Fig. 10 to form the updated graph G' in Fig. 11. We mark in green the nodes in C^a and yellow the nodes in C^b. The nodes whose distances to $a = v_2$ change are in $C^a = \{v_3, v_{10}\}$. For $b = v_3$, $C^b = \{v_2, v_{11}\}$.

On undirected graphs, w.l.o.g., we assume $|C^a| \leq |C^b|$. Denote by $\overline{C^a} = V \setminus C^a$ the complement of C^a.

Lemma 14 *For two nodes $u, v \in V$, if $dist'(u,v) \neq dist(u,v)$, then either $u \in C^a$ and $v \in C^b$ or $u \in C^b$ and $v \in C^a$. Besides, C^a and C^b are disjoint.*

Proof Please find the proof in "Appendix F". □

To accelerate the maintenance of *node closeness*, Yen et. al. [33] also made an observation that the sizes of the two sets C^a and C^b have a large gap. Recall that we assume $|C^a| \leq |C^b|$ without loss of generality.

Observation 1 ([33]). $|C^a| \ll |C^b|$.

We verified this observation on small-world networks in five categories: Youtube (social networks), HepPh (collaboration networks), Superuser (interaction networks), Wiki-talk (communication networks), and Google (Web graphs). For each graph, the average of $\frac{|C_a|}{n}$ and $\frac{|C_b|}{n}$ was obtained by repeating the following process 100 times:

Table 2 $|C^a| \ll |C^b|$ on small-world networks

	n	m	$\frac{\lvert C^a\rvert}{\lvert n\rvert}$	$\frac{\lvert C^b\rvert}{\lvert n\rvert}$
Youtube	1,134,890	2,987,624	1.16E−04	1.92E−01
HepPh	11,204	117,619	8.93E−05	4.02E−01
Superuser	189,191	712,870	3.91E−05	1.73E−01
Wiki-talk	2,388,953	4,656,682	1.85E−05	1.19E−01
Google	266,388	2,228,348	1.85E−05	8.46E−02

– select an edge e uniformly at random from the edge set E of the graph G,
– delete e from E to generate G' and then get the sizes of $|C^a|$ and $|C^b|$.

The dramatic gap between $|C^a|$ and $|C^b|$ can be observed in Table 2. For example, $|C^b|$ is about 4431 times larger than $|C^a|$ on the graph of Superuser. Note that it suffices to consider only edge deletion since the same C^a and C^b will be identified if one swaps G and G' by reversing the edge deletion to edge insertion.

By using Lemma 14 and Observation 1, closenesses can be efficiently maintained [15,25,33]. Unfortunately, the additive nature of the closeness definition which enables the efficient maintenance no longer exists when it comes to the definition of eccentricity. Specifically, for a node u, its closeness (and eccentricity, resp.) is defined as the summation (and maximization, resp.) of the shortest distances from u to all the other nodes. According to Observation 1, $|C^a|$ is relatively small, and thus, computing single-source shortest distance from each node in C^a is inexpensive. Then, we can update the closeness for all nodes, especially for the nodes $u \in C^b$: the summation $\Sigma_{v \in \overline{C^a}} dist(u,v) = \Sigma_{v \in \overline{C^a}} dist'(u,v)$ can be computed by subtracting $\Sigma_{v \in C^a} dist(u,v)$ from the old closeness of u, which, combined with the updated $\Sigma_{v \in C^a} dist'(u,v)$, produces the updated closeness of u directly. Such a method, however, fails when it comes to eccentricity, that is, $\max_{v \in \overline{C^a}} dist(u,v)$ can hardly be computed without computing the distances from u to all nodes in $\overline{C^a}$, and thus, the cost of eccentricity maintenance is still high.

In the following two sections, we first coarsely scope the maintenance dirty nodes and then identify rules to directly determine the updated eccentricity of nodes in the scope and then facilitate a more targeted recomputation.

4.2 Scope the maintenance dirty nodes

An exact identification of $C^{\mathsf{m-dirty}}$ can be achieved by maintaining, for each node u, the supporting set of u: $\{v | dist(u,v) = ecc(u)\}$. Note that, an edge insertion will not decrease $ecc(u)$ unless all nodes in the supporting set have their distances to u decreased. The cost for recording and maintaining the supporting set for each node can be extremely expensive. Instead of identifying the maintenance dirty nodes, this section considers scoping the maintenance dirty nodes, that is, to find a superset of $C^{\text{m-dirty}}$.

The next lemma shows that if a node has its distances to both a and b unaffected by the update, and then, it is not maintenance dirty.

Lemma 15 *$C^{\text{m-dirty}} \subseteq C^a \cup C^b$, that is, for node $v \in V$, if $dist'(v,a) = dist(v,a)$ and $dist'(v,b) = dist(v,b)$, then $ecc'(v) = ecc(v)$.*

Proof According to Lemma 14, for a node $u \notin C^a \cup C^b$, $dist(u,v) = dist'(u,v)$ for any node $v \in V$, and thus, $ecc(u) = ecc'(u)$. □

Example 14 Suppose $G' = (V, E \cup (v_2, v_3))$ for graphs in Fig. 11. $C^{\text{m-dirty}}$ only includes node v_3 and v_{10}, whose eccentricity changes. $C^a = \{v_3, v_{10}\}$, $C^b = \{v_2, v_{11}\}$ since their distance changes with $a = v_2$ and $b = v_3$, respectively. Thus, $(C^a \cup C^b) = \{v_3, v_{10}, v_2, v_{11}\} \supset C^{\text{m-dirty}} = \{v_3, v_{10}\}$. Moreover, only distances between the node pairs from $\{v_3, v_{10}\}$ and $\{v_2, v_{11}\}$ change.

By performing a BFS for each node in $C^a \cup C^b$, we can obviously update the eccentricities of all nodes. However, spending $O((|C^a \cup C^b|) \cdot m)$ time for a single edge update is not appealing.

4.3 Refine the eccentricity update

This section provides rules to update the eccentricities of nodes in $C^a \cup C^b$. In the case of edge insertion, we only need to recompute the eccentricity for a small portion of nodes in C^b (less than 10^2 in our empirical studies) and in the case of edge deletion, eccentricity maintenance is solely rule based, which is extremely efficient.

Effective rules are obtainable since according to Observation 1, a majority of nodes in $C^a \cup C^b$ are contributed by C^b. To avoid the BFS computation over C^b, we compute single-source shortest distances from each node in C^a before and after the update. For each node $v \in C^b$, we then obtain partial eccentricities $pecc(v|C^a)$ and $pecc'(v|C^a)$, respectively, on G and G'. It then remains to identify rules to determine the eccentricity $ecc'(v)$ for each node v in C^b based on $pecc(v|C^a)$ and $pecc'(v|C^a)$, the partial eccentricities of v on C^a.

Example 15 Let G be the graph in Fig. 11 and $G' = (V, E \setminus (v_2, v_3))$ be the graph in Fig. 12. Conduct BFS from each node in $C^a = \{v_3, v_{10}\}$ on G'. Then, for each node in C^b, we can find its partial eccentricity: $pecc'(v_2|C^a) = max(dist'(v_2, v_3), dist'(v_2, v_{10})) = 3$ and $pecc'(v_{11}|C^a) = max(dist'(v_{11}, v_3), dist'(v_{11}, v_{10})) = 4$.

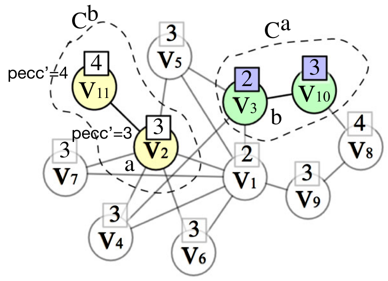

Fig. 12 $pecc'(v|C^a)$

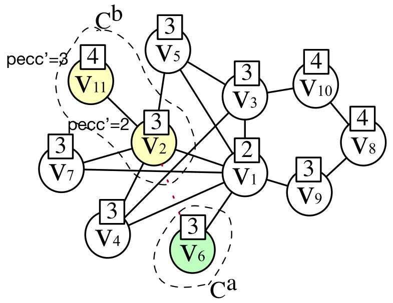

Fig. 13 D-Rule: Lemma 16

4.3.1 Update eccentricities upon an edge deletion

This section serves as a proof to Theorem 7.

Theorem 7 *When an edge $e(a,b)$ is deleted from G, the eccentricity distribution can be maintained in $O(n + m + |C^a|m)$ time complexity.*

Lemma 16 *For a node $v \in C^b$, if $pecc'(v|C^a) < ecc(v)$, then $ecc'(v) = ecc(v)$.*

Proof According to Lemma 14, the distances from v to all nodes in $\overline{C^a}$ remain unchanged, that is, the partial eccentricity $pecc(v|\overline{C^a}) = pecc'(v|\overline{C^a})$. If $pecc'(v|C^a) < ecc(v) = \max\{epcc(v|C^a), pecc(v|\overline{C^a})\}$, since $pecc(v|C^a) \leq pecc'(v|C^a)$ in deleting an edge, we have $pecc(v|C^a) < ecc(v)$, and therefore, $ecc(v) = pecc(v|\overline{C^a}) = pecc'(v|\overline{C^a})$. Thus, $ecc'(v) = \max\{pecc'(v|C^a), pecc'(v|\overline{C^a})\} = pecc(v|\overline{C^a}) = ecc(v)$. □

Example 16 In Fig 13, $G' = (V, E \backslash (v_2, v_6))$ and $C^a = \{v_6\}$, $C^b = \{v_2, v_{11}\}$. We first conduct BFS on C^a to gain the distance information. For $v_2 \in C^b$, we find that $pecc'(v_2|C^a) = 2 < ecc(v_2) = 3$; therefore, the eccentricity of v_2 does not change, and thus, $ecc'(v_2) = ecc(v_2) = 3$.

Lemma 17 *If $pecc'(v|C^a) = ecc(v)$, then $ecc'(v) = ecc(v)$.*

Proof Since $ecc'(v) = \max\{pecc'(v|C^a), pecc'(v|\overline{C^a})\}$, $pecc'(v|\overline{C^a}) = pecc(v|\overline{C^a}) \leq ecc(v)$ and $pecc'(v|C^a) = ecc(v)$, $ecc'(v) = ecc(v)$.

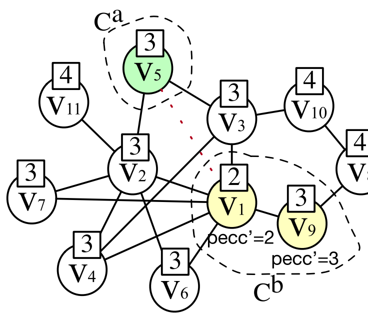

Fig. 14 D-Rule: Lemma 17

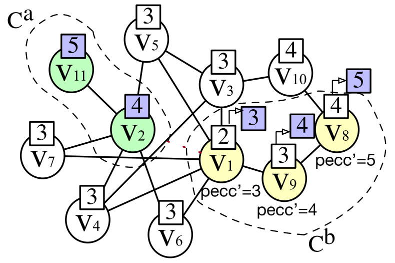

Fig. 15 D-Rule: Lemma 18

Example 17 In Fig. 14, $G' = (V, E \setminus (v_1, v_5))$ and $C^a = \{v_5\}$, $C^b = \{v_1, v_9\}$. For $v_1 \in C^b$, we find that $pecc'(v_1|C^a) = 2 = ecc(v_1)$; therefore, the eccentricity of v_1 does not change, and thus, $ecc'(v_1) = ecc(v_1) = 2$.

Lemma 18 *If $pecc'(v|C^a) > ecc(v)$, then $ecc'(v) = pecc'(v|C^a)$.*

Proof Since $ecc'(v) = \max\{pecc'(v|C^a), pecc'(v|\overline{C^a})\}$, $pecc'(v|\overline{C^a}) = pecc(v|\overline{C^a}) \leq ecc(v)$ and $pecc'(v|C^a) > ecc(v)$, $ecc'(v) = pecc'(v|C^a)$. □

Example 18 In Fig. 15, $G' = (V, E \setminus (v_1, v_2))$ and $C^a = \{v_2, v_{11}\}$, $C^b = \{v_1, v_8, v_9\}$. We first conduct BFS on C^a to gain the distance information. For $v_1 \in C^b$, we find that $pecc'(v_1|C^a) = 3 > ecc(v_1) = 2$; therefore, the eccentricity of v_1 changes to $pecc'(v_1|C^a)$, and thus, $ecc'(v_1) = pecc'(v_1|C^a) = 3$.

Remarks 3 In maintaining the eccentricity distribution when deleting edge $e(a, b)$ from G, it suffices to compute the shortest distance from a, b, and each node in C^a, respectively, using BFS.

4.3.2 Update eccentricities upon an edge insertion

Edge insertion triggers more complicated situation on C^b — updating solely based on the partial eccentricities on C^a is no longer feasible. This section aims at determining $ecc'(v)$ for most $v \in C^b$ in a rule-based approach while recomputing the eccentricities for a small subset $C^{b'} \subseteq C^b$. Our imperial study will show that the size of $C^{b'}$ can be reduced to a level less than 10^2.

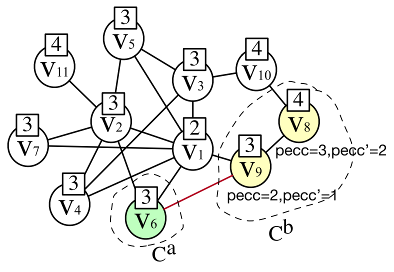

Fig. 16 I-Rules: Lemmas 19 and 21

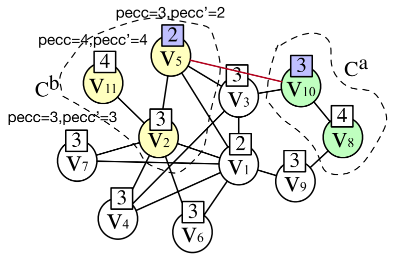

Fig. 17 I-Rule: Lemma 20

Example 19 In Fig. 16, $G' = (V, E \cup (v_6, v_9))$. $C^a = \{v_6\}$, and $C^b = \{v_8, v_9\}$. In G, the largest distance between v_9 and nodes in C^a is 2 while in G', the largest distance is 1. Therefore, $pecc(v_9|C^a) = 2$ and $pecc'(v_9|C^a) = 1$.

Lemma 19 *If $pecc(v|C^a) < ecc(v)$, then $ecc'(v) = ecc(v)$.*

Proof For a node $v \in C^b$, if its eccentricity $pecc(v|C^a) < ecc(v)$, then we know $ecc(v) = pecc(v|\overline{C^a})$ while the latter partial eccentricity is not affected by the edge update (Lemma 14). In this sense, the updated eccentricity $ecc'(v) = ecc(v)$. □

Example 20 In Fig. 16, $G' = (V, E \cup (v_6, v_9))$. $C^a = \{v_6\}$, and $C^b = \{v_8, v_9\}$. For node v_8, $pecc(v_8|C^a) = 3 < ecc(v_8)$; therefore, the eccentricity of v_8 does not change and $ecc'(v_8) = ecc(v_8) = 4$.

Lemma 20 *If $pecc(v|C^a) = pecc'(v|C^a) = ecc(v)$, then $ecc'(v) = ecc(v)$.*

Proof $ecc(v) = pecc(v|C^a)$ indicates that $pecc(v|\overline{C^a}) \leq pecc(v|C^a)$. Since $pecc(v|\overline{C^a}) = pecc'(v|\overline{C^a})$, $ecc'(v) = \max\{pecc'(v|C^a), pecc'(v|\overline{C^a})\} = pecc(v|C^a) = ecc(v)$. □

Example 21 In Fig. 17, $G' = (V, E \cup (v_5, v_{10}))$. $C^a = \{v_8, v_{10}\}$, and $C^b = \{v_2, v_5, v_{11}\}$. In G, the largest distance between v_{11} and nodes in C^a is 4 and in G', the largest distance is 4. Both of the distances equal to $ecc(v_{11})$ Therefore, $ecc'(v_{11}) = ecc(v_{11}) = 4$.

When $pecc(v|C^a) = ecc(v)$ and $pecc'(v|C^a) < ecc(v)$, it is hard to decide whether $ecc'(v) = ecc(v)$ or not. Here, we show the last rule to avoid a BFS search.

Lemma 21 *For $v \in C^b$, let p be a neighbor of v, that is, $(v, p) \in E$. If $ecc'(p) > ecc(v)$, then $ecc'(v) = ecc(v)$.*

Proof Under edge insertion, $ecc'(v) \leq ecc(v)$. Since $ecc'(v) \in [ecc'(p) - 1, ecc'(p) + 1]$ and $ecc'(p) > ecc(v)$, we have $ecc'(v) = ecc(v)$. □

Example 22 In Fig. 16, $G' = (V, E \cup (v_6, v_9))$. $C^a = \{v_6\}$, and $C^b = \{v_8, v_9\}$. v_9 has a neighbor v_8 with $ecc'(v_8) = 4 > ecc(v_9) = 3$; therefore, the eccentricity of v_9 will not change.

Lemma 22 *Applying Lemmas 19 and 20 takes $O(|C^a| \cdot m)$ time. Applying Lemma 21 takes $O(m)$ time.*

Proof Since each node in C^b needs to check its neighbors, the total cost will be no larger than $O(m)$. □

Theorem 8 *When edge $e(a, b)$ is inserted into G, the eccentricity distribution can be maintained in $O(m+n+m(|C^a|+|C^{b'}|))$ time complexity.*

Remarks 4 The challenge of eccentricity maintenance is that the partial eccentricity $pecc(v|\overline{C^a})$ for a node $v \in C^b$ cannot be easily obtained. In inserting an edge to G, the eccentricities of nodes in $C^{b'}$ cannot be decided unless a recomputation is engaged. However, the rules in Lemmas 19–21 screen a large number of nodes from C^b. Our empirical study indicates that the size of $C^{b'}$ is reduced to a level which is no more than 10^2.

Algorithm 6: ECC-DY

```
   Input: Graph $G(V, E)$,$e(a, b)$, $ecc(u)$, $\forall u \in V$
   Output: $ecc'(u)$, $\forall u \in V$
1  Compute $C^a$ and $C^b$ by performing BFS from $a$ and $b$ on $G$ and $G'$,
   respectively;
2  for each $v \in V \setminus (C^a \cup C^b)$ do $ecc'(v) \leftarrow ecc(v)$;
3  for each node in $C^a$ do perform BFS on $G$ and $G'$ ;
4  for each $v \in C^a$ do $ecc'(v) \leftarrow max_{u \in V}(dist'(v, u))$ ;
5  for each node $v \in C^b$ do
6      Compute the partial eccentricity $pecc(v|C^a)$ on $G$ and $pecc'(v|C^a)$ on $G'$,
       respectively;
7  if $e(a, b)$ is deleted from $G$ then
8      for each $v \in C^b$ do
9          Apply Lemmas 16–18 to $ecc'(v)$;
10 if $e(a, b)$ is inserted into $G$ then
11     $C^{b'} \leftarrow \emptyset$;
12     for each $v$ in $C^b$ do
13         if none of Lemmas 19–21 applies to $v$ then
14             Add $v$ to $C^{b'}$;
15         else
16             Apply Lemmas 19–21 to $ecc'(v)$;
17     for each $v$ in $C^{b'}$ do
18         Conduct BFS to get the eccentricity;
19 return the updated eccentricities $ecc'(u)$, $\forall u \in V$
```

4.3.3 Eccentricity maintenance

We conclude this section with Algorithm 6 of eccentricity maintenance. We first obtain set C^a and C^b by performing BFS from a and b, respectively (Line 1). Perform BFS from each node in C^a (Line 3) (i) to update the eccentricities of nodes in C^a (Line 4) and (ii) to provide the partial eccentricities for nodes in C^b on C^a (Lines 5–6). After that, the deletion (Lines 7–9) and insertion (Lines 10–18) are separately handled according to Sects. 4.3.1 and 4.3.2, respectively.

5 Related work

Section 5.1 studies eccentricity computation on static graphs, while Sect. 5.2 shows the results on eccentricity maintenance.

5.1 Eccentricity computation

Exact eccentricity A straightforward method to compute the exact eccentricity for all nodes is to apply all-pairs shortest path (APSP) algorithms or to pose pair-wise shortest distance (PWSD) queries quadratic times. These algorithms, however, require a high time complexity and thus are impractical to handle large graphs [14]. Although optimization strategies are proposed [4,29], their approaches still cannot scale to handle large real-world graphs. An efficient approach to the PWSD problem is called Pruned landmark labeling (PLL) [3]; its detail has been introduced in Sect. 2.2.

In the literature, to compute the exact eccentricity, Henderson [13] speeds up the computation by making use of articulation points and eccentricity bounds. The state-of-the-art algorithm is proposed by Takes et al. [27], which has been introduced in Sect. 3.1 in details. Borassi et al. [6] focus on the "directed" aspect of the diameter/radius computation on directed graphs; their techniques fall into the framework of [27] when it comes to undirected scenarios.

As a related problem, graph diameter is defined as the maximum eccentricity among all nodes. A pruning-based method to compute the graph diameter is introduced in [28], and the method is further improved by Akiba et al. [2] using eccentricity bounds propagation.

Approximate eccentricity In the literature, because of the huge computational cost for exact eccentricity, several approaches focus on computing approximate eccentricities with error bounds. A straightforward approach is to adopt the approximate APSP [1]. However, this method does not consider the properties involved in eccentricity. Roditty et al. [23] present an algorithm to estimate eccentricity $\widetilde{ecc}(v)$ using sampling. $ecc(v)$ is bounded by $[\frac{2}{3}\widetilde{ecc}(v), \frac{3}{2}\widetilde{ecc}(v)]$, for each node v in an undirected and unweighted graph. The time complexity is $O(m\sqrt{n \log n})$. The method is further improved by Chechik et al. [8] by transforming the graph to a bounded-degree graph. $ecc(v)$ is bounded by $[\widetilde{ecc}(v), \frac{5}{3}\widetilde{ecc}(v)]$. The complexity is $O((m \log m)^{\frac{3}{2}})$.

When error bound is not required, approximate eccentricities can be computed more efficiently. Takes and Kolsters [27] follow Algorithm 1 (Sect. 3.1): instead of determining the eccentricities for all nodes (Line 3), [27] only aims at determining the diameter and radius. In other words, the search halts when the eccentricities are determined for the nodes with either the largest eccentricity upper bound or the smallest eccentricity lower bound. Another approach [26] estimates the eccentricities by computing the shortest distances from $2k$—k is a parameter—selected nodes in two phases. Phase-1 picks a set S of k nodes sourced from which the shortest distances are computed. Phase-2 chooses a set S' of k nodes with the maximum, among all graph nodes, the maximum distance to S, and then computes the shortest distances from S'. Each node estimates its eccentricity as its maximum distance to $S \cup S'$. The accuracy, though is normally high, is controlled by k and subjected to individual graph properties (as can be seen in our experiments). In particular, when it comes to a specific node, without an error bound, the estimation with a chance of a high relative error (due to the small diameter) can hardly be trusted and doubtlessly engaged.

Other graph centrality measures In addition to graph eccentricity, there are some other famous graph centrality measures. For example, closeness centrality, which is the inverse of the average shortest distance from the vertex to any other vertex in the graph [21], is useful to measure the efficiency of each vertex in spreading information to all other vertices. Betweenness centrality, which is the fraction of shortest paths between node pairs that pass through the target node [20], is used to measure the ability of a node to control the information flow between other nodes. A recent survey of graph centrality measures and their application in different domains can be found in [18].

5.2 Eccentricity maintenance

The problem of maintaining the eccentricity distribution on dynamic graphs, as far as we know, has not been studied in the literature. A straightforward solution is to resort to dynamic APSP algorithms, the classic algorithms in which has been captured in a survey [9]. Dynamic APSP algorithms, however, necessitate expensive yet unnecessary space and time cost to maintain the distance information—eccentricity only tracks the maximum distance, as opposed to all the distances, from a node to all the other nodes.

Diameter maintenance A related topic is to maintain the diameter of a dynamic graph; however, existing work [10,24] only considers insertion-only graphs. The proposed algorithms [10,24] keep the set of node pairs whose distances equal the diameter and update the node-pair set when a new node is added. Their inefficiency is because i) the cardinality of the node-pair set kept can be large and ii) the cost of refreshing the distances of the node pairs in the set can be expensive. Besides, these approaches cannot handle edge deletions.

Centrality maintenance The problem of maintaining centrality measures on dynamic graphs has been studied on, apart from closeness centrality [33], a wide spectrum of centrality definitions such as betweenness centrality [22], harmonic closeness centrality [5], and personalized centrality [19]. Their solutions vary dramatically based on the properties of the centrality definition. For eccentricity centrality, the most related definition is closeness centrality [33] for which the state-of-the-art solution has been introduced in Sect. 4.

6 Experiments

This section evaluates the performance of the eccentricity computation and maintenance solution proposed in the paper. We first introduce the experiment setting and then show the results on eccentricity computation in Sect. 6.1 and maintenance in Sect. 6.2.

Datasets Our experiments were conducted on 20 real-world graphs with various properties. The first seven graphs are online social networks, and the following four graphs are collaboration networks. Askubuntu, Mathoverflow, and Superuser are interaction networks on the Stack Exchange Web site—nodes represent users and edges indicate the answer and comment relationships. Wiki-temporal and Wiki-talk are communication networks. Skitter is the computer network, while Google, In, and Indochina are Web graphs. All graphs are considered as undirected and connected graphs: if a graph is not connected, we used the largest connected component of the graph. The details, which are the total number of nodes n, the total number of edges m, radius r, and diameter d, of all graphs are presented in Table 3. Among these graphs, the largest diameter is 43 and the smallest is 7. The average label size for PLL is no larger than 100 on most graphs.[3] All graphs were downloaded from Stanford Large Network Dataset Collection[4] [16] and Laboratory for Web Algorithms.[5] To better explain the results, we use four graphs, Slashdot, Twitter, DBLP, and Wiki-talk, to *represent* the large graphs in some experiments while showing the results of the other graphs in "Appendix J".

Eccentricity computation algorithms We compare our proposed algorithms against the state-of-the-art algorithm

[3] The label size may differ during every execution of the PLL approach due to the randomness in determining the node order [3].

[4] http://snap.stanford.edu/data/.

[5] http://law.di.unimi.it/datasets.php.

Table 3 Dataset description and comparison with the state-of-the-art methods

Dataset	Statistical information				Proposed algorithm ECC (s)				Baseline (s)		PLL
	n	m	r	d	Labeling	RefPool	Eccentricity	Total	BoundEcc	BoundPLL	Avg label size
Brightkite	56,739	212,945	9	18	0.95	0.14	0.07	1.16	88.99	664.43	64.72
Epinions	75,877	405,739	8	15	1.07	0.20	0.14	1.41	20.09	116.25	59.19
Gowalla	196,591	950,327	8	16	8.09	0.66	0.20	8.95	395.00	2559.31	91.54
Slashdot	77,360	469,180	6	12	1.77	0.22	0.29	2.28	86.40	516.46	66.90
Twitter	81,306	1,342,296	4	7	3.05	0.24	36.80	40.09	166.15	548.51	70.20
Youtube	1,134,890	2,987,624	12	24	79.75	4.35	5.35	89.45	23, 216.60	–	115.03
Lastfm	1,191,805	4,519,330	5	10	464.24	5.96	1263.62	1733.8	29, 039.06	–	295.96
AstroPh	17,903	196,972	8	14	0.40	0.03	0.14	0.57	4.14	26.97	75.03
CondMat	21,363	91,286	8	15	0.30	0.05	0.06	0.41	4.39	37.69	68.87
DBLP	317,080	1,049,866	12	23	78.07	1.29	9.66	89.02	774.18	14,865.80	263.66
HepPh	11,204	117,619	7	13	0.16	0.02	0.08	0.26	1.40	5.93	63.23
Askubuntu	152,599	453,221	7	13	1.78	0.60	0.27	2.65	33.28	172.99	54.56
Mathoverflow	24,668	187,939	5	9	0.22	0.05	0.02	0.29	4.63	18.62	51.80
Superuser	189,191	712,870	6	12	2.91	0.58	0.33	3.82	295.93	1519.66	57.38
Wiki-talk	2,388,953	4,656,682	6	11	36.43	8.13	12.67	57.23	1187.17	12,245.40	61.68
Wiki-temporal	1,091,742	2,786,764	5	9	12.39	3.43	9.60	25.42	4857.36	29,522.60	56.02
Skitter	1,694,616	11,094,209	16	31	360.75	7.91	14.03	382.69	8211.21	–	187.78
Google	855,802	4,291,352	12	24	97.51	3.03	19.38	119.92	12, 820.50	–	137.47
In	1,353,703	13,126,172	22	43	114.01	5.13	62.70	181.84	723.51	13,650.20	167.22
Indochina	7,320,539	149,054,854	22	43	4757.55	41.76	1101.02	5900.33	30, 638.80	–	413.71

BoundEcc [27] for exact eccentricity computation. The source code is downloaded from the author's webpage.[6] All parameters were set to their default values. Moreover, we plugged the distance labeling algorithm PLL into BoundEcc by replacing all online distance queries with the distance labeling queries and the final algorithm is called BoundPLL.

Our techniques include the following two methods:

- ECC: Invoke EccentricityOneNode for each node in the graph. (Algorithm 2 in Sect. 3.3).
- ECC-LS: Update the eccentricity bounds using the local spread technique. (Algorithm 5 in Sect. 3.4).

To better analyze the result, we will show the cost of distinct *phases* of ECC as well. The phases include:

- Labeling: build the PLL structure of the graph [3].
- RefPool: compute the reference-node pool, the lists L_z for each z in the pool, and the initial $\underline{ecc}$ and $\overline{ecc}$ for each node of the graph (Algorithm 3 in Sect. 3.3.2).
- Eccentricity: compute the eccentricity distribution.

Note that ECC-LS shares the first two phases with ECC while differs with ECC on the third phase.

Eccentricity maintenance algorithms We also compare our proposed maintenance algorithm ECC-DY (see Algorithm 6) against the baseline approach Recomp which triggers ECC-LS upon each edge update.

Environment and settings All algorithms were implemented in C++ and compiled with GNU GCC 4.8.5 and -O3 level optimization. All experiments were conducted on a machine with an Intel Xeon 2.3 GHz CPU and 128 GB main memory running Linux (Red Hat Linux 4.8.5, 64bit). The cost was evaluated in the wall clock time, and the cutoff time was set to 24 h. The costs of the three phases were evaluated, respectively. By default, the number of reference nodes was set to be $k = 16$; otherwise, the varying number k of reference nodes ranged from 1 to 32.

6.1 Eccentricity computation

Exp-1: Comparison with the state of the art This experiment compares our algorithm ECC with BoundEcc and BoundPLL. The results are shown in Table 3.

Table 3 indicates that our method ECC outperforms BoundEcc on all types of graphs by up to two orders of magnitude over BoundEcc and up to three orders of magnitude over BoundPLL. For example, On Wiki-temporal, ECC is more than 191 times faster than BoundEcc and 1161 times faster than BoundPLL. For Youtube and Lastfm, our ECC

[6] https://franktakes.nl/research/.

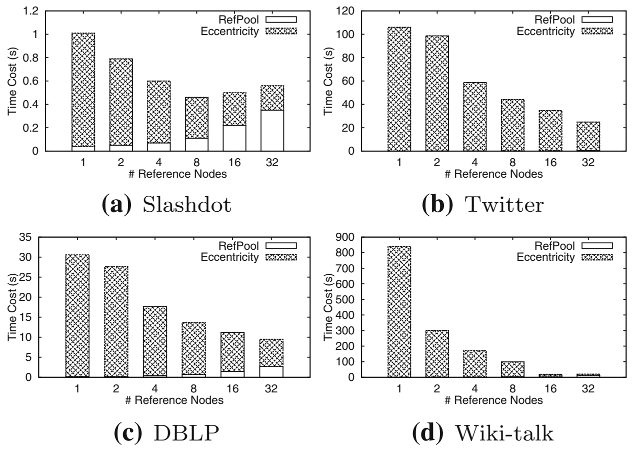

Fig. 18 Testing ECC (varying # reference nodes)

had completed the eccentricity computation in, respectively, 2 min and half an hour, while BoundEcc needs > 6 h and BoundPLL cannot finish the computation within 24 h.

Second, we observe that, in the three phases of our proposed algorithm ECC, the time for labeling dominates the overall cost for most of the graphs. The actual time used for eccentricity is within 2 min on all of the graphs except Lastfm and Indochina. This means that ECC can be scaled to handle larger graphs upon an accelerated labeling method. The results in Table 3 demonstrate that our proposed method is superior to the state-of-the-art methods.

Exp-2: Testing ECC This experiment shows the performance of ECC under a varying number k of reference nodes. k ranges from 1 to 32. Since the time for Labeling is independent of the number of reference nodes, we only report the processing time for RefPool and Eccentricity. We show the experimental results for four representative large graphs in Fig. 18 while leaving the results on the other graphs in Fig. 29 in "Appendix J".

Figure 18 indicates that the time for RefPool increases with an increasing k, and the time for Eccentricity decreases with the increasing k. k reference nodes incur k BFSs, and thus, the time of RefPool increases with k. The efficiency of Eccentricity for one node u is dependent on the distance $dist(u, z)$ from u to its reference node z (Theorem 4) while an enlarged reference-node pool decreases $dist(u, z)$. Therefore, increasing k reduces the cost of Eccentricity.

Figure 18 also shows a trade-off between RefPool and Eccentricity in ECC. For Slashdot and Wiki-talk, the running time first decreases and then increases with an increasing k; for Twitter and DBLP, the running time drops with an increasing k. The results over all graphs suggest that $k = 16$ is a reasonable number of reference nodes to balance the cost for RefPool and Eccentricity.

Exp-3: Testing ECC-LS We examine the local spread technique by comparing ECC with ECC-LS. Since ECC and ECC-LS

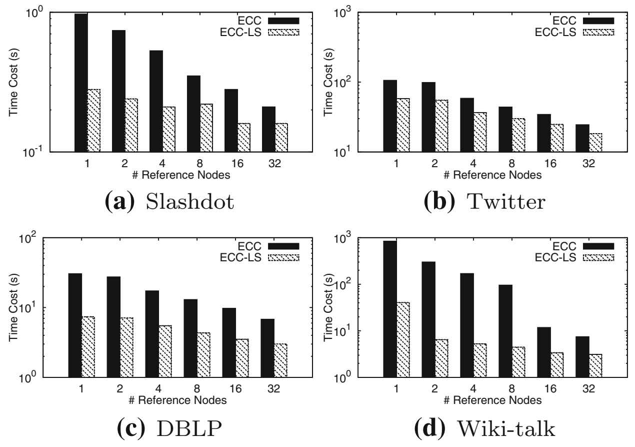

Fig. 19 Testing ECC-LS (processing time for Eccentricity)

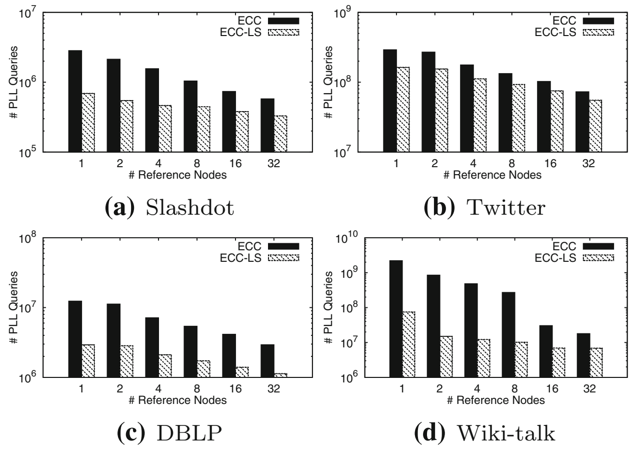

Fig. 20 Testing ECC-LS (# PLL queries)

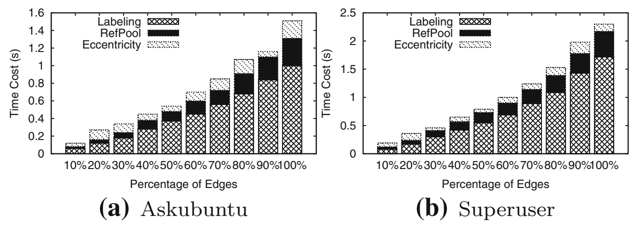

Fig. 21 Scalability testing

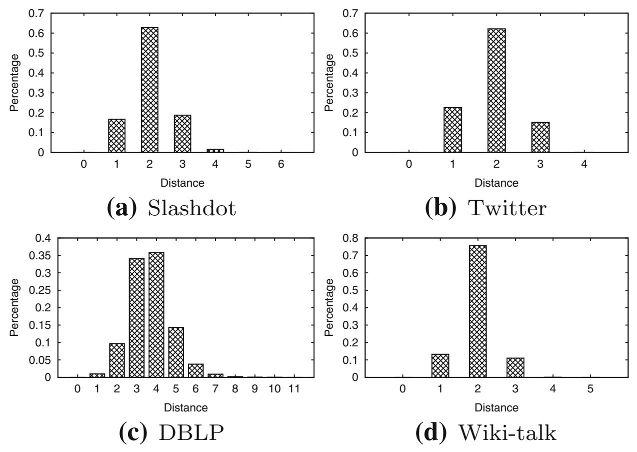

Fig. 22 Testing distribution of distance to reference nodes

have the same costs for Labeling and RefPool, we only report the cost of the third phase—Eccentricity. We show the results on four representative graphs in Figs. 19 and 20 while leaving the results on the other graphs in Figs. 30 and 31 in “Appendix J”.

Figure 19 shows the processing time when varying the number k of reference nodes. The processing time for both ECC and ECC-LS increases with an increasing k. Besides, local spread speeds up the Eccentricity by tightening the eccentricity bounds: ECC-LS outperforms ECC on Eccentricity by a factor of 2 to 10.

We also report the number of PLL queries for ECC and ECC-LS in Fig. 20 when varying the number of reference nodes. The trend is similar to that for the processing time in Fig. 19. For example, for the DBLP dataset, when the number of reference nodes is 4, ECC is 3 times slower than ECC-LS while the number of PLL queries for ECC is 3 times larger than ECC-LS. This shows that the processing time for Eccentricity is proportional to the number of PLL queries.

Exp-4: Scalability testing We chose temporal graphs Askubuntu and Superuser for the scalability test. Edges of a temporal graph are associated with timestamps. For each graph, edges were sorted in the ascending order of their timestamps. The first 10%, 20%, . . ., 100% of the sorted edges consisted of a series of 10 generated graphs, respectively, with increasing sizes. Each generated graph is a real-world graph with the similar graph properties as the original graph.

Figure 21 demonstrates the processing time for the three phases Labeling, RefPool, and Eccentricity, respectively, of our algorithm ECC, on the two graphs. When the graph size increases, the processing time for all three phases Labeling, RefPool, and Eccentricity increases. The three phases Labeling, RefPool, and Eccentricity share a linear trend with an increasing graph size. This indicates that ECC has a high scalability. The curves for ECC-LS in scalability are similar to those of ECC, which are not shown in the paper due to the space limitation.

Exp-5: Testing distance to reference nodes We show the results on four representative graphs in Figs. 22 and 23 while leaving the results on the other graphs in Figs. 32 and 33 in “Appendix J”.

Figure 22 shows the distribution of the distance $\lambda_0(u)$ from a node u to its reference node z. The reference node z of u is the node that is nearest to u in the reference-node pool. The number k of reference nodes was set to be 16. For DBLP, most distances fall into the range of [2, 5]. For the other three

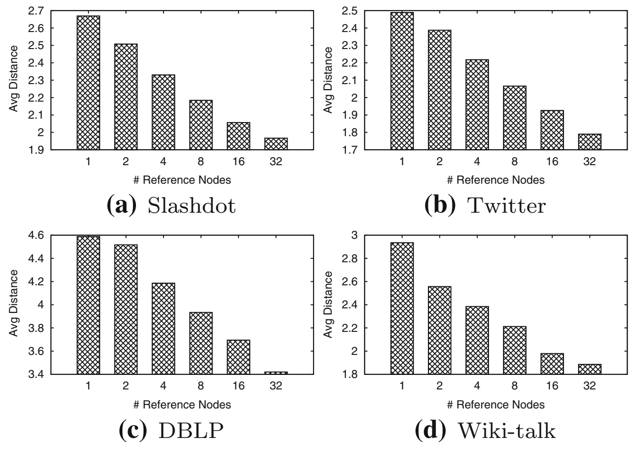

Fig. 23 Testing average distance to reference nodes

graphs, more than 60% of nodes u have $\lambda_0(u) \leq 2$; for all the nodes u in the graph, $\lambda_0(u) \leq 6$.

Figure 23 illustrates the average distance of a node to its nearest reference node when varying the number of reference nodes k from 1 to 32 on the four graphs. The average distance decreases with an increasing k on all graphs. When there is only one reference node ($k = 1$), the average distance is 4.6 for DBLP and less than 3 on the other three graphs. When $k = 32$, the average distance decreases to less than 3.5 for DBLP and less than 2 on the other three graphs. This result justifies the claim in Sect. 3.3.2.

6.2 Eccentricity maintenance

Exp-6: Testing ECC-DY To examine the performance of the proposed dynamic algorithm ECC-DY, we report the speedup of ECC-DY over Recomp. The speedup of ECC-DY is defined as the ratio of the update cost of Recomp over the update cost of ECC-DY.

We constructed an update for a data graph as follows. An edge deletion used the date graph as the original graph G, while an edge insertion used the data graph as the updated graph G'. This setting ensured that edge insertion and edge deletion were updating real edges on or to real graphs. The update edge e was uniformly selected from the edge set of the data graph.

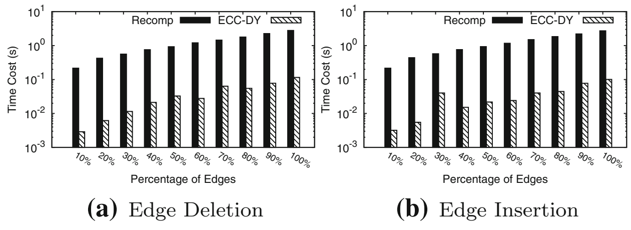

Fig. 25 Scalability testing for ECC-DY on Superuser

Each workload includes 100 updates and reports the average speedup. Figure 24 shows the result. ECC-DY takes on average 0.66% total time of Recomp on each edge deletion while takes on average 0.79% total time of Recomp on each edge insertion. The largest speedup is observed on Indochina for edge deletion and edge insertion, which is 954.90 and 863.14, respectively. Note that Recomp is using ECC-LS to recompute the eccentricity while ECC-LS beats the state-of-the-art eccentricity computation algorithms by up to three orders of magnitude. ECC-DY, which is always one or two orders faster than Recomp, therefore, justifies its superiority in maintaining the eccentricity on real dynamic graphs.

Exp-7: Scalability testing of ECC-DY We used the temporal graph Superuser to generate a sequence of real graphs by getting the snapshots of the graph on the first 10%, 20%, ..., 100% edges, respectively. The performance of ECC-DY on each snapshot was examined using workloads with 100 updates. The average cpu-time of ECC-DY and Recomp is shown in Fig. 25a for edge deletion and Fig. 25b for edge insertion, respectively.

Both Recomp and ECC-DY increase the average cpu-time almost linearly with the number of edges in the graph. But on edge deletion, ECC-DY, whose time cost is capped by 10^{-1}, is more scalable than Recomp. It is not surprising since on edge deletion, ECC-DY can update the eccentricity by computing BFS only on C^a nodes while $|C^a| \ll |C^b| \leq |V|$. Note that

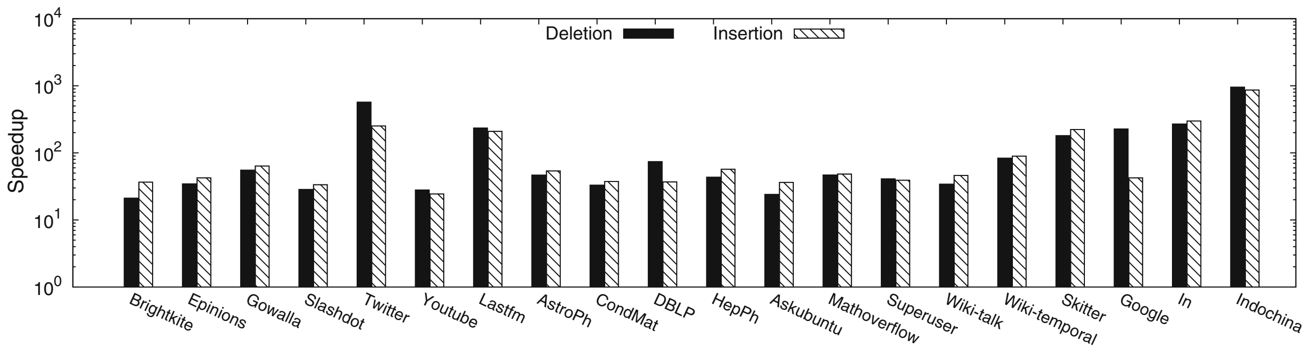

Fig. 24 Speedup of ECC-DY over ECC-LS

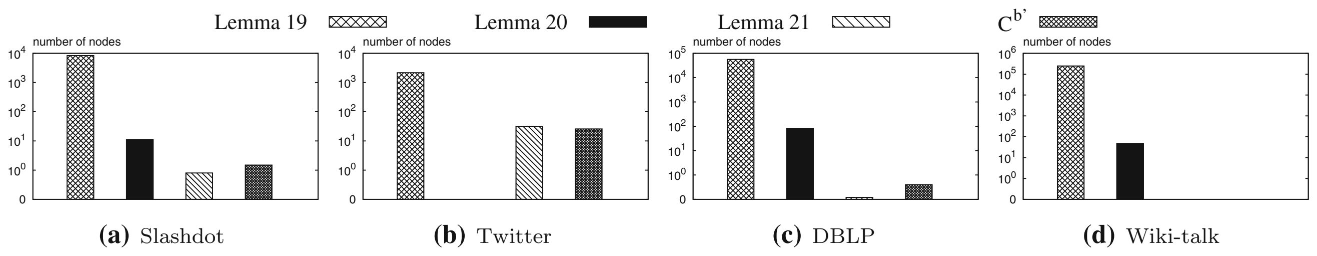

Fig. 26 Testing the average number of nodes in each step of ECC-DY for edge insertion

for ECC-DY, the time cost depends on the number of possible affected nodes. This number is related to not only the graph size but also the graph structure and the specific edge to be inserted or deleted. This explains a general increase in the maintenance cost along the increasing graph size and a mild fluctuation observed in edge insertion (20–30%) and deletion (50–80%). In all cases, the time of ECC-DY is bounded by Recomp, which confirms the superior of ECC-DY in tracking the eccentricity when the graph is continuously changing.

Exp-8: Rule-effectiveness for edge insertion Section 4 illustrated Lemmas 19–21 in determining the eccentricities for nodes in $C^b \setminus C^{b'}$. Here, we show that the size of $C^{b'}$ is small on an edge insertion workload with 100 update generated in the same way of Exp-7.

We show the number of nodes pruned by each of Lemmas 19–21 and the number of nodes in $C^{b'}$ on four representative networks in Fig. 26 while leaving the results on the other graphs in Fig. 34 in "Appendix J". The number of nodes left in $C^{b'}$ is normally small, which is no larger than 10^2 on all the tested graphs. Moreover, for Wiki-talk, the size of $C^{b'}$ is nearly to zero. This means that Lemmas 19–21 can determine the updated eccentricity for most of the nodes in the graph.

7 Conclusions

This paper provides a solution to efficient eccentricity computation and maintenance based on a spectrum of insights into the bottleneck of existing approaches. The superior efficiency has been confirmed by a comprehensive experimental evaluation. It can be observed from our evaluation results that the current bottleneck becomes the expensive construction of the pair-wise shortest distance index of PLL. Our future work will investigate this bottleneck to (i) speed up the construction of PLL using parallelism or (ii) identify a "hot spot" of the posed pair-wise shortest distance queries which can be answered by a much smaller and cheaper index.

Acknowledgements Miao Qiao is supported by Marsden Fund UOA1732, Royal Society of New Zealand. Lu Qin is supported by ARC DP160101513. Ying Zhang is supported by ARC FT170100128 and DP180103096. Lijun Chang is supported by ARC DP160101513 and FT180100256. Xuemin Lin is supported by NSFC 61672235, DP170101628 and DP180103096.

A The proof of Lemma 4

Based on Definition 3, $pecc(x|V^z_{\leq\lambda}) = \max_{u\in V^z_{\leq\lambda}} dist(x,u)$. $dist(x,u) \leq dist(x,z) + dist(z,u) \leq dist(x,z) + \lambda$, for $\forall u \in V^z_{\leq\lambda}$. Therefore, $pecc(x|V^z_{\leq\lambda}) = \max_{u\in V^z_{\leq\lambda}} dist(x,u) \leq \max_{u\in V^z_{\leq\lambda}}(dist(x,z)+\lambda) = dist(x,z)+\lambda$.

B The proof of Lemma 5

According to the definition of a λ-partial set, $V' \cup V^z_{\leq\lambda} = V$, and the definition of the eccentricity $ecc(x) = max_{u\in V} dist(u,x)$, $ecc(x) = \max\{pecc(x|V'), pecc(x|V^z_{\leq\lambda}\}$.

C The proof of Lemma 9

Let a shortest path from u to x be $\langle v_0, v_1, v_2, \ldots, v_k\rangle$ with $v_0 = u$, $v_k = x$ and $k = dist(x,u)$. Since the new snapshot is taken on a stable state, consider edges $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1},v_k)$, we have $\overline{ecc}_{new}(u) = \overline{ecc}_{new}(v_0) \leq \overline{ecc}_{new}(v_1)+1 \leq \overline{ecc}_{new}(v_2)+2 \leq \ldots \leq \overline{ecc}_{new}(v_k)+k = \overline{ecc}_{new}(x)+k$. Therefore, $\overline{ecc}_{new}(u) \leq \min\{\overline{ecc}_{old}(u), \overline{ecc}_{new}(x)+dist(x,u)\}$. Similar proof can be applied on showing that $\underline{ecc}_{new}(v_0) \geq \underline{ecc}_{new}(v_1)-1 \geq \ldots \geq \underline{ecc}_{new}(v_k)-k$. By plugging $v_0 = u$, $v_k = x$ and $k = dist(x,u)$ in the above inequality, we complete the proof.

D The proof of Lemma 10

Observe that in Step 2) of the iterative update, $\overline{ecc}(l)$ of node l will be updated only if $\overline{ecc}(r)$ of its neighbor r is small enough such that $\overline{ecc}(l) > \overline{ecc}(r)+1$. Once the update takes place, we conceptually associate with $\overline{ecc}(l)$ a source $\overline{ecc}(l).s \leftarrow r$ to record the source of the bound. Note that this source field may be overwritten upon a subsequent update; however, it will not be removed once created.

$\overline{ecc}_{new}(y).s$ exists since $\overline{ecc}(y)$ must have been updated to let $\overline{ecc}_{new}(y) < \overline{ecc}_{old}(y)$. Now, we trace from y via the source link of $\overline{ecc}_{new}(\cdot).s$, generating a path $\langle v'_0, v'_1, \ldots, v'_{k'}\rangle$ with $v'_0 = y$, $v'_i = \overline{ecc}_{new}(v'_{i-1}).s$, for each $i \in [1, k']$ while $\overline{ecc}_{new}(v'_{k'})$ does not have a source. Note that in this sequence, we have $\overline{ecc}_{new}(v'_{i-1}) = \overline{ecc}_{new}(v'_i) + 1$ for all $i \in [1, k']$; thus, the sequence cannot contain a loop and thus $k' \le n$. We have $\overline{ecc}_{new}(y) = \overline{ecc}_{new}(v'_0) = \overline{ecc}_{new}(v'_{k'}) + k'$.

We prove that $v'_{k'}$ must be the trigger node x. If otherwise, $\overline{ecc}_{new}(v'_{k'})$ has no source, and thus, $\overline{ecc}_{new}(v'_{k'}) = \overline{ecc}_{old}(v'_{k'})$. Therefore, $\overline{ecc}_{old}(y) > \overline{ecc}_{new}(y) = \overline{ecc}_{old}(v'_{k'}) + k'$. According to pigeon principle, there must be $\exists j \in [1, k']$ such that $\overline{ecc}_{old} v'_{j-1} > \overline{ecc}_{old} v'_j + 1$—violating the assumption that the old snapshot is stable.

The fact that $v'_{k'} = x$ implies three important results:

1. For v'_j with $j \in [0, k')$, $\overline{ecc}_{new}(v'_j) < \overline{ecc}_{old}(v'_j)$: if otherwise, the path would have stopped at j instead of k'.
2. $\overline{ecc}_{new}(x) = ub_x < \overline{ecc}_{old}(x)$. Since if $\overline{ecc}_{new}(x) = \overline{ecc}_{old}(x)$, there will be a violation to the assumption that the old snapshot stable. Besides, $\overline{ecc}_{new}(x)$ has no source; thus, it has not been updated in Step 2) of the iterative update. Therefore, $\overline{ecc}_{new}(x) = \min\{\overline{ecc}_{old}(x), ub_x\} = ub_x < \overline{ecc}_{old}(x)$.
3. $\langle v'_0, v'_1, \ldots, v'_{k'}\rangle$ is a shortest path from y to x. Since k' is the length of a path from x to y, $k' \ge dist(x, y)$. Based on Lemma 9, that is, $\overline{ecc}_{new}(y) \le \overline{ecc}_{new}(x) + dist(x, y)$, it can be assured that $k' = dist(x, y)$ since $\overline{ecc}_{new}(y) = \overline{ecc}_{new}(x) + k'$.

From the above three results, we complete the proof.

E The proof of Lemma 13

We only prove the lemma for the case of edge insertion since the case of edge deletion can be symmetrically proved. Let p (and p', resp.) be a shortest path from v to w in G (and in G', resp.) with length $L(p)$ (or $L(p')$, resp.). If p' does not include edge e, then both p and p' are paths between v and w in both G and G'. Since p and p' are shortest paths of G and G', respectively, $dist(v, w) = L(p) = L'(p) = dist'(v, w)$, contradiction. Therefore, p' must include e.

F The proof of Lemma 14

We only consider the case of inserting the edge of $e(a, b)$ to G since the case of edge deletion can be symmetrically proved. For two nodes $u, v \in V$, if $dist(u, v) \neq dist'(u, v)$, then all of the shortest paths from u to v on graph G' must pass e (Lemma 13), i.e., either $dist'(u, a) = dist'(u, b) - 1$ or $dist'(u, b) = dist'(u, a) - 1$. When $dist'(u, a) = dist'(u, b) - 1$, since $dist'(b, v) = dist(b, v)$ and $dist'(u, a) = dist(u, a)$ (the shortest paths from u to a and from b to v on G' do not include e), and $dist(u, v) \neq dist'(u, v)$, we have $dist(u, b) \neq dist'(u, b)$, and $dist(v, a) \neq dist'(v, a)$, $u \in C^b$ and $v \in C^a$. Similarly, when $dist(u, b) = dist(u, a) - 1$, $u \in C^a$ and $v \in C^b$.

G Comparison with approximate methods

We show that computing exact eccentricities is necessary by evaluating the approximate algorithms of HybridEcc [27] and kBFSEcc [26] on four graphs: Twitter, Youtube, Lastfm, and Indochina. The algorithms of HybridEcc and kBFSEcc have been introduced in Sect. 5; the descriptions of the four real graphs and the computation time of ECC are given in Table 5. All algorithms were run with a single thread.

We measure an approximate algorithm with its *accuracy*—the percentage of nodes whose eccentricities are correctly estimated—and its *running time*.

Table 5 shows the running time and accuracy of HybridEcc. In comparison with ECC, HybridEcc shows a trade-off between the precision and efficiency. On Twitter, HybridEcc uses longer running time to achieve a high accuracy (higher than 96%). On Indochina, HybridEcc performs well both in terms of accuracy and running time. On Youtube and Lastfm, the running time of HybridEcc is significantly lower than that of ECC while the accuracy is below 66%. HybridEcc has its accuracy varying dramatically on different graphs and thus fails in providing stable and reliable estimations.

The performance of kBFSEcc is highly dependent on a key parameter of k, and we show the accuracy of kBFSEcc in Fig. 27 and running time in Fig. 28 with k varying from 1 to $2^{14} = 16{,}384$. Note that, in order to eliminate the impact of randomness, we used the same random seed for different values of k. This means that the sampled node set S of Phase 1 (see Sect. 5) of a smaller k is a subset of that of a larger k.

Figure 27 shows the accuracy of kBFSEcc. On 2 out of 4 graphs, kBFSEcc achieves 100% accuracy with a small k: $k = 2^4$ on Youtube, $k = 2^8$ on Indochina. In contrast, on Twitter, it requires a large $k = 2^{14}$ to reach 100% accuracy while it never achieves 100% accuracy on Lastfm. A weird fluctuation of the accuracy of kBFSEcc has been observed. Note that this phenomena is random-seed independent—similar phenomena can be observed under other random seeds. On all four graphs, an increase of k can dramatically reduce the accuracy. For example, the accuracy drops from 95.59% to 39.01% on Lastfm when k is increased from 2^3 to 2^4. In this sense, kBFSEcc still needs analysis on the relation-

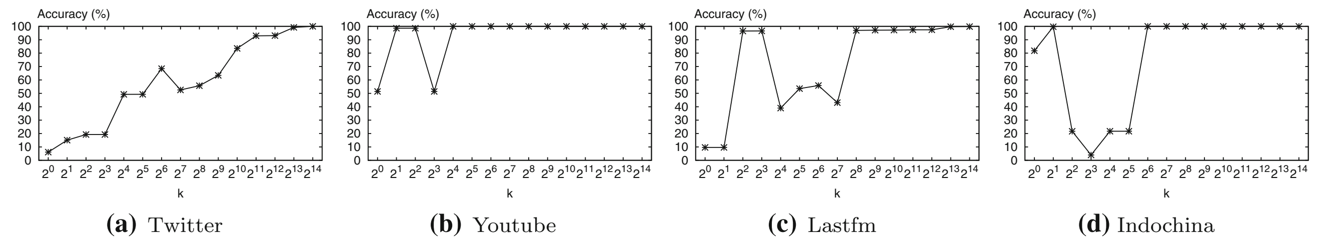

Fig. 27 Testing the accuracy of kBFSEcc

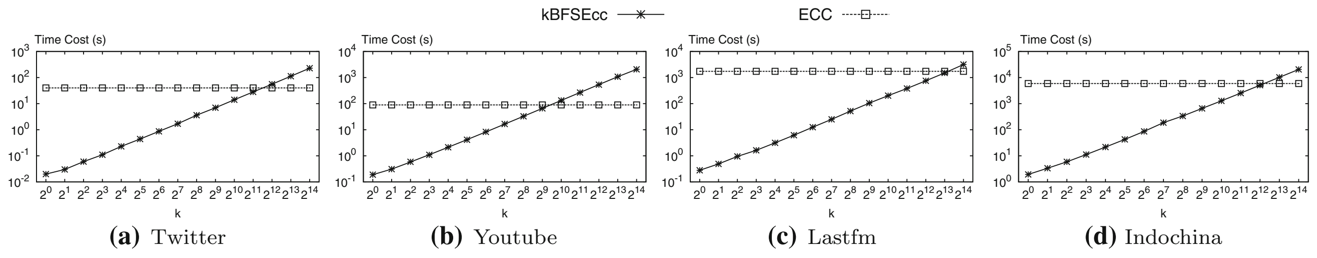

Fig. 28 Testing the running time of kBFSEcc and ECC

Table 4 Testing ECC on road networks

Dataset	*n*	*m*	*r*	*d*	Labeling (s)	RefPool (s)	Eccentricity (s)	Total (s)	BoundEcc(*s*)
Luxembourg	114,599	119,666	669	1337	5.17	0.38	104.1	109.65	30.06
Usroads	126,146	161,950	309	617	39.73	0.46	3894.79	3934.98	161.56
RoadNetPA	1,087,562	1,541,514	402	794	887.95	4.96	–	–	8789.47

Table 5 Dataset description and the results of HybridEcc

Dataset	*n*	*m*	*r*	*d*	ECC (s)	HybridEcc	
						Time (s)	Accuracy
Twitter	81,306	1,342, 296	4	7	40.1	89.0	99.2%
Youtube	1,134, 890	2,987, 624	12	24	89.5	1.8	51.4%
Lastfm	1,191, 805	4,519, 330	5	10	1733.8	110.3	65.9%
Indochina	7,320, 539	149,054, 854	22	43	5900.33	67.24	99.9%

ship between k and the accuracy to establish the reliability of the estimation.

Figure 28 shows the running time of kBFSEcc that increases linearly with k. In general, kBFSEcc reaches a high accuracy within 10% of the running time of ECC. In particular, on Indochina, kBFSEcc reached 100% accuracy 17.5× times faster than ECC. However, this superiority is not guaranteed: when ECC completed its computation on Twitter ($k = 2^8$), kBFSEcc can only achieve an accuracy of 93.10%. In conclusion, in comparison with ECC, kBFSEcc provides a faster yet less reliable estimation on the eccentricity distribution.

H Corner case on road networks

We show the applicability of our algorithm by conducting experiments on road networks whose diameters are large. Three road networks Luxembourg, Usroads, and RoadNetPA were used (downloaded from Network Repository[7]). Tables 4, 5 show the comparison of ECC and BoundEcc on road networks. ECC is slower than BoundEcc in all three graphs; ECC cannot finish the computation within one day on RoadNetPA. Since our algorithm is highly dependent on the features of small-world networks, it is not applicable to graphs with large diameters.

[7] http://networkrepository.com/networks.php.

I Detailed explanation of PLL

Algorithm 7 illustrates the main steps of the PLL approach. In iteration i, node v_i performs pruned BFS (Lines 1–4). When v_i visits a node u (Line 5), we first check whether the current labels S can answer the distance between v_i and u. If so, u is pruned and we stop the traversal from u (Lines 6–7). Otherwise, $(v_i, dist(v_i, u))$ is inserted into $S(u)$ (Line 8). Moreover, we continue BFS from u by adding the neighbors of u into the queue Q (Lines 9–12). Finally, the total set $S = \{S(v)|v \in V\}$ is returned as the answer PLL (Line 13).

Algorithm 7: The PLL Approach

```
Input: Graph $G(V, E)$
Output: The index PLL
1  for $i = 1, 2, \cdots, n$ do
2      $Q \leftarrow$ a queue with only one element $v_i$;
3      $dist(v_i) \leftarrow 0$ and $dist(v_i) \leftarrow \infty, \forall v \in V \setminus v_i$;
4      while $Q \neq \varnothing$ do
5          $u \leftarrow Q.pop()$;
6          if $Query(v_i, u, S) \leq dist(u)$ then
7              continue;
8          Insert $(v_i, dist(u))$ into $S(u)$;
9          for $w \in N(u)$ do
10             if $dist(w) = \infty$ then
11                 $dist(w) \leftarrow dist(u) + 1$;
12                 $Q.push(w)$;
13 return $\{S(v)|v \in V\}$
```

J Additional experimental results

This section shows the results on the graphs apart from Slashdot, Twitter, DBLP, and Wiki-talk in the following four experiments.

– Exp-2: Testing ECC. Figure 29 shows the performance of ECC under a varying number k of reference nodes.

– Exp-3: Testing ECC-LS. Figure 30 shows the processing time when varying the number k of reference nodes. Figure 31 shows the number of PLL queries for ECC and ECC-LS.

– Exp-5: Testing distance to reference nodes Figure 32 shows the distribution of the distance $\lambda_0(u)$ from a node

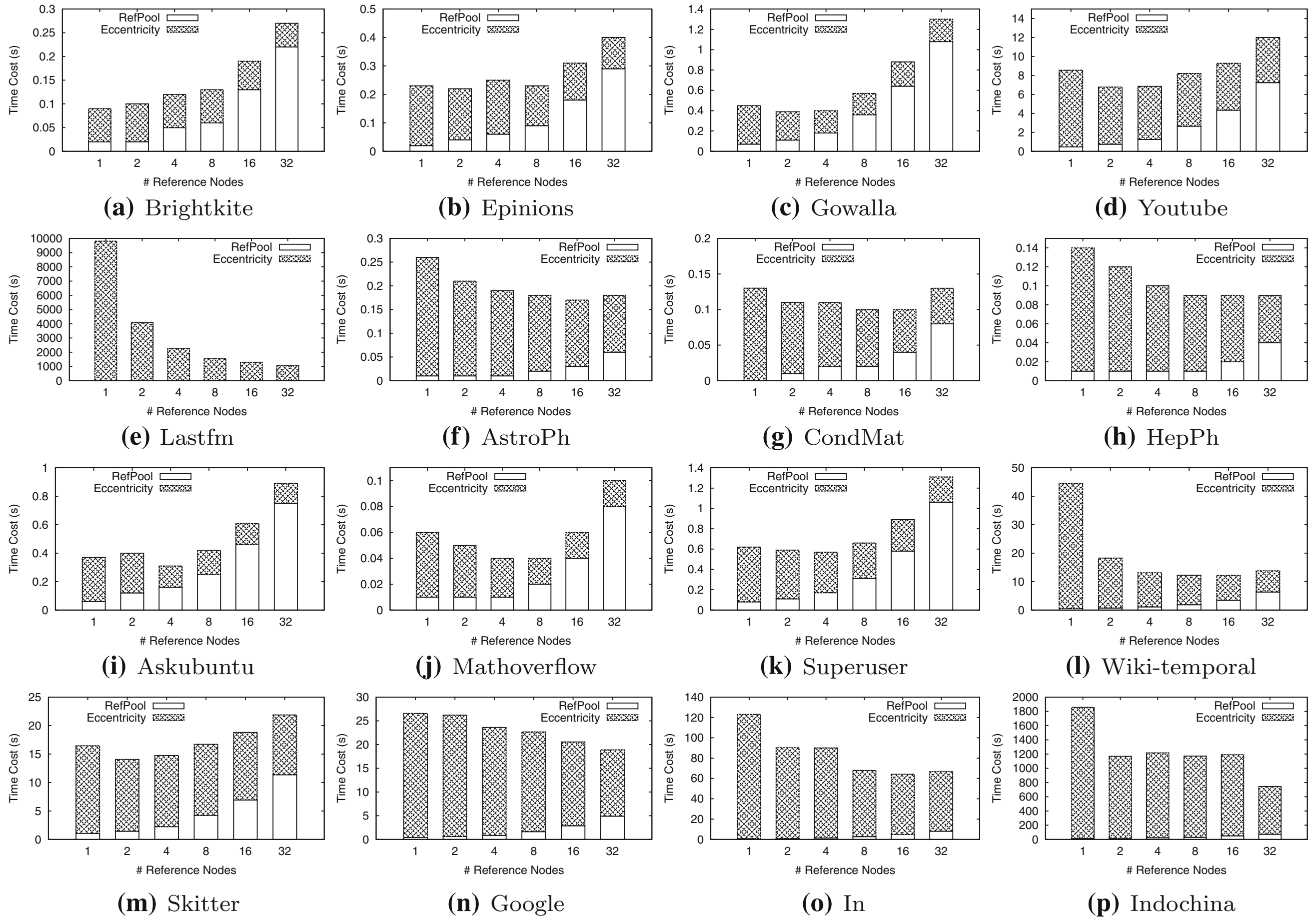

Fig. 29 Testing ECC (varying # reference nodes)

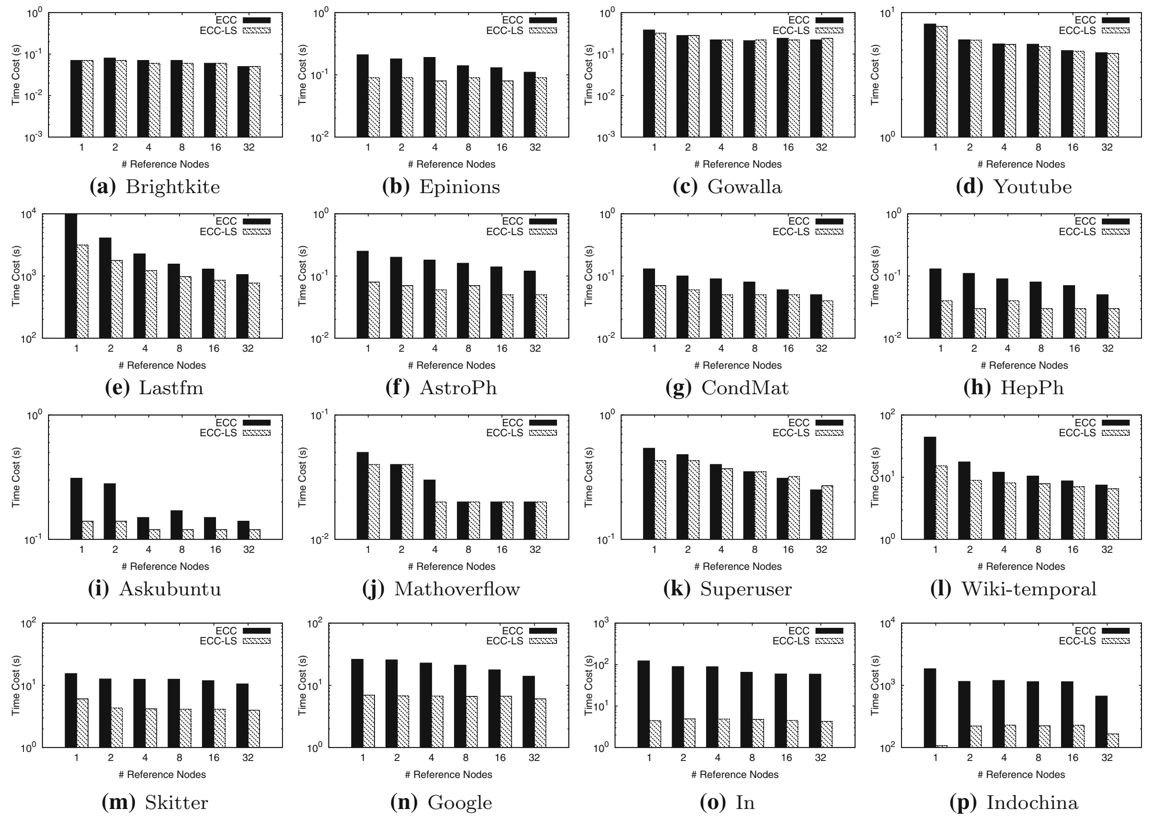

Fig. 30 Testing ECC-LS (processing time for Eccentricity)

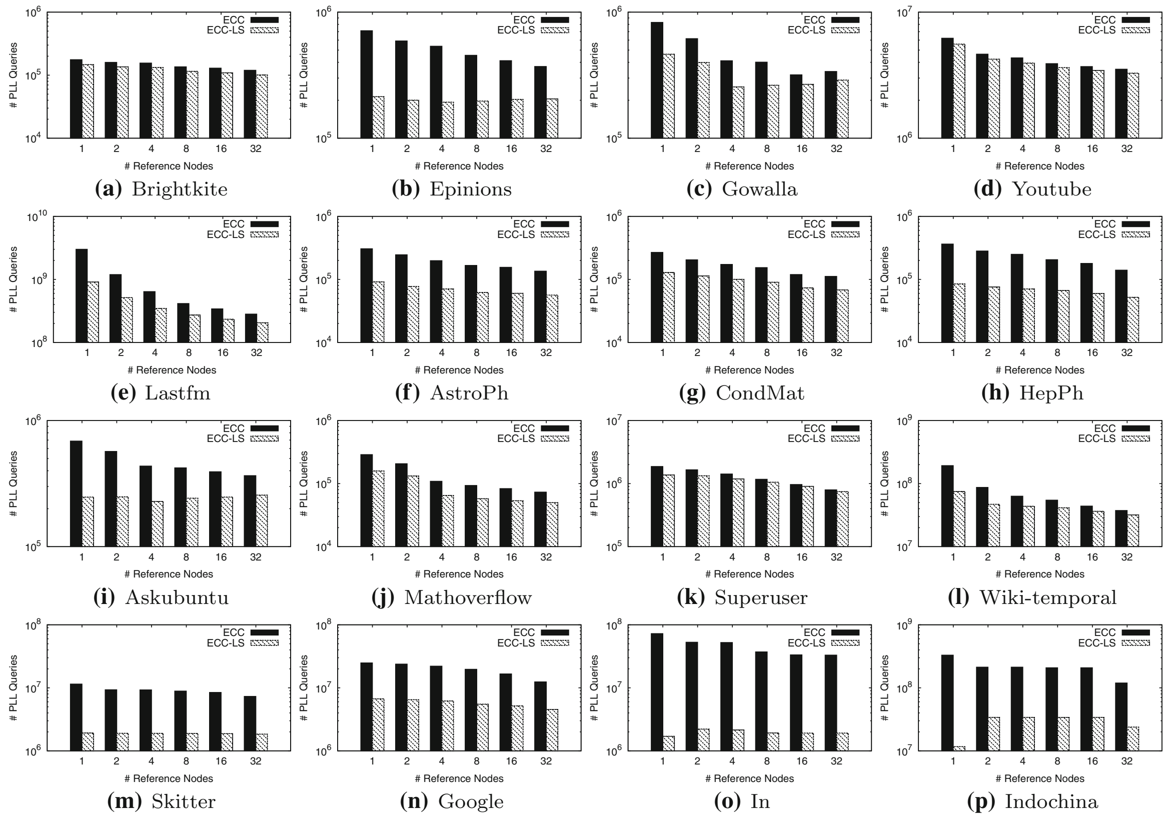

Fig. 31 Testing ECC-LS (# PLL queries)

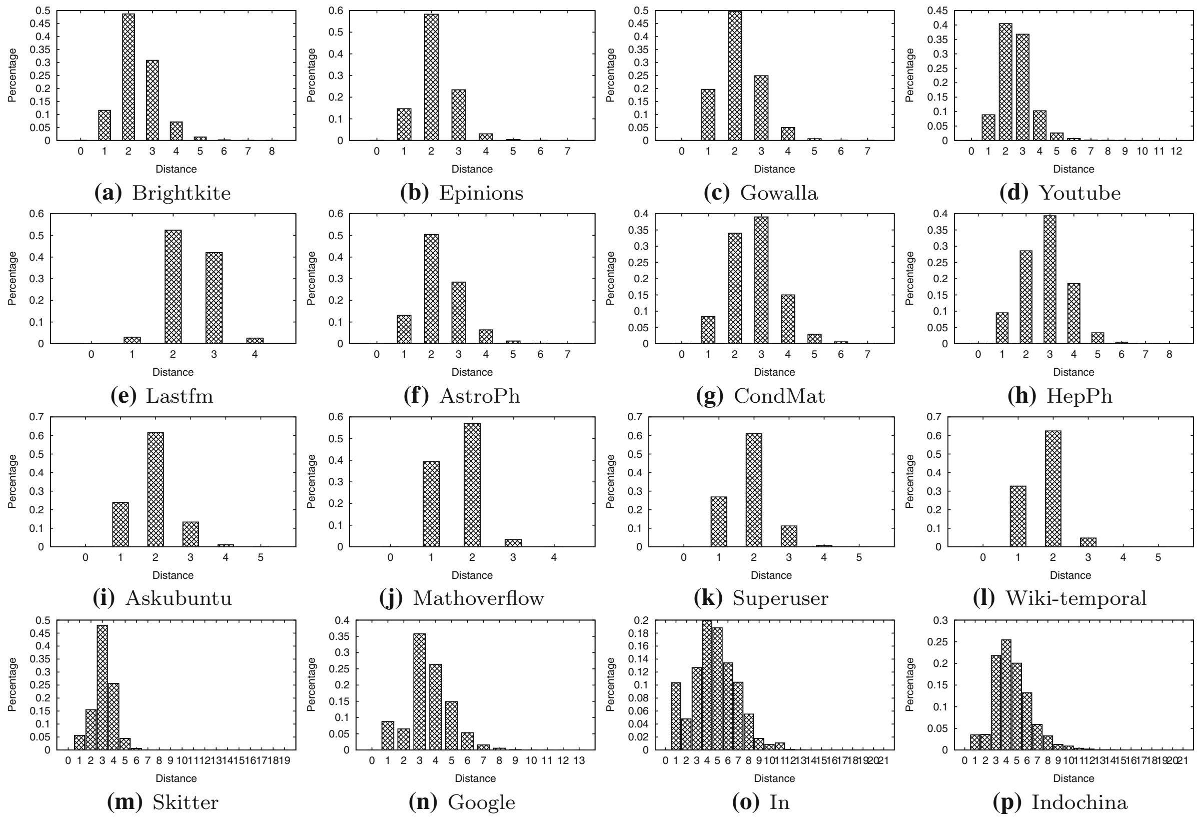

Fig. 32 Testing distribution of distance to reference nodes

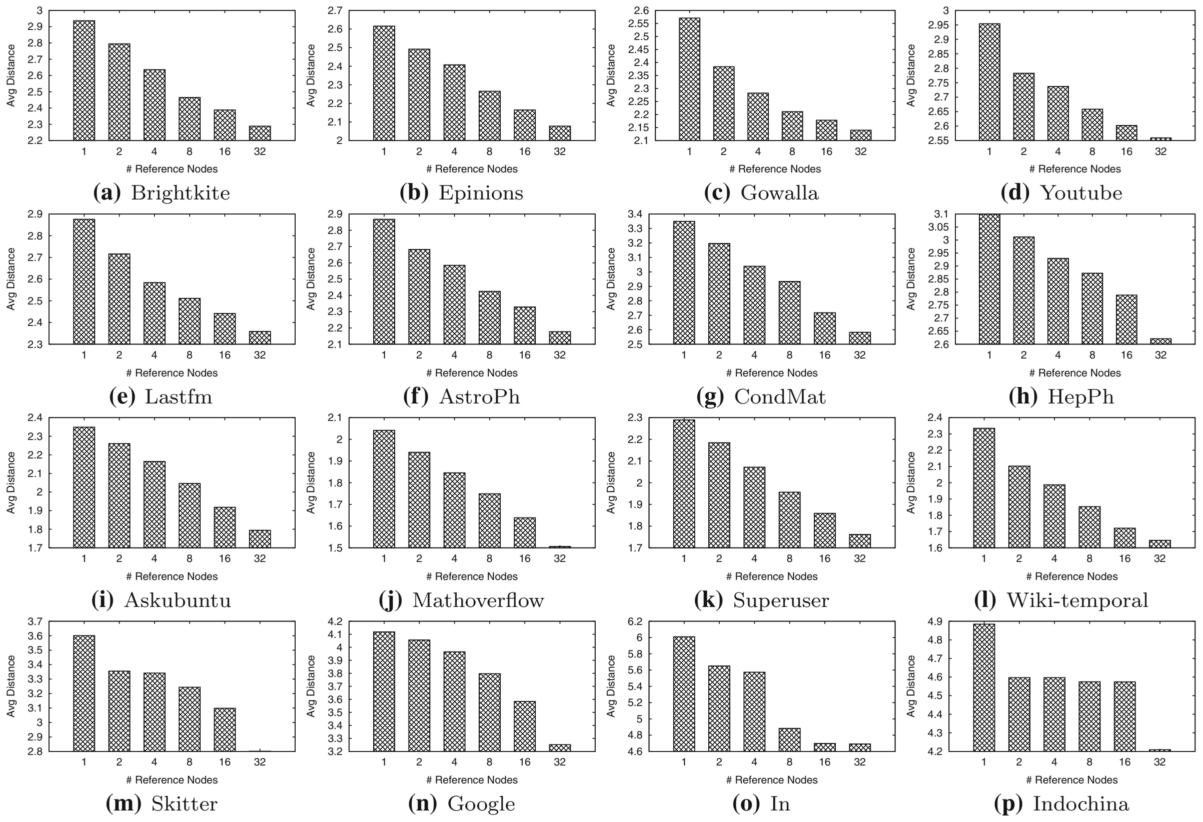

Fig. 33 Testing average distance to reference nodes

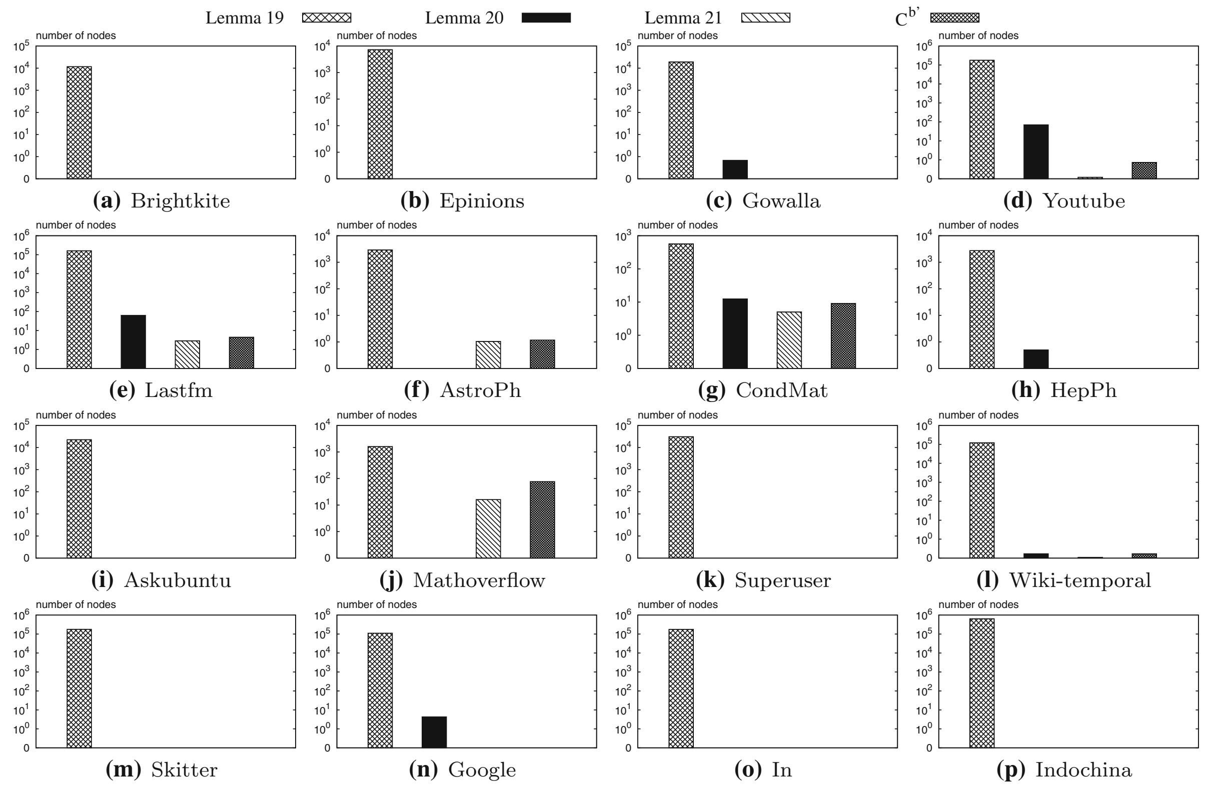

Fig. 34 Testing the average number of nodes in each step of ECC-DY for edge insertion

u to its reference node z. Figure 33 illustrates the average distance of a node to its nearest reference node.

– Exp-8: Rule-effectiveness for edge insertion. Figure 34 shows the number of nodes pruned by each of Lemmas 19–21 and the number of nodes in $C^{b'}$.

References

1. Aingworth, D., Chekuri, C., Indyk, P., Motwani, R.: Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM J. Comput. **28**(4), 1167–1181 (1999)
2. Akiba, T., Iwata, Y., Kawata, Y.: An exact algorithm for diameters of large real directed graphs. In: International Symposium on Experimental Algorithms, pp. 56–67. Springer, Berlin (2015)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 349–360. ACM, New York (2013)
4. Almeida, P., Baquero, C., Cunha, A.: Fast distributed computation of distances in networks. In: 2012 IEEE 51st IEEE Conference on Decision and Control (CDC), pp. 5215–5220. IEEE, New York (2012)
5. Bisenius, P., Bergamin, E., Angriman, E., Meyerhenke, H.: Computing top-k closeness centrality in fully-dynamic graphs. In: 2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 21–35. SIAM (2018)
6. Borassi, M., Crescenzi, P., Habib, M., Kosters, W.A., Marino, A., Takes, F.W.: Fast diameter and radius bfs-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games. Theoret. Comput. Sci. **586**, 59–80 (2015)
7. Chan, T.M.: All-pairs shortest paths for unweighted undirected graphs in o (mn) time. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, pp. 514–523. Society for Industrial and Applied Mathematics (2006)
8. Chechik, S., Larkin, D.H., Roditty, L., Schoenebeck, G., Tarjan, R.E., Williams, V.V.: Better approximation algorithms for the graph diameter. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1041–1052. Society for Industrial and Applied Mathematics, Philadelphia (2014)
9. Demetrescu, C., Italiano, G.F.: Experimental analysis of dynamic all pairs shortest path algorithms. ACM Trans. Algorithm. (TALG) **2**(4), 578–601 (2006)
10. Fujiwara, Y., Onizuka, M., Kitsuregawa, M.: Real-time diameter monitoring for time-evolving graphs. In: International Conference on Database Systems for Advanced Applications, pp. 311–325. Springer, Berlin (2011)
11. Gaston, M.E., Kraetzl, M., Wallis, W.D.: Using graph diameter for change detection in dynamic networks. Australas. J. Comb. **35**, 299–311 (2006)
12. Guare, J.: Six Degrees of Separation: A Play. Vintage, New York (1990)
13. Henderson, K.: Opex: Optimized eccentricity computation in graphs. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (2011)
14. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM (JACM) **24**(1), 1–13 (1977)
15. Kas, M., Carley, K.M., Carley, L.R.: Incremental closeness centrality for dynamically changing social networks. In: Proceedings of the 2013 IEEE/ACM International Conference on Advances in

Social Networks Analysis and Mining, pp. 1250–1258. ACM, New York (2013)
16. Leskovec, J., Krevl, A.: Snap datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
17. Li, Z., Sun, D., Xu, F., Li, B.: Social network based anomaly detection of organizational behavior using temporal pattern mining. In: Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, pp. 1112–1119. ACM, New York (2017)
18. Lü, L., Chen, D., Ren, X.-L., Zhang, Q.-M., Zhang, Y.-C., Zhou, T.: Vital nodes identification in complex networks. Phys. Rep. **650**, 1–63 (2016)
19. Nathan, E., Zakrzewska, A., Riedy, J., Bader, D.: Local community detection in dynamic graphs using personalized centrality. Algorithms **10**(3), 102 (2017)
20. Newman, M.E.J.: A measure of betweenness centrality based on random walks. Social Netw. **27**(1), 39–54 (2005)
21. Okamoto, K., Chen, W., Li, X.-Y.: Ranking of closeness centrality for large-scale social networks. In: International Workshop on Frontiers in Algorithmics, pp. 186–195. Springer, Berlin (2008)
22. Riondato, M., Upfal, E.: Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. ACM Trans. Knowl. Discov. Data (TKDD) **12**(5), 61 (2018)
23. Roditty, L., Vassilevska Williams, V.: Fast approximation algorithms for the diameter and radius of sparse graphs. In: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, pp. 515–524. ACM, New York (2013)
24. Sagharichian, M., Langouri, M.A., Naderi, H.: A fast method to exactly calculate the diameter of incremental disconnected graphs. World Wide Web **20**(2), 399–416 (2017)
25. Sariyüce, A.E., Kaya, K., Saule, E., Çatalyiirek, Ü.V.: Incremental algorithms for closeness centrality. In: 2013 IEEE International Conference on Big Data, pp. 487–492. IEEE, New York (2013)
26. Shun, J.: 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, pp. 1095–1104. ACM, New York (2015)
27. Takes, F., Kosters, W.: Computing the eccentricity distribution of large graphs. Algorithms **6**(1), 100–118 (2013)
28. Takes, F.W., Kosters, W.A.: Determining the diameter of small world networks. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, pp. 1191–1196. ACM, New York (2011)
29. Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T.-A., Pham, K., Kemper, A., Neumann, T., Huy, T.V.: The more the merrier: efficient multi-source graph traversal. Proc. VLDB Endow. **8**(4), 449–460 (2014)
30. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world'networks. Nature **393**(6684), 440 (1998)
31. West, D.B., et al.: Introduction to Graph Theory, vol. 2. Prentice Hall, Upper Saddle River, NJ (1996)
32. Williams, R.: Faster all-pairs shortest paths via circuit complexity. In: Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, pp. 664–673. ACM, New York (2014)
33. Yen, C.-C., Yeh, M.-Y., Chen, M.-S.: An efficient approach to updating closeness centrality and average path length in dynamic networks. In: 2013 IEEE 13th International Conference on Data Mining, pp. 867–876. IEEE, New York (2013)