Efficient Pseudoarboricity Computation in Large Static and Dynamic Graphs

ABSTRACT

The arboricity $a(G)$ of a graph G is defined as the minimum number of edge-disjoint forests that the edge set of G can be partitioned into. It is a fundamental metric and has been widely used in many graph analysis applications. However, computing $a(G)$ is typically a challenging task. To address this, a concept called pseudoarboricity was proposed as an alternative measure which is relatively easier to compute. Pseudoarboricity has been shown to be closely connected to many important measures in graphs, including the arboricity and the densest subgraph density $\rho(G)$. Computing the exact pseudoarboricity can be achieved by employing a parametric max-flow algorithm, but it becomes computationally expensive for large graphs. Existing 2-approximation algorithms, while more efficient, often lack satisfactory approximation accuracy. To overcome these limitations, we propose two new linear-time approximation algorithms with theoretical guarantees to estimate the pseudoarboricity. We show that our approximation algorithms can significantly reduce the search space for the exact parametric max-flow algorithm, greatly improving its efficiency for accurate pseudoarboricity computation. In addition to static graph analysis, we also study the pseudoarboricity maintenance problem in dynamic graphs. We propose two novel and efficient algorithms for maintaining the pseudoarboricity when the graph is updated by edge insertions or deletions. Furthermore, we develop two incremental pseudoarboricity maintenance algorithms specifically designed for scenarios where only edge insertions occur in the dynamic graph. We conduct extensive experiments on 195 real-world graphs, and the results demonstrate the high efficiency and scalability of the proposed algorithms in computing pseudoarboricity for both static and dynamic graphs.

ACM Reference Format:
. 2023. Efficient Pseudoarboricity Computation in Large Static and Dynamic Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Arboricity is defined as the minimum number of edge-disjoint forests into which the edges of a graph can be partitioned [38, 39]. As a classic measure of graph's sparsity, arboricity has been widely used to analyze the time or space complexity of numerous graph analysis algorithms, such as triangle counting [28**?**], k-clique listing [15, 18, 34], truss decomposition [32, 43], structural graph clustering [12], influential community mining [35, 36], structural diversity search [13, 29], top-k ego-betweenness search [44], and so on.

Despite its importance in network analysis, computing the arboricity of a graph is typically a time-consuming task [25]. To address this, Picard and Queyranne introduced an alternative concept of pseudoarboricity [39], which is relatively easier to compute. Specifically, pseudoarboricity is defined as the minimum number of edge-disjoint pseudoforests into which the edges of a graph can be partitioned [38, 39]. Here a pseudoforest refers to a graph in which each connected component contains at most one cycle. Picard and Queyranne proved that the pseudoarboricity is equal to either the arboricity or arboricity minus one, thus making it a useful metric for approximating arboricity.

More importantly, pseudoarboricity is a concept that is well understood and offers several advantages over arboricity in various application scenarios. We outline three key applications as follows. (1) We show that pseudoarboricity can also be applied to analyze the time or space complexity of all the above-mentioned graph analysis algorithms. Furthermore, since pseudoarboricity is no larger than arboricity, it can obtain a tighter bound in analyzing these algorithms. (2) In terms of measuring graph sparsity, pseudoarboricity is equal to the round-up value of the density of the densest subgraph, which is more intuitive compared to arboricity. Additionally, we demonstrate that pseudoarboricity can be used to detect densely-connected communities in networks, further highlighting its usefulness. (3) In the problem of minimizing the maximum indegree through orienting an undirected graph, existing studies typically rely on forest partition to achieve small indegree orientations [4, 7]. However, pseudoforest partition can offer even smaller indegree orientations compared to forest partition. Motivated by these observations, we focus on investigating the problem of efficiently computing the pseudoarboricity of a graph.

Previous approaches to compute the pseudoarboricity mainly rely on the connection between pseudoarboricity and optimal graph orientation. The objective of the optimal graph orientation is to assign directions to the edges of a graph such that the maximum indegree of the resulting oriented graph is minimized. The pseudoarboricity was shown to be equal to the smallest maximum indegree achievable in any orientation of the graph [9]. Based on this observation, Bezakova proposed an algorithm that uses the re-orientation network flow and binary search techniques to compute the optimal orientation, where the resulting smallest maximum indegree corresponds to the pseudoarboricity [9]. The key idea of Bezakova's algorithm is that it leverages a re-orientation network flow to test whether a graph can be oriented to a directed graph such that its maximum indegree is smaller than a given parameter k. If so, the pseudoarboricity of the graph must be less than k, and no less than k otherwise. The algorithm then employs a binary search procedure to find the optimal parameter k, which represents the pseudoarboricity. The time complexity of this algorithm is $O(|E|^{3/2} \log p)$, where p denotes the pseudoarboricity.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Building upon Bezakova's algorithm, Blumenstock [10] developed several effective pruning strategies to reduce the input graph size and further proposed an advanced binary search technique to speed up the computation, which achieves the state-of-the-art (SOTA) performance for exact pseudoarboricity computation. However, even with these optimizations, the SOTA algorithm still incurs significant computational costs when handling large graphs.

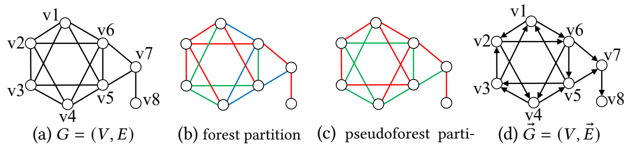
In addition to exact computation algorithms, there are also several more efficient 2-approximation algorithms available for approximating the pseudoarboricity [5, 9, 10, 21]. These algorithms are mainly based on the *peeling* idea, where the algorithm iteratively removes the vertex with the minimum degree in the graph and orients all adjacent edges towards it until all vertices are deleted. After removing all vertices, the maximum indegree of the resulting orientation provides a 2-approximation of the pseudoarboricity. The time complexity of these approximation algorithms is linear with respect to (w.r.t.) the graph size. However, as evidenced by our experiments, the quality of the approximation provided by these algorithms is often unsatisfactory, i.e., the approximated pseudoarboricity tends to be close to twice of the true pseudoarboricity.

Contributions. To overcome these limitations, we first propose an improved linear-time 2-approximation algorithm to estimate the pseudoarboricity. Our algorithm offers a theoretical improvement over previous approaches by achieving a slightly better approximation factor. It follows a similar peeling algorithm as before but introduces an additional step where certain edges pointing towards vertices with high indegrees are reversed. This results in an orientation with a reduced maximum indegree. To further improve the efficiency, we also propose a novel approximate algorithm with a different theoretical guarantee. This approach first constructs the orientation by considering the indegree of each vertex. Subsequently, it iteratively verifies if each edge points towards the vertex with smaller indegree. This approximate algorithm yields such a high-quality approximation that it frequently provides the exact value of the pseudoarboricity. As a result, it is highly effective in practical scenarios. Furthermore, both of our approximation algorithms can serve as a valuable preprocessing step to reduce the graph size when employing exact pseudoarboricity computation algorithms. The high-quality approximation offered by our algorithms can even obviate the necessity for a binary search in the exact algorithms.

Other than computing the pseudoarboricity in static graphs, we also study the problem of maintaining the pseudoarboricity in dynamic graphs with edge insertions or deletions. Specifically, we first present a pseudoarboricity update theorem, with which we develop two basic maintenance algorithms, called BasicINS and BasicDEL, to handle edge insertion and deletion respectively. The key idea of both BasicINS and BasicDEL is that they maintain the pseudoarboricity by maintaining the optimal orientation. However, a major drawback of these methods is that they require invoking the max-flow algorithm whenever handling edge insertions or deletions, which is often costly for large graphs. To improve the efficiency, we propose two novel and more efficient algorithms, namely INS and DEL, to handle edge insertion and deletion respectively. Instead of maintaining the optimal orientation, both INS and DEL maintains a different concept called *unreversible orientation* (detailed definition can be found in Section 4), which is a special kind of optimal orientation originally proposed in [42]. The striking feature of our novel algorithms is that: in most cases, they can avoid invoking a max-flow algorithm and solely perform a Breadth-First Search (BFS) algorithm to maintain the pseudoarboricity, thus they are much more efficient than the basic algorithms. Additionally, we also develop two incremental pseudoarboricity maintenance algorithms, called INC and INS++, specifically designed for scenarios where only edge insertions occur in the dynamic graph. We show that both the two incremental algorithms can be much more efficient than INS. Compared to INC, INS++ maintains an additional structure D_{top} which can significantly prune redundant BFS searches, making it more efficient. Moreover, we show that the D_{top} structure is very close to the densest subgraph. The difference of the densities between the subgraph induced by D_{top} and the densest subgraph is no larger than 1, indicating that D_{top} can also be used to detect the dense subgraph in a graph.

We conduct extensive experiments using 195 real-life graphs to evaluate the proposed algorithms. The results are summarized as follows. (1) Most real-world graphs have small pseudoarboricity except for a few substantially large biological graphs, collaboration graphs and hyperlink graphs, which confirm the "small arboricity" assumption in most real-world graphs made in many previous studies [15, 28, 36, 37, 45]. (2) Both of our two approximation algorithms outperform previous approaches in terms of both approximation quality and running time. In addition, when compared to other competitors, our best approximation algorithm consistently achieves a significantly higher level of approximation quality while requiring an order of magnitude less time. The discrepancy between the approximate and exact pseudoarboricity on all tested datasets is no larger than 4. Leveraging the superior performance of our best approximation algorithm, the proposed algorithm for exact pseudoarboricity computation can achieve a speedup of up to 21 times compared to the SOTA algorithm. This result demonstrates the high efficiency and effectiveness of the proposed algorithm for pseudoarboricity computation on static graph. (3) To dynamically maintain the pseudoarboricity, the proposed INS and DEL algorithms are very efficient, and they can be 2-3 orders of magnitude faster than the basic algorithms BasicINS and BasicDEL. On a large graph **SKALL** with 50.6 million nodes and 1.8 billion edges, INS (DEL) only takes 21 (3.5) seconds to handle 10,000 edge insertions (deletions). (4) When only considering edge insertions, our incremental maintenance algorithms INC and INS++ are extremely efficient, both of them can handle 10,000 insertions within less than 0.01 seconds on **SKALL**. (5) The subgraph induced by D_{top}, obtained as a by-product of pseudoarboricity computation, can be used to effectively detect communities in real-life graphs with a density difference from the densest subgraph density of no more than 0.01 over all datasets.

Reproducibility. For reproducibility purpose, we release the source code of this paper at an anonymous link: https://anonymous.4open.science/r/Pseudoarboricity-Computation-86BF.



(a) $G = (V, E)$ (b) forest partition (c) pseudoforest partition (d) $\vec{G} = (V, \vec{E})$

Figure 1: Running example.

2 PRELIMINARIES

2.1 Notations and problem definition

Let $G = (V, E)$ be an unweighted and undirected graph, where V denotes the set of nodes and E denotes the set of edges. An undirected graph $G = (V, E)$ can be transformed into a directed graph $\vec{G} = (V, \vec{E})$ by assigning a direction to each edge in G. This assignment of directions yields an orientation of G, which we refer to as $\vec{G}$. Let (u, v) and $\langle u, v\rangle$ denote an undirected edge and a directed edge, respectively. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, denoted as $G' \subseteq G$. Let $d_u(G)$ be the degree of node u in G (i.e., the number of neighbors of u in G), and $d_u(\vec{G})$ be the in-degree of u in $\vec{G}$ (i.e., the number of in-going neighbors of u in $\vec{G}$). If the context is clear, we will use d_u to denote the degree (in-degree) of a u in G (in $\vec{G}$). In the directed graph $\vec{G} = (V, \vec{E})$, A *path* is a sequence of vertices $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$, for $i = 1, \ldots, k-1$, $\langle v_i, v_{i+1}\rangle \in E$, and the length of this path is $k - 1$. If we *reverse* a directed edge, the direction of the edge is changed. If we *reverse* a path, all edges in the path are reversed. For convenience, we denote a path from v_1 to v_k as $v_1 \rightsquigarrow v_k$.

Definition 1. **(Arboricity)** [38] *The arboricity of a graph G, denoted by $a(G)$, is defined as the smallest number of edge-disjoint forests that the edge set E of G can be partitioned into.*

Arboricity is a classic metric in graph theory which was frequently used to measure the sparsity of a graph [15, 21]. However, arboricity is often difficult to compute on large graphs [25]. In their work [39], Picard and Queyranne introduced an alternative metric called pseudoarboricity, which provides a very tight approximation of the arboricity while being significantly easier to compute.

Specifically, pseudoarboricity is defined using the concepts of *pseudotrees* and *pseudoforests* [39]. A *pseudotree* is an undirected graph that is connected and contains precisely one cycle [39]. Clearly, the number of vertices in a *pseudotree* is equal to the number of edges it has. On the other hand, a *pseudoforest* is a graph where each connected component is either a tree or a *pseudotree*. Based on these concepts, the pseudoarboricity can be defined as follows.

Definition 2. **(Pseudoarboricity)** [39] *The pseudoarboricity of a graph G, denoted by $p(G)$, is the minimum number of edge-disjoint pseudoforests that the edge set E can be partitioned into.*

As shown in [39], the pseudoarboricity of a graph G is either equal to the arboricity or only 1 less than the arboricity, *i.e.,* $p(G) \in \{a(G), a(G) - 1\}$, indicating that pseudoarboricity is a very tight approximation of arboricity.

Example 1. *Consider a graph G shown in Fig. (1a). A feasible forest partition and pseudoforest partition of G are shown in Fig. (1b) and Fig. (1c) respectively, where each color represents a distinct (pseudo)forest. It is easy to check that the number of partitions is minimized. Thus, we have $a(G) = 3$ and $p(G) = 2$.*

Based on Definition 2, the goal of this paper is to efficiently compute the pseudoarboricity in both large static and dynamic graphs. More formally, we formulate our problem as follows.

Problem definition. For a static graph G, our goal is to efficiently compute $p(G)$. However, in the case of a dynamic graph G with potential edge updates, the problem is to maintain $p(G)$ after an edge insertion or deletion.

2.2 Motivation: why pseudoarboricity?

Pseudoarboricity, like arboricity, provides an intuitive measure of graph sparsity. However, we argue that pseudoarboricity is a more valuable metric in graph analysis applications for the following reasons: (1) Pseudoarboricity is simpler to compute compared to arboricity; (2) Pseudoarboricity is closely related to many other graph analysis problems, establishing valuable connections for further exploration and analysis; (3) Pseudoarboricity proves highly useful in analyzing the complexity of numerous graph mining applications.

More specifically, as shown in [25], the computation of arboricity can be solved using a matroid partition algorithm, which is typically more complex than the parametric max-flow algorithm used to compute the pseudoarboricity [10]. Moreover, the pseudoarboricity is also closely related to several significant metrics of the graph.

Pseudoarboricity and $p(G)$-orientation. Let $\vec{G} = (V, \vec{E})$ be an orientation of $G = (V, E)$. A k-orientation of G is an orientation $\vec{G}$ in which the maximum indegree in $\vec{G}$ is k. The optimal orientation is the orientation in which the maximum degree is minimized, i.e., the minimum k-orientation. For example, the orientation in Fig. (1d) is an optimal orientation of Fig. (1a). In [9], Bezakova established an interesting connection between the pseudoarboricity and the optimal orientation of a graph. Specifically, Bezakova shown that a k-orientation of G is optimal if and only if $k = p(G)$. This result suggests that computing the pseudoarboricity of a graph G is equivalent to finding the optimal orientation of G. In this work, we will develop several efficient algorithms for pseudoarboricity computation on static and dynamic graphs based on this crucial connection.

Pseudoarboricity and graph density. Given a graph G, the densest subgraph is the subgraph of G with the highest density, where the *density* $\rho(g)$ of a graph g is defined as $\rho(g) \triangleq |E_g|/|V_g|$. In [39], Picard and Queyranne proved that $p(G)$ is the rounded-up value of the densest subgraph density, *i.e.,* $p(G) = \max_{G' \subseteq G} \lceil |E'|/|V'| \rceil = \lceil \rho(G') \rceil$. Based on this, an alternative approach to compute $p(G)$ is by utilizing existing densest subgraph computation algorithms [19, 26]. Nevertheless, our experiments show that the performance of the state-of-the-art densest subgraph algorithm [19] is often inferior to that of the algorithm designed for computing the optimal orientation.

Pseudoarboricity and degeneracy. Pseudoarboricity is also closely connected to the degeneracy of a graph, which is another classic metric used to quantify the sparsity of a graph [22]. Before giving the definition of degeneracy, we first give the definition of k-core. Specifically, a k-core is the maximum subgraph of G where every node has degree no less than k. The degeneracy of G, denoted $\delta(G)$, is the maximum k such that the k-core of G exists. As shown in [11], for a graph G, the degeneracy $\delta(G)$ is a 2-approximation of the pseudoarboricity $p(G)$, *i.e.*, $p(G) \leq \delta(G) \leq 2p(G)$. Based on this, the k-core based pruning technique can be used to speed up the pseudoarboricity computation [10] (the vertices outside the $\delta(G)/2$-core can be safely pruned without changing the pseudoarboricity).

Except for the above-mentioned connections with other graph metrics, the pseudoarboricity can also be used to bound the time or space complexity for many graph analysis algorithms. Moreover, it is slightly tighter than arboricity to bound the complexity of these graph analysis algorithms.

A tighter bound for complexity analysis. Chiba and Nishizeki [15] proved an important inequality: $\sum_{(u,v)\in E} \min\{d_u, d_v\} \leq 2|E|a(G)$. With this inequality, the arboricity is widely used to bound the time or space complexity of many graph analysis algorithms, such as triangle listing [15, 28], k-clique counting [18, 34], truss decomposition [32, 43], structural graph clustering [12], structural diversity search [30, 31, 45], and influential communities search [35, 36]. Below, we show that replacing the arboricity $a(G)$ with the pseudoarboricity $p(G)$ in this inequality can lead to a tighter bound for analyzing the time or space complexity.

Theorem 1. *Given an undirected graph $G = (V, E)$, the inequality $\sum_{(u,v)\in E} \min\{d_u, d_v\} \leq 2|E|p(G) \leq 2|E|a(G)$ holds.*

Proof. Based on the relationship between the pseudoarboricity and the optimal orientation, G has an optimal orientation $\vec{G}$ where the indegree of every vertex is less than $p(G)$. For each edge $(u,v) \in G$, let $h(u,v)$ be the vertex that (u,v) is oriented toward in $\vec{G}$. Then we have $\sum_{(u,v)\in E} \min\{d_u, d_v\} \leq \sum_{(u,v)\in E} d_{h(u,v)} = \sum_{u\in V} d_u(G) d_u(\vec{G}) \leq \sum_{u\in V} d_u(G)p(G) \leq 2|E|p(G)$. □

Based on Theorem 1, we can always use pseudoarboricity, instead of arboricity, to bound the time (or space) complexity of the above-mentioned graph analysis algorithms.

3 COMPUTING $p(G)$ IN STATIC GRAPHS

3.1 Existing algorithms

Existing algorithms for pseudoarboricity computation can be divided into two categories: approximate algorithms and exact algorithms. Based on the connection between the pseudoarboricity and the optimal orientation of a graph [9], existing approximate algorithms aim to find a near-optimal orientation and use the maximum indegree as an approximation of the pseudoarboricity [10]. On the other hand, the exact algorithms employ the re-orientation network flow technique to obtain optimal orientations [5, 10]. Below, we briefly review both approximate and exact algorithms for computing pseudoarboricity.

Approximate algorithms. The widely-used approximate algorithm for pseudoarboricity computation is outlined in Lines 3-10 of Algorithm 1 [5, 9, 21]. This algorithm iteratively removes the vertex with the minimum degree u in the graph and orients all edges linked to u toward u, until all vertices are removed. After deleting all vertices, all edges are oriented. At this point, a 2-approximate orientation can be obtained, and the maximum indegree is a 2-approximation of the pseudoarboricity. Based on this 2-approximation algorithm, Blumenstock [10] introduced a k-core based pruning technique for speeding up exact pseudoarboricity computation. Such a core-pruning method is shown in Lines 11-12 of Algorithm 1.

Algorithm 1: DEGREE (G)

```
1  Input: A simple graph $G = (V, E)$.
2  Output: A 2-approximate orientation $\vec{G}$.
3  Let $\vec{E} \leftarrow \emptyset, \vec{G} \leftarrow (V, \vec{E})$;
4  Let $core[1 \dots |V|]$ be an array and $nowcore \leftarrow 0$;
5  while $V$ is not empty do
6      $u \leftarrow \arg\min_{u\in V} d_u(G)$;
7      $nowcore \leftarrow \max\{nowcore, d_u\}$; $core[u] \leftarrow nowcore$;
8      for $e = (v, u) \in E$ do
9          $\vec{E} \leftarrow \vec{E} \cup \langle v, u\rangle$; $E \leftarrow E - (v, u)$; $d_v(G) \leftarrow d_v(G) - 1$;
10     $V \leftarrow V - u$;
11 Let $p_0 \leftarrow \lceil \max d_u(\vec{G})/2 \rceil$ and $V' \leftarrow$ all vertices in the $p_0$-core of $G$;
12 $\vec{G} \leftarrow$ the subgraph of $\vec{G}$ induced by $V'$;
13 return $\vec{G}$;
```

Algorithm 2: ReTest ($\vec{G}, k$)

```
1  Input: An orientation $\vec{G} = (V, \vec{E})$ and a test value $k$.
2  Output: Test whether $p(G) \leq k$, and the updated $\vec{G}$.
3  $V' \leftarrow V \cup \{s, t\}$, where $s$ is a source node and $t$ is a sink node;
4  foreach $\langle v, u\rangle \in \vec{E}$ do
5      Add arc $\langle u, v\rangle$ to $A$ and let $c(u, v) \leftarrow 1$;
6  foreach $u, d_u(\vec{G}) > k$ do
7      Add arc $\langle s, u\rangle$ to $A$ and let $c(s, u) \leftarrow d_u(\vec{G}) - k$;
8  foreach $u, d_u(\vec{G}) < k$ do
9      Add arc $\langle u, t\rangle$ to $A$ and let $c(u, t) \leftarrow k - d_u(\vec{G})$;
10 Compute the maximum flow value $f_{\max}$ of $(V', A, c)$;
11 foreach $\langle v, u\rangle \in \vec{E}$ do
12     if $\langle u, v\rangle \in A$ is saturated then reverse the edge $\langle v, u\rangle \in \vec{G}$;
13 if $f_{\max} = \sum_{\langle s,u\rangle\in A} c(s, u)$ then return (True,$\vec{G}$);
14 else return (False,$\vec{G}$);
```

Lemma 1. [10] *Given a graph G and an integer $p_0 \leq p(G)$. Denote by G' the p_0-core of G, then we have $p(G) = p(G')$.*

Example 2. *Assume that the graph G shown in Fig. (1a) is the input graph of Algorithm 1. Upon execution of the 'while' loop of Algorithm 1, a possible processing order of vertices is $(v_8, v_7, \dots, v_1)$ and an approximate orientation can be obtained with a maximum indegree of 4. Thus, we have $p_0 = \lceil 4/2 \rceil = 2$. Based on Lemma 1, Algorithm 1 computes the 2-core of G, which is induced by $\{v_1, \dots, v_7\}$. Finally, Algorithm 1 outputs the approximate orientation of the 2-core of G.*

We can easily derive that both the time and space complexity of Algorithm 1 are $O(|E| + |V|)$. A notable limitation of Algorithm 1 is that: it may allocate indegrees unevenly, neglecting the ability of some vertices to carry more indegrees. For instance, in Example 2, the last-deleted vertex v_1 has an indegree of 0, while the vertex v_6 has an indegree of 4. By contrast, the optimal orientation depicted in Fig. (1d) achieves a more desirable even allocation of indegrees.

Exact algorithms. The state-of-the-art exact algorithm employs a *parameterized* re-orientation network flow technique to calculate the pseudoarboricity by choosing a test value k and checking whether $p(G) \leq k$ [5, 9, 33]. Given an orientation $\vec{G} = (V, \vec{E})$ and an integer $k \geq 0$, the re-orientation network is defined as $(V \cup \{s, t\}, A, c)$, where (i) $\langle u, v\rangle \in A, c(u, v) = 1, if\ \langle v, u\rangle \in E$; (ii) $\langle s, u\rangle \in A, c(s, u) = d_u(\vec{G}) - k, if\ d_u(\vec{G}) > k$; and (iii)

Algorithm 3: iDEGREE (G)

```
1  Input: A simple graph $G = (V, E)$.
2  Output: An improved 2-approximate orientation $\vec{G}$.
3  Let $\vec{E} \leftarrow \emptyset, \vec{G} \leftarrow (V, \vec{E}), \rho^* \leftarrow 0$;
4  Let $core[1 \dots |V|]$ be an array and $nowcore \leftarrow 0$;
5  Let $order$ be an empty stack;
6  while $V$ is not empty do
7      $u \leftarrow \arg\min_{u \in V} d_u(G)$;
8      $nowcore \leftarrow \max\{nowcore, d_u\}, core[u] \leftarrow nowcore$;
9      $order.push(u)$;
10     for $e = (v, u) \in E$ do
11         $\vec{E} \leftarrow \vec{E} \cup \langle v, u \rangle; E \leftarrow E - (v, u); d_v(G) \leftarrow d_v(G) - 1$;
12     $V \leftarrow V - u$;
13     $\rho^* \leftarrow \max\{\rho^*, |E|/|V|\}$;
14 Let $p_0 \leftarrow \lceil \rho^* \rceil$ and $V' \leftarrow$ all vertices in the $p_0$-core of $G$;
15 $\vec{G} \leftarrow$ the subgraph of $\vec{G}$ induced by $V'$;
16 while true do
17     $u \leftarrow order.pop()$;
18     if $u \notin \vec{G}$ then break;
19     While $\exists \langle u, v \rangle \in \vec{E}, d_v \geq d_u + 2$ do reverse $\langle u, v \rangle$;
20 return $\vec{G}$;
```

$\langle u, t \rangle \in A, c(u, t) = k - d_u(\vec{G}), if\ d_u(\vec{G}) < k$. The integer k is a test value that checks whether a k-orientation exists, *i.e.*, it verifies whether $p(G) \leq k$. The ReTest algorithm is outlined in Algorithm 2. Intuitively, $c(s, u) = d_u - k$ implies that d_u needs to be reduced by $c(s, u)$, while $c(u, t) = k - d_u$ implies that u has the ability to carry extra $c(u, t)$ indegrees. After performing the max-flow algorithm, if all arcs originating from s are saturated, a k-orientation can be obtained by reversing edge $\langle u, v \rangle$ in $\vec{G}$ with saturated arc $\langle v, u \rangle \in A$. Conversely, if not all arcs originating from s are saturated, a k-orientation does not exist, *i.e.*, $p(G) > k$. For any test value k, the time complexity of ReTest is $O(|E|^{3/2})$ using the classic Dinic's max-flow algorithm [10, 23]. The practical performance of this algorithm is often sensitive to the input orientation. Generally, the algorithm runs faster when the maximum indegree of the input orientation is smaller.

Based on such a re-orientation network flow technique, the state-of-the-art algorithm for pseudoarboricity computation, proposed by Blumenstock [10], includes three steps: (i) Calculate a 2-approximate orientation by DEGREE and record its maximum indegree $d_{\max}$; (ii) Use the reduction technique in DEGREE to reduce the graph size; (iii) Perform a binary search on the range $[d_{\max}/2, d_{\max}]$ by iteratively invoking ReTest to obtain the exact pseudoarboricity. It is easy to derive that the total time complexity of this algorithm can be bounded by $O(|E|^{3/2} \log p(G))$.

3.2 An improved 2-approximation algorithm

To address the shortcoming of DEGREE, we propose an improved 2-approximate algorithm, called iDEGREE, by making two improvements on DEGREE. First, we additionally compute the subgraph density ρ^* as an 1/2-approximation of pseudoarboricity to achieve a better reduction performance. Second, we re-orient some edges toward the later-deleted vertices after the reduction to balance the indegree distribution, and further obtain an orientation with a smaller maximum indegree. The detailed descriptions of iDEGREE is outlined in Algorithm 3. The following example illustrates how iDEGREE works.

Example 3. *Consider the graph G in Fig. (1a). The working-flow of* iDEGREE *before core-pruning is almost the same as* DEGREE*, except that* iDEGREE *computes the subgraph density $\rho^* = 2$. Since 'order' is a stack structure,* iDEGREE *begins to re-orient edges in the reverse order of vertex deletion,* i.e.*, $v_1, \dots, v_7$. For v_1 and v_2, the algorithm re-orients $\langle v_1, v_3 \rangle$, $\langle v_1, v_5 \rangle$ and $\langle v_2, v_6 \rangle$. For the vertices v_3, v_4, v_5, v_6 and v_7, the orientation of the edges associated with them remains unchanged. Finally, the vertex with the maximum indegree is v_6 and thus we obtain a 3-orientation, which is better than the orientation calculated by* DEGREE.

Below, we show that iDEGREE can always output an orientation whose maximum indegree is no larger than that of the orientation output by DEGREE, thus iDEGREE can often achieve a better approximation quality.

Lemma 3.1. *For a graph $G = (V, E)$, let $\vec{G_1}$ and $\vec{G_2}$ be the approximate orientations output by* DEGREE *and* iDEGREE*, respectively. Then, we have $\max_{u \in V} d_u(\vec{G_1}) \geq \max_{u \in V} d_u(\vec{G_2})$.*

Proof. After the execution of the first 'while' loop (Lines 5-10 of Algorithm 1, Lines 6-13 of Algorithm 3), DEGREE and iDEGREE obtain the same $\vec{G}$, thus we focus on the second 'while' loop in iDEGREE (Lines 16-19 of Algorithm 3). In this loop, only the indegree of vertex u increases when $d_v \geq d_u + 2$. Note that u is not the vertex with the maximum indegree, therefore, the second 'while' loop in iDEGREE does not increase the maximum indegree. □

Except for offering better approximation quality, iDEGREE can also provide more effective pruning performance compared to DEGREE. After the execution of the first 'while' loop of Algorithm 1, we can obtain an orientation $\vec{G}$. Let ρ^* be the graph density obtained by Algorithm 1. Then, we have the following result.

Lemma 3.2. $\rho^* \geq \max_{u \in V} d_u(\vec{G})/2$.

Proof. Let $v \in \vec{G}$ be an arbitrary vertex that has the maximum indegree after the first 'while' loop. Then, before v is deleted from V, v is the vertex with smallest degree, thus we have $|E| = \frac{1}{2}\sum_{u \in V} d_u(G) \geq \frac{1}{2} d_v(G)|V| = \frac{1}{2} d_v(\vec{G})|V|$. Since $\rho^* \geq |E|/|V|$, we can derive that $\rho^* \geq d_v(\vec{G})/2 = \max_{u \in V} d_u(\vec{G})/2$. □

Lemma 3.2 indicates that the core-pruning method used in Algorithm 3 can achieve a better pruning performance compared to that applied in Algorithm 1. In addition, we can also easily derive that both the time and space complexity of Algorithm 3 are $O(|E| + |V|)$.

3.3 A novel approximate algorithm

We propose a novel approximate algorithm with a different guarantee, called INDEGREE, to approximate the pseudoarboricity. The INDEGREE algorithm is outlined in Algorithm 4. The key idea of INDEGREE is to iteratively reverse the edges so that each edge is directed toward the endpoint with smaller indegree. In particular, INDEGREE first constructs an orientation $\vec{G}$ where each edge initially points towards the endpoint with the smaller indegree (Lines 5-9). However, some early-oriented edges may later point at the endpoint with higher indegree as some later-oriented edges choose to point at them. Therefore, the algorithm

Algorithm 4: INDEGREE (G)

```
Input: A simple graph $G = (V, E)$.
Output: An approximate orientation $\vec{G} = (V, \vec{E})$.
$\vec{G} \leftarrow \emptyset$;
Initialize an array $d_i = 0, i = 1, \ldots, |V|$;
foreach $(u, v) \in G$ do
    if $d_u < d_v$ then
        $\vec{G} \leftarrow \vec{G} \cup \langle v, u \rangle$; $d_u \leftarrow d_u + 1$
    else
        $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$; $d_v \leftarrow d_v + 1$
repeat
    foreach $\langle from, to \rangle \in \vec{G}$ do
        if $d_{to} \geq d_{from} + 2$ then
            reverse $\langle from, to \rangle$; $d_{from} \leftarrow d_{from} + 1$; $d_{to} \leftarrow d_{to} - 1$;
until Current iteration does not reduce the maximum indegree of $\vec{G}$;
return $\vec{G}$;
```

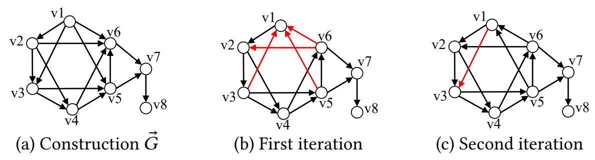
examines whether each edge points towards the vertex with lower indegree, and reverses it if not (Lines 10-14). Note that if $d_{to} = d_{from} + 1$, the algorithm does not reverse the edge $\langle from, to \rangle$ because reversing it only exchanges the indegree of $from$ and to, which is of no benefit. Intuitively, INDEGREE is more effective in approximating the pseudoarboricity for two reasons: (i) unlike DEGREE, which orients edges based on the degree of vertices, INDEGREE focuses on reducing the indegree of vertices, which is closer to the objective of the optimal orientation problem (i.e., minimizing the maximum indegree); (ii) INDEGREE can perform multiple iterations of edge reversal to iteratively decrease the maximum indegree of an orientation, thereby achieving a more and more precise approximation of the pseudoarboricity.

Stop condition of iterations. As shown in Theorem 2, the iterative edge reversal process in Algorithm 4 (Lines 10-14) will terminate when $d_{to} \leq d_{from} + 1$ holds for every edge $< from, to >$ in $\vec{G}$.

Theorem 2. *For any input graph G, INDEGREE converges to a stable orientation $\vec{G}$, where $d_{to} \leq d_{from} + 1$ holds for each edge $\langle from, to \rangle$.*

Proof. Let $U(\vec{G}) \triangleq \sum_{u \in \vec{G}} d_u^2$ be the *uneven index*. In INDEGREE, if the edge $\langle from, to \rangle$ is reversed, then $d_{to} \geq d_{from} + 2$ and $U(\vec{G})$ decreases by $((d_{from}+1)^2 + (d_{to}-1)^2) - (d_{from}^2 + d_{to}^2) = 2(d_{to} - d_{from}) - 2 \geq 2$. As $U(\vec{G}) \geq 0$, it follows that $U(\vec{G})$ cannot be indefinitely small, which means that the edges cannot be reversed for an infinite number. Thus, after a sufficient number of iterations, the orientation will converge to a stable orientation, where $\nexists \langle from, to \rangle$, such that $d_{to} \geq d_{from} + 2$. □

Clearly, the number of iterations for edge reversal is critical to the efficiency and effectiveness of INDEGREE. In general, fewer iterations may result in a large $d_{\max}$ of the orientated graph, thus obtaining a poor approximation of pseudoarboricity; while the excessive number of iterations increases the time cost. To strike a balance between the number of iterations and the approximation quality, we propose to use "the maximum indegree does not decrease in current iteration" as the stop condition (Line 14), which is shown to be efficient and effective in our experiments. The following example illustrates how INDEGREE works.



(a) Construction $\vec{G}$ (b) First iteration (c) Second iteration

Figure 2: Example of running INDEGREE

Example 4. *Consider a graph G in Fig. (1a). Suppose that the INDEGREE algorithm constructs the orientation $\vec{G}$ based on the order of the adjacency list,* i.e., $(v_1, v_2), (v_1, v_3), \ldots, (v_7, v_8)$. *Fig. (2a) shows the constructed $\vec{G}$, in which the indegree distribution is uneven. Subsequently, INDEGREE performs the first iteration of edge reversal and Fig. (2b) illustrates the resulting orientation, whose maximum indegree is smaller than that of the initial orientation. Then, INDEGREE proceeds with the second iteration to further balance the indegree distribution of all vertices. The newly-obtained orientation is shown in Fig. (2c), which is already an optimal orientation. Further iterations will not result in any edge reversals. After the third iteration, the maximum indegree does not decrease in this iteration and thus the INDEGREE algorithm terminates. Finally, INDEGREE obtains the maximum indegree which is equal to 2.*

Let τ be the number of edge-reversal iterations performed by the INDEGREE algorithm. Then, the time complexity of INDEGREE is $O(\tau|E| + |V|)$ and the space complexity is $O(|E| + |V|)$. As shown in our experiments, τ is often a small constant, thus INDEGREE is very efficient in practice. Below, we analyze the approximation quality of INDEGREE.

Theorem 3. *Let $\vec{G}$ be a stable orientation of $G = (V, E)$ obtained by INDEGREE and $d_{\max} = \max_{u \in \vec{G}} d_u$. Then, we have $\prod_{k=p}^{d_{\max}} \frac{k}{p} \leq |V|$.*

Proof. Let $v \in \vec{G}$ be an arbitrary vertex with $d_{\max}$ indegree. V_i is the set of vertices at a distance no greater than i from v, and E_{i+1} is the set of edges with both endpoints in V_{i+1}. Because $\vec{G}$ is a stable orientation, each vertex in V_i has an indegree of at least $d_{\max} - i$, thus $|E_{i+1}| \geq (d_{\max} - i)|V_i|$ holds. Combined with $|E_{i+1}|/|V_{i+1}| \leq \rho(G) \leq p$, we have $|V_{i+1}| \geq \frac{d_{\max}-i}{p}|V_i|$. By iteratively applying this inequality for $i = 0, 1, \ldots, d_{\max} - p$, we can derive that $|V| \geq |V_{d_{\max}-(p-1)}| \geq \prod_{k=p}^{d_{\max}} \frac{k}{p}$. □

With Theorem 3, the maximum indegree $d_{\max}$ provides a reliable approximation of $p(G)$, and it is typically not significantly larger than $p(G)$. For instance, let us assume that $p(G) = 80$. In such a case, it is possible to observe that $d_{\max} < 2 \times p(G) = 160$. If this condition is not met, the number of vertices $|V|$ would have to exceed 3.7×10^{13}, which is not commonly observed in real-world graphs. In practical scenarios, the approximation quality of INDEGREE is very good. Our experiments demonstrate that $d_{\max}$ obtained by INDEGREE does not exceed $p(G) + 4$ over all datasets.

3.4 Exact pseudoarboricity computation

For both iDEGREE and INDEGREE, we can combine them with ReTest to devise an exact algorithm for computing the pseudoarboricity. Since our approximation algorithms are often very accurate, we can first invoke ReTest $(\vec{G}, d_{\max}(\vec{G}))$ to check whether $p(G) = d_{\max}(\vec{G})$, where $\vec{G}$ is the approximate orientation obtained

Algorithm 5: BasicINS ($\vec{G}$, (u, v), p)

```
1 Input: An orientation $\vec{G} = (V, \vec{E})$, the edge $(u, v)$ to be inserted, and the
    pseudoarboricity $p$ before insertion.
2 Output: The updated orientation $\vec{G}$ and pseudoarboricity.
3 Suppose $d_v \leq d_u$, otherwise swap the input edge $(u, v)$;
4 $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
5 if ReTest $(\vec{G}, p)$ =false then $p$++;
6 return $(\vec{G}, p)$;
```

by iDEGREE or INDEGREE. If so, we can skip the binary search. Otherwise, we perform the binary search on $[d_{\max}(\vec{G})/2, d_{\max}(\vec{G})]$ and iteratively apply ReTest on the output orientation of iDEGREE or INDEGREE to calculate the exact pseudoarboricity.

Note that we can also use the advanced binary search technique developed in [10] to achieve the same worst-case time complexity of the state-of-the-art exact algorithm. However, compared to the state-of-the-art exact algorithm [10], our exact algorithm is equipped with a more powerful approximation technique, which can efficiently produce a high-quality approximation than the algorithm developed in [10], thus it is often substantially faster than the state-of-the-art exact algorithm as shown in our experiments.

4 MAINTAINING $p(G)$ IN DYNAMIC GRAPHS

Real-world graphs are frequently updated, requiring efficient algorithms to maintain the pseudoarboricity when the graph undergoes edge insertions or deletions. In this section, our goal is to devise such algorithms. We start by establishing a pseudoarboricity update theorem, which serves as the foundation for designing pseudoarboricity maintenance algorithms. Then, we will develop several advanced and more efficient maintenance algorithms.

THEOREM 4. ***(Pseudoarboricity update theorem).*** *After an edge insertion (resp. deletion) of G, the pseudoarboricity of G increases(resp. decreases) by at most one.*

PROOF. We first consider the edge insertion case. Since $p(G) = \max_{G' \subseteq G} \lceil |E'|/|V'| \rceil$, the maximum density does not decrease after inserting an edge. As a consequence, the $p(G)$ also does not decrease after an edge insertion. By the definition of pseudoforest, the newly-inserted edge can only form a new pseudoforest consisting of itself, and thus the pseudoarboricity could increase by at most one. For the edge deletion case, we let e be the deleted edge and $\bar{G}$ denotes the graph after removing e from $G = (V, E)$, i.e., $\bar{G} = (V, E \setminus \{e\})$. Then, we have $p(\bar{G}) = \max_{G' \subseteq \bar{G}} \lceil |E'|/|V'| \rceil \geq \max_{G' \subseteq G} \lceil (|E'| - 1)/|V'| \rceil \geq p(G) - 1$. Therefore, the pseudoarboricity could decrease by at most one after deleting an edge. □

4.1 A basic maintenance algorithm

The basic algorithm for edge insertion. With Theorem 4, the basic algorithm for edge insertion, denoted by BasicINS, is shown in Algorithm 5. Specifically, the BasicINS algorithm first orients the inserted edge (u, v) toward its smaller-indegree endpoint. For example, if the in-degree d_v is smaller than d_u, then we obtain an directed edge $\langle u, v \rangle$. Then, BasicINS tests whether $p(G) \leq p$ using ReTest, where p is the pseudoarboricity before inserting (u, v). If not, the pseudoarboricity p increases by 1 according to Theorem 4, and p remains unchanged otherwise. It is easy to see that the wort-case time complexity of Algorithm 5 is $O(|E|^{3/2})$, as it only invokes ReTest once.

The basic algorithm for edge deletion. Similarly, based on Theorem 4, we present a basic algorithm for edge deletion, called BasicDEL. The BasicDEL algorithm first removes the deleted edge (u, v) from $\vec{G}$, and then invokes ReTest with parameter $k = p - 1$ to check whether $p(G) \leq (p - 1)$ holds. If ReTest outputs TRUE, BasicDEL sets p as $p - 1$, and p keeps unchanged otherwise. For brevity, we omit the pseudo-code of BasicDEL. Similar to BasicINS, the time complexity of BasicDEL is $O(|E|^{3/2})$, since it invokes ReTest once. Below, we analyze the correctness of BasicINS and BasicDEL.

THEOREM 5. *Given a graph $G = (V, E)$, the updated edge (u, v), and the pseudoarboricity p, the* BasicINS *and* BasicDEL *algorithms correctly maintain p.*

PROOF. Recall that invoking once ReTest $(\vec{G}, k)$ can test whether $p(G) \leq k$. By Theorem 4, after an edge insertion, the pseudoarboricity is either p or $p + 1$. Thus, invoking ReTest $(\vec{G}, p)$ can check whether the pseudoarboricity is p or $p + 1$. Similarly, after an edge deletion, the pseudoarboricity is either $p - 1$ or p, which can be determined by performing ReTest $(\vec{G}, p - 1)$. □

4.2 A novel and faster maintenance algorithm

The basic algorithm involves constructing the re-orientation network and invoking the max-flow algorithm for every edge insertion or deletion, which is clearly time-consuming. To improve the efficiency, we propose a novel pseudoarboricity maintenance algorithm based on a concept of *unreversible orientation*, which is a special kind of optimal orientation originally proposed in [42]. The striking feature of our novel algorithm is that: in most cases, it can avoid using a re-orientation network and solely perform a Breadth-First Search (BFS) algorithm to maintain the pseudoarboricity, thus it is often much faster than the basic algorithm which is based on the max-flow computation. Below, we first introduce the concept of *unreversible orientation*.

For an orientation $\vec{G}$, let $d_{\max} = \max_{u \in \vec{G}} d_u$ be the maximum indegree. Define a *reversible path* as a path $s \rightsquigarrow t$ where $d_t = d_{\max}$ and $d_s \leq d_{\max} - 2$. It is important to note that if we reverse such a *reversible path*, i.e., reverse the directions of all the edges of the *reversible path*, the maximum indegree of the vertices in the path will decrease by 1. The reason is that after reversing a *reversible path* $s \rightsquigarrow t$, the indegree of s increases by 1, the indegree of t decreases by 1, and the indegrees of the other vertices keep unchanged. Since $d_t = d_{\max}$ and $d_s \leq d_{\max} - 2$, the maximum indegree of the vertices in the *reversible path* $s \rightsquigarrow t$ is $d_{\max} - 1$ after reversing $s \rightsquigarrow t$.

The *unreversible orientation* is an orientation without any *reversible path* [42]. As shown in [42], the maximum indegree of an unreversible orientation $\vec{G}$ is equal to $p(G)$. Based on this result, an immediate question is that can we dynamically maintain the pseudoarboricity of a graph G via maintaining an unreversible orientation of G. In the following, we develop an efficient algorithm to achieve this goal.

Novel algorithm for edge insertion. For edge insertion, we have the following important lemma.

Algorithm 6: INS $(\vec{G}, (u,v), p)$

```
1 Input: An unreversible orientation $\vec{G} = (V, \vec{E})$, the edge $(u,v)$ to be inserted, and
    the pseudoarboricity $p$ before insertion.
2 Output: The updated unreversible orientation $\vec{G}$ and pseudoarboricity.
3 Suppose $d_v \le d_u$, otherwise swap the input edge $(u,v)$;
4 $\vec{G} \leftarrow \vec{G} \cup \langle u,v\rangle$;
5 if $d_v = \max_{i \in \vec{G}} d_i$ (i.e., $d_v = p+1$ or $p$) and there is a reversible path ending at $v$ then
6     reverse the path;
7 if $\max_{i\in\vec{G}} d_i > p$ then $p$++;
8 return $(\vec{G}, p)$;
```

Lemma 2. *Given an unreversible orientation $\vec{G}$ and an edge $\langle u,v\rangle$ for insertion, $\vec{G} \cup \langle u,v\rangle$ is an unreversible orientation if there is no reversible path ending at v in $\vec{G} \cup \langle u,v\rangle$.*

Proof. Let d_v and d_v^* denote the indegree of v before and after edge insertion, respectively. For the case of $d_v = p$, the edge insertion of $\langle u,v\rangle$ makes d_v^* equal $p+1$. Thus, v becomes the only vertex with the maximum indegree; and therefore, any reversible path must end at v by definition. As a result, if there is no reversible path ending at v in $\vec{G}\cup\langle u,v\rangle$, then we can conclude that no reversible path exists in $\vec{G} \cup \langle u,v\rangle$, indicating that $\vec{G} \cup \langle u,v\rangle$ is an unreversible orientation. For the case of $d_v \le p-1$, we prove it by contradiction. Suppose that there is no reversible path ending at v but a reversible path $s \rightsquigarrow t$ after inserting $\langle u,v\rangle$. The path $s \rightsquigarrow t$ must contain $\langle u,v\rangle$, otherwise a reversible path $s \rightsquigarrow t$ already exists before insertion, a contradiction to $\vec{G}$ being an unreversible orientation before the edge insertion. Since $s \rightsquigarrow t$ is a reversible path, we have $d_s \le p-2$ and $d_t = p$. Before the edge insertion, $v \rightsquigarrow t$ must not be a reversible path as $\vec{G}$ is an unreversible orientation, thus $d_v > p-2$ holds. Due to $d_v \le p-1$, we can derive $d_v = p-1$, and further $d_v^* = p$. Now, there is a reversible path $s \rightsquigarrow v$ ending at v, which leads to a contradiction. Putting all it together, the lemma is established. □

Equipped with Lemma 2, we present a novel algorithm, called INS, to handle the edge insertion by maintaining an unreversible orientation. The pseudo-code of INS is depicted in Algorithm 6. When inserting $\langle u,v\rangle$, the INS algorithm tries to find a reversible path starting from v and reverse it to obtain an unreversible orientation if $d_v = \max_{i\in\vec{G}} d_i$ (Lines 5-6), otherwise there is no reversible path after edge insertion. Note that to find the reversible path ending at v, we can perform a BFS from v and traverse along the opposite direction of each edge, until finding a vertex with the indegree of $\le p-2$ or the BFS-search queue is empty. INS can determine the pseudoarboricity by verifying $\max_{i\in\vec{G}} d_i > p$ directly (Line 7). Finally, INS outputs the maximum indegree of $\vec{G}$ as the updated pseudoarboricity and also returns the updated unreversible orientation. Note that INS at most needs to reverse 1 reversible path to obtain the unreversible orientation. Below, we prove the correctness of INS in Theorem 6.

Theorem 6. *The* INS *algorithm correctly maintains the unreversible orientation and pseudoarboricity.*

Proof. If INS does not find any reversible path ending at v in $\vec{G}$, then $\vec{G}$ is an unreversible orientation by Lemma 2 and the correctness can be easily guaranteed. On the other hand, when the INS algorithm finds a reversible path ending at v, it is sufficient to prove that there is no reversible path after reversing this path. We prove it by contradiction, *i.e.*, assuming that after reversing a reversible path $w \rightsquigarrow v$, there is still a reversible path $s \rightsquigarrow t$. Clearly, the two reversible paths $s \rightsquigarrow t$ and $w \rightsquigarrow v$ must overlap each other, otherwise $s \rightsquigarrow t$ exists in the input unreversible orientation. We denote the first and the last overlapping vertices in $s \rightsquigarrow t$ as x and y. Then, before reversing $w \rightsquigarrow v$, there are two cases of d_v: (i) $d_v = p+1$ and (ii) $d_v = p$. For the first case, d_v increases from p to $p+1$ when inserting $\langle u,v\rangle$ into $\vec{G}$. After reversing, the maximum indegree decreases from $p+1$ to p. Since $s \rightsquigarrow t$ is a reversible path, we have $d_s \le p-2$ and $d_t = p$. If u is on the reversible path $w \rightsquigarrow v$, we have $d_u \ge p$ which causes a contradiction: the input orientation has a reversible path $s \rightsquigarrow x \rightsquigarrow u$. If u is not on the path $w \rightsquigarrow v$, the input orientation also has a reversible path $s \rightsquigarrow x \rightsquigarrow v$, which is a contradiction. For the second case, *i.e.*, $d_v = p$, since $w \rightsquigarrow v$ is a reversible path, we have $d_w \le p-2$. After reversing, since $s \rightsquigarrow t$ is a reversible path, we have $d_s \le p-2$ and $d_t = p$. Then, there is a contradiction: the input orientation has a reversible path $w \rightsquigarrow y \rightsquigarrow t$ and thus not be unreversible. In summary, INS can update the unreversible orientation and pseudoarboricity correctly. □

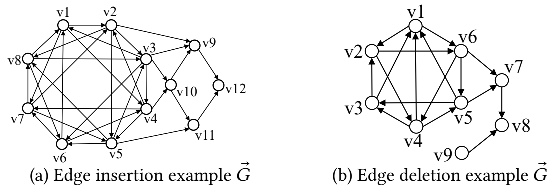

v1
v2
v3
v9
v8
v10
v12
v7
v4
v11
v6
v5
(a) Edge insertion example $\vec{G}$
v1
v2
v6
v7
v3
v5
v8
v4
v9
(b) Edge deletion example $\vec{G}$

Figure 3: Example of dynamic grpahs

Algorithm 7: DEL $(\vec{G}, (u,v), p)$

```
1 Input: An unreversible orientation $\vec{G} = (V, \vec{E})$, the edge $(u,v)$ to be deleted, and the
    pseudoarboricity $p$ before deletion.
2 Output: The updated unreversible orientation $\vec{G}$ and pseudoarboricity.
3 $\vec{G} \leftarrow \vec{G} - \langle u,v\rangle$   // suppose $(u,v)$ is oriented as $\langle u,v\rangle$;
4 if $d_v \ge \max_{i\in\vec{G}} d_i - 2$ and there is a reversible path starting from $v$ then
5     reverse the path;
6 if $\max_{i\in\vec{G}} d_i < p$ then
7     $p$− −; ReTest $(\vec{G}, p-1)$;
8 return $(\vec{G}, p)$;
```

The following example illustrates how Algorithm 6 works.

Example 5. *Consider the orientation $\vec{G}$ shown in Fig. (3a). It is easy to derive that $\vec{G}$ is an unreversible orientation and $p(G) = 3$. If we insert (v_1, v_9), the $d_{\max}$ in $\vec{G} \cup \langle v_1, v_9\rangle$ is equal to 4. The* INS *algorithm finds a reversible path $v_{10} \longrightarrow v_9$ and reverses it. Then $d_{\max}$ decreases to 3 and $\vec{G} \cup \langle v_1, v_9\rangle$ become unreversible again, thus the pseudoarboricity does not change. When inserting (v_1, v_5), we can find that $\vec{G} \cup \langle v_1, v_5\rangle$ is still be unreversible. However, $d_{\max}$ increases to 4, thus the pseudoarboricity also increases to 4.*

Below, we analyze the time complexity of the INS algorithm. INS performs BFS from v to find a reversible path which costs $O(|E|)$ time. Besides, the algorithm can check $\max_{i\in\vec{G}} d_i > p$ within $O(1)$ time because the maximum indegree is either p or d_v. Therefore, the time complexity of INS is $O(|E|)$.

Novel algorithm for edge deletion. Algorithm 7 outlines the pseudo-code of our new algorithm, called DEL, for handling edge deletion. Similar to INS, DEL maintains the pseudoarboricity and an unreversible orientation by finding a reversible path starting from v and reversing it when $d_v \geq \max_{i\in\vec{G}} d_i - 2$ (Lines 4-5); on the other hand, $d_v < \max_{i\in\vec{G}} d_i - 2$ indicates that there is no reversible path starting from v since the input orientation is unreversible. DEL updates the pseudoarboricity according to whether $\max_{i\in\vec{G}} d_i < p$ holds (Line 6). Unlike INS, for the edge deletion case, a single BFS invoking is not sufficient to maintain the unreversible orientation when the pseudoarboricity decreases. The reason is as follows. Let p_0 be the pseudoarboricity before deleting an edge. Since the input orientation is unreversible, it can guarantee that there is no path $s \rightsquigarrow t$ where $d_s = p_0$ and $d_t \leq p_0 - 2$, but it cannot guarantee there is no path $s \rightsquigarrow t$ where $d_s = p_0 - 1$ and $d_t \leq p_0 - 3$. When the pseudoarboricity decreases by one, the condition for the orientation to be reversible becomes there being no path $s \rightsquigarrow t$ where $d_s = p_0-1$ and $d_t \leq p_0 - 3$. However, there may exist many such paths in the input orientation. The BFS algorithm can only reverse one path at a time and is not suitable for situations where multiple paths need to be reversed. Therefore, based on the results shown in Lemma 3, DEL invokes the ReTest algorithm to reverse these paths, making the orientation unreversible again (Line 7). Example 6 illustrates how Algorithm 7 works.

Lemma 3. *Given a graph G, let $\vec{G}$ be an optimal orientation of G, then after invoking* ReTest $(\vec{G}, p(G) - 1)$, *$\vec{G}$ will become an unreversible orientation.*

Proof. Assume that when ReTest terminates, there is still a reversible path $x \rightsquigarrow y$ with $d_x \leq p - 2$ and $d_y = p$. That means there is a path $s \to y \to \cdots \to x \to t$ from the source to the sink in the re-orientation network, which can increase the total flow by one unit. This contradicts the fact that the maximum flow algorithm terminates before the total flow increases to its maximum value, thus the lemma is established. □

Example 6. *Consider the orientation $\vec{G}$ shown in Fig. (3b) as the input of* DEL. *We can easily check that $\vec{G}$ is an unreversible graph and $p(G) = 3$. If we delete $\langle v_6, v_7\rangle$,* DEL *cannot find any reversible path starting from v_7, and $\vec{G}$ is still unreversible. The maximum indegree and pseudoarboricity do not change. While deleting $\langle v_3, v_2\rangle$,* DEL *finds a reversible path $v_2 \to v_4 \to v_1$ and reverses it. The maximum indegree decreases from 3 to 2, so the pseudoarboricity also decreases to 2. Note that at this moment $\vec{G}$ is not unreversible since there is a reversible path $v_9 \to v_8$. Then* ReTest $(\vec{G}, 1)$ *is invoked to reverse the edge $\langle v_9, v_8\rangle$, and thus $\vec{G}$ becomes unreversible again.*

Theorem 7. DEL *can correctly maintain the unreversible orientation and pseudoarboricity.*

Proof. In Line 6 of DEL, if $\max_{i\in\vec{G}} d_i < p$ holds, then the pseudoarboricity decreases to $p - 1$ and $\vec{G}$ is an optimal orientation with $p - 1$ maximum indegree. By Lemma 3, invoking ReTest $(\vec{G}, p - 1)$ can make $\vec{G}$ become unreversible again. Hence, DEL can correctly output unreversible and pseudoarboricity; on the other hand, we consider the case of $\max_{i\in\vec{G}} d_i = p$. Note that in Line 4, if $d_v < \max_{i\in\vec{G}} d_i - 2$, we can derive that before the deletion $d_v \leq p-2$ and there is no reversible path starting from v. Next, we prove that the output orientation is unreversible by discussing whether DEL can find a reversible path starting from v. If DEL cannot find such a reversible path, we can easily prove that there is also no reversible path from other vertices, thus $\vec{G}$ obtained by DEL is unreversible; for the case of DEL can find a reversible path starting from v, we prove it by contradiction. Assume that after reversing the reversible path $v \rightsquigarrow w$, there is still a reversible path $s \rightsquigarrow t$. The two reversible paths must overlap each other, otherwise $s \rightsquigarrow t$ exists in the input orientation. We denote the first and the last overlapping vertices in $s \rightsquigarrow t$ as x and y. Before reversing, we have $d_v \leq p - 2$, $d_w = p$, $d_s \leq p - 2$, and $d_t = p$. Thus, there is a reversible path $s \rightsquigarrow x \rightsquigarrow w$ in the input orientation, causing a contradiction. In summary, DEL can maintain an unreversible orientation and the pseudoarboricity correctly. □

Below, we analyze the time complexity of DEL. Like INS, DEL can find the reversible path using BFS and traverse along the directed edges within $O(|E|)$ time (Lines 4-5). However, unlike INS, DEL may invoke ReTest (Line 7), which takes $O(|E|^{3/2})$ time. As a result, the worst-case time complexity of DEL is $O(|E|^{3/2})$.

Discussions. Recall that in the worst case, the time complexity of INS ($O(|E|)$) is much better than BasicINS ($O(|E|^{3/2})$), and both DEL and BasicDEL have the same worst-case time complexity ($O(|E|^{3/2})$). However, compared to the basic algorithms, the practical performance of both INS and DEL are often substantially better for the following reasons. First, the BFS algorithm is only invoked by INS and DEL when d_v is sufficiently large, thus most random insertions and deletions can be processed within $O(1)$ time. Second, when INS and DEL need to find the reversible path, the search space of the BFS algorithm is typically very small. The reason is that, for the INS algorithm, the BFS algorithm can stop the search after encountering vertices with indegree no larger than $\max_{i\in\vec{G}} d_i - 2$. This indicates that the search space of the BFS algorithm only includes the vertices with indegrees greater than $\max_{i\in\vec{G}} d_i - 2$, which are located in the dense region of the graph. For real-world graphs, the dense region is often very small (compared to the graph), and thus BFS typically has a small search space. Similar analysis can be used for the DEL algorithm. Finally, the worst-case time complexity of $O(|E|^{3/2})$ for DEL only occurs when the pseudoarboricity decreases, which is a rare occurrence in the case of randomly deleting edges. As confirmed in our experiments, both INS and DEL are several orders of magnitude faster than BasicINS and BasicDEL respectively.

5 INCREMENTAL ALGORITHMS

In the previous section, we propose several efficient algorithms to maintain $p(G)$ when G is updated by both edge insertions and deletions (i.e., the fully-dynamic case). In this section, we consider the case when the graph is only updated by edge insertion (no edge deletion occurs). We show that without edge deletion, the pseudoarboricity maintenance algorithm can be simper and more efficient than the proposed INS algorithm (Algorithm 6). Below, we first devise a simple but efficient incremental insertion algorithm based on the connection between the pseudoarboricity and optimal orientation, followed by an improved algorithm.

Algorithm 8: INC $(\vec{G}, (u,v), p)$

```
1 Input: An optimal orientation $\vec{G} = (V, \vec{E})$, the edge $(u, v)$ to be inserted, and the
    pseudoarboricity before insertion $p$.
2 Output: The updated optimal orientation $\vec{G}$ and pseudoarboricity.
3 Suppose $d_v \leq d_u$, otherwise swap the input edge $(u, v)$;
4 $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
5 if $d_v = p + 1$ then
6     if there is a reversible path ending at $v$ then reverse the path;
7     else $p$++;
8 return $(\vec{G}, p)$;
```

5.1 A basic incremental insertion algorithm

Recall that after inserting an edge, the pseudoarboricity increases at most 1. Since the pseudoarboricity is equal to the maximum indegree of the graph with an optimal orientation, we can maintain an optimal orientation of a graph to maintain the pseudoarboricity. The proposed algorithm, called INC, is detailed in Algorithm 8. Specifically, after inserting $\langle u, v \rangle$, if $d_v < p$, the optimal orientation does not change. Thus, in this case, the pseudoarboricity also keeps unchanged. If $d_v = p + 1$ and there is a reversible path ending at v, then INC reverses the path to decrease d_v (Line 6). Note that in this case, after reversing such a reversible path, the optimal orientation is maintained. If $d_v = p + 1$ and there is no reversible path ending at v, then the pseudoarboricity must increase 1 (Line 7). This is because in this case only the vertex v having the maixmum indegree $p + 1$ and there is no reversible path in $\vec{G}$ by definition. At this moment, the maximum indegree $p + 1$ is equal to the pseudoarboricity, and the optimal orientation is also obtained. The correctness of INC can be straightforwardly guaranteed by Theorem 4.

It is easy to derive that the time complexity of INC is $O(|E|)$. Although INC has the same time complexity as INS, it is often faster than INS. This is because INC only need to consider the case when $d_v = p + 1$ (Line 5 of Algorithm 8), while INS has to consider the cases when $d_v = p$ and $d_v = p + 1$ (Line 5 of Algorithm 6). However, it is worth mentioning that INC does not maintain the unreversible orientation, thus it cannot be used in conjunction with DEL to deal with the fully-dynamic maintenance case.

5.2 An advanced incremental insertion algorithm

To further improve the efficiency of INC, we propose a more efficient incremental insertion algorithm, called INS++, by maintaining the unreversible orientation. INS++ is similar to INS, except that INS++ additionally maintains a carefully-defined structure D_{top}, which contains a set of high-indegree vertices. The motivation of why INS++ maintains such a D_{top} structure is as follows. Recall that if $d_v = p$, INS needs to invoke BFS to search for a reversible path ending at v. If such a path does not exist, it will consume $O(|E|)$ time. If we can previously maintain a set of vertices that can reach a vertex with indegree equaling p, i.e., the D_{top} structure, then we can easily determine whether there exists a reversible path ending at v using $O(1)$ time, significantly reducing the number of BFS calls. Formally, the definition of D_{top} is given as follows.

Definition 3. *Given an unreversible orientation $\vec{G} = (V, \vec{E})$, $D_{top} \triangleq \{i \in V | d_i = p(G)$ or i can reach a vertex with indegree equaling $p(G)\}$.*

Algorithm 9: INS++ $(\vec{G}, (u,v), p, D_{top})$

```
1 Input: An unreversible orientation $\vec{G} = (V, \vec{E})$, the edge $(u, v)$ to be inserted, the
    pseudoarboricity $p$, and the $D_{top}$ before insertion.
2 Output: The updated unreversible orientation $\vec{G}$, pseudoarboricity and $D_{top}$.
3 Suppose $d_v \leq d_u$, otherwise swap the input edge $(u, v)$;
4 $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
5 if ($v \in D_{top}$ and $d_v = p + 1$) or ($v \notin D_{top}$ and $d_v = p$) then
6     if there is a reversible path ending at $v$ then reverse the path;
7 if $v \notin D_{top}$ and $d_v = p$ then $D_{top} \leftarrow D_{top} \cup$ {all vertices that can reach $v$} $\cup \{v\}$;
8 else if $v \in D_{top}$ and $d_v = p + 1$ then
9     $p$++; $D_{top} \leftarrow$ {all vertices that can reach $v$} $\cup \{v\}$;
10 return $(\vec{G}, p, D_{top})$;
```

By Definition 3, the vertices in D_{top} must have indegree no less than $p(G) - 1$. This is because $\vec{G}$ does not contain a reversible path ($\vec{G}$ is an unreversible orientation), the vertices that can reach a vertex with indegree equaling $p(G)$ must have indegree larger than $p(G) - 2$.

The INS++ algorithm is outlined in Algorithm 9. INS++ can efficiently handle the insertion of edge $\langle u, v \rangle$ by skipping the process of finding a reversible path if v is in D_{top} and has indegree p. In contrast, the INS algorithm needs to perform a BFS to search for a reversible path in the same situation, incurring additional time overhead. Therefore, compared to INS, INS++ is more efficient for maintaining the pseudoarboricity and the unreversible orientation. Due to the requirement of maintaining D_{top}, INS++ is not compatible with DEL in handling the fully-dynamic case. It is important to note that unlike edge insertion, maintaining D_{top} for edge deletion is challenging. For edge insertion, a single BFS is sufficient to identify all reachable vertices from v and update D_{top} accordingly. However, in the case of edge deletion, there is no direct correlation between changes in D_{top} and the set of vertices reachable from v. Therefore, it is not straightforward to utilize a single BFS to determine the alterations in D_{top}, and maintaining D_{top} during edge deletions can be significantly time-consuming.

INS++ can also be faster than INC benefiting from the lower costs for finding reversible path using BFS. Specifically, since all edges between D_{top} and $V \setminus D_{top}$ are toward $V \setminus D_{top}$, the search space of the BFS algorithm used in INS++ is limited to D_{top}, which is often very small in real-world graphs. However, the search space of the BFS algorithm used in INC can be very large. Despite the additional cost involved in maintaining D_{top} in INS++, it is practically faster compared to INC, as confirmed in our experiments.

Theorem 8. *INS++ can correctly update the unreversible orientation, pseudoarboricity, and D_{top}.*

Proof. It can be easily derived from the correctness of INS that INS++ can update the unreversible orientation and pseudoarboricity. For brevity, we do not prove it again. Here we show that INS++ can correctly maintain D_{top} by the following three cases, *i.e.*, the 'if' conditions in Line 7 and Line 8. (i) $v \notin D_{top}$ and $d_v = p$. In this case, regardless of whether INS++ can discover a reversible path in Line 6, the reachability of D_{top} remains unaltered. Moreover, given that v emerges as a new vertex with a maximum indegree of p, INS++ updates D_{top} by incorporating the vertices that can reach v, along with v itself (Line 7); (ii) $v \in D_{top}$ and $d_v = p + 1$. This case indicates that v becomes the only vertex with the maximum indegree. Thus D_{top} needs to be updated as the union of all vertices that can reach v and v (Lines 8-9); (iii) ($v \notin D_{top}$ and $d_v \leq p-1$) or ($v \in D_{top}$ and $d_v = p$). In this situation, no reversible path exists in $\vec{G} \cup \langle u, v\rangle$. Therefore, v has no impact on the reachability of the vertices in D_{top} and D_{top} remains unchanged. In summary, the INS++ algorithm maintains D_{top} correctly. □

Discussion. Since every vertex in D_{top} has an indegree of at least $p(G)-1$, the density of the subgraph induced by D_{top}, denoted by $\rho(D_{top})$, satisfies $\rho(D_{top}) \geq p(G)-1$. That is to say, the difference between $\rho(D_{top})$ and $\rho(G)$ (the densest subgraph's density) will be no more than 1. As a consequence, the subgraph induced by D_{top} is a highly-dense subgraph which can be used to detect communities in real-life networks. A nice feature of the INS++ algorithm is that it not only identifies such a highly-dense subgraph, but it can also efficiently and incrementally maintain it.

6 EXPERIMENTS

6.1 Experimental Setup

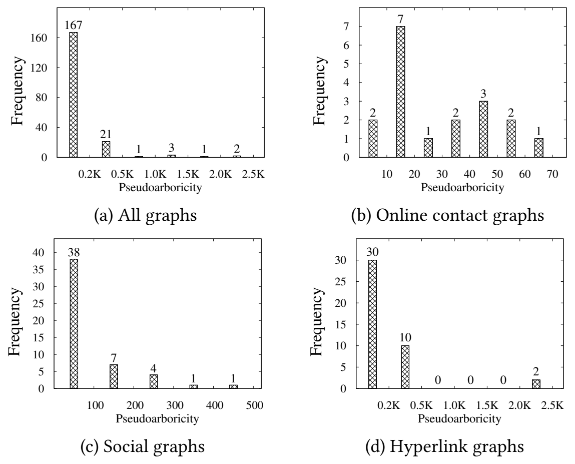
Different algorithms. In our experiments, we implement three approximate algorithms: DEGREE (Algorithm 1), iDEGREE (Algorithm 3), and INDEGREE (Algorithm 4), which calculate the maximum indegree of orientation as an estimation of pseudoarboricity. To exactly compute the pseudoarboricity, we implement the state-of-the-art algorithm, namely DEGREE+ReTest, and our two improved algorithms iDEGREE+ReTest and INDEGREE+ReTest. Since pseudoarboricity is the round-up value of the densest subgraph density, we also compare these three exact algorithms with Convex, which is the state-of-the-art algorithm for computing the densest subgraph [19]. For Convex, we use their original C++ implementation [19] for comparison. For fully-dynamic algorithms, we implement the basic maintenance algorithms BasicINS (Algorithm 5) and BasicDEL, the improved maintenance algorithms INS and DEL (Algorithm 6 and Algorithm 7). For incremental algorithms, we implement both INC (Algorithm 8) and INS++ (Algorithm 9).

Datasets. We collect 195 real-life graphs with various types downloaded from the Network Repository [41] and the Koblenz Network Collection (http://konect.cc/). The detailed statistics of the datasets are summarized in Table 1.

All algorithms are implemented in C++, utilizing the gcc compiler with O3 optimization. All experiments are conducted on a PC with a 2.2GHz AMD 3990X 64-Core CPU and 256GB memory, running the CentOS Linux operating system.

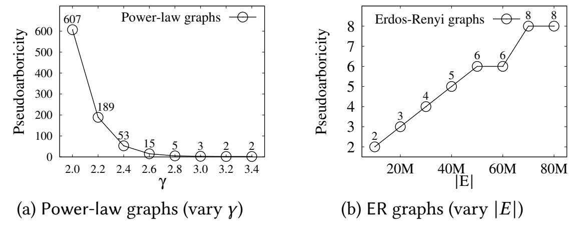
6.2 Pseudoarboricity of Various Graphs

Pseudoarboricity of real-world graphs. In this experiment, we systematically evaluated the pseudoarboricity of 195 real-life graphs with various types, and the results are reported in Table 1. As can be seen, the pseudoarboricities of citation graphs, online contact graphs, infrastructure graphs, technological graphs, software graphs, and lexical graphs are often very small. However, some other types of large graphs, such as biological graphs, collaboration graphs, and hyperlink graphs, may have significantly large pseudoarboricity. For example, the Hollywood collaboration graph has a pseudoarboricity of 1,104, and the SKALL hyperlink graph has a pseudoarboricity of 2,258.



(a) All graphs (b) Online contact graphs (c) Social graphs (d) Hyperlink graphs

Figure 4: Pseudoarboricity distribution of real-world graphs.



(a) Power-law graphs (vary γ) (b) ER graphs (vary $|E|$)

Figure 5: Pseudoarboricities of random graphs ($|V| = 10M$).

Fig. (4) depicts the distributions of pseudoarboricity for different types of graphs. Clearly, 167 out of the 195 real-world graphs (85%) have a pseudoarboricity of less than 200 in Fig. (4a), validating that most real-world graphs have small pseudoarboricities. Fig. (4b) shows that the pseudoarboricities of all online contact graphs are less than 70. In Fig. (4c), over 88 percents of social graphs have small pseudoarboricities less than 200. In Fig. (4d), most hyperlink graphs have small pseudoarboricity, while large hyperlink graphs tend to have very large pseudoarboricities. These results suggest that the "small-(pseudo)arboricity" assumption made in many existing works [15, 28, 36, 37] may be excessively optimistic for large hyperlink graphs.

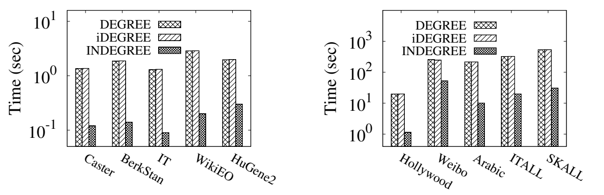
Pseudoarboricity of synthetic graphs. We generate two sets of random graphs with 10 million vertices: the power-law random graphs and the Erdos-Renyi (ER) graphs. For the power-law random graphs, we vary the power-law exponent γ from 2 to 3.4 as most real-world power-law graphs fall within this range [8]. For the ER graphs, we vary the number of edges from 10 million to 80 million. We calculate the pseudoarboricity of these synthetic graphs and depict the results in Fig. (5). As shown in Fig. (5a), the pseudoarboricity of the power-law graphs decreases with an increasing γ. Furthermore, the pseudoarboricity of the power-law graph is significantly small in the case of $\gamma > 2.2$. These findings further confirm that the pseudoarboricity of most real-world graphs is small. From Fig. (5b), the pseudoarboricity of the ER graphs increases as $|E|$ grows, but at a sluggish rate. This is because, in ER random graphs, the existence of an edge is independent of others, often leading to a uniform

Table 1: Networks Statistics and the Pseudoarboricity Results (1K=1,000,1M=1,000,000,and 1G=1,000,000,000)

Networks	Name	$\|V\|$	$\|E\|$	p	Name	$\|V\|$	$\|E\|$	p	Name	$\|V\|$	$\|E\|$	p
Biological	Diseasome	0.5K	1.2K	6	G-Worm	3.5K	6.5K	8	CE-GN	2.2K	53.7K	40
	ecoMangwet	0.1K	1.4K	17	G-FisYeast	2.0K	12.6K	27	DR-CX	3.3K	84.9K	66
	Yeast	1.5K	1.9K	3	G-Fruitfly	7.3K	24.9K	10	HS-CX	4.4K	108.8K	65
	Celegans	0.5K	2.0K	8	G-Human	9.4K	31.2K	10	G-Yeast	6.0K	156.9K	56
	ecoFoodweb	0.1K	2.1K	18	SC-GT	1.7K	34.0K	47	CE-CX	15.2K	246.0K	55
	ecoFlorida	0.1K	2.1K	18	SC-CC	2.2K	34.9K	36	**HuGene2**	**14.0K**	**9.0M**	**1,326**
	G-Plant	1.7K	3.1K	9	HS-LC	4.2K	39.5K	44	HuGene1	21.9K	12.3M	1,452
	DM-HT	3.0K	4.7K	6	CE-PG	1.9K	47.8K	62	MoGene	43.1K	14.5M	809
Collaboration	caNetscience	1.5K	2.7K	10	caHepPh	11.2K	117.6K	119	caCoCite	22.9K	2.4M	317
	caErdos	6.9K	11.9K	8	caAstroPh	17.9K	197.0K	33	caIMDB	896.3K	3.8M	21
	caGrQc	4.2K	13.4K	23	caCiteseer	227.3K	814.1K	43	caDBLP	540.5K	15.2M	168
	caCondMat	21.4K	91.3K	14	caMath	391.5K	873.8K	12	**Hollywood**	**1.1M**	**56.3M**	**1,104**
Citation	ctDBLP	12.6K	49.6K	11	ctCiteseer	384.1K	1.7M	14	ctHepPh	28.1K	3.1M	266
	ctCora	23.2K	89.2K	10	ctHepTh	22.9K	2.4M	317	ctPatent	3.8M	16.5M	41
Online contact	emUniv	1.1K	5.5K	8	WikiTalkLV	41.4K	50.6K	13	comMath	24.8K	188.0K	65
	emDNC	0.9K	10.4K	41	emEU	32.4K	54.4K	19	comEnron	87.0K	297.5K	44
	comCore	0.9K	10.4K	13	WikiTalkEL	40.3K	72.1K	22	emEuAll	265.0K	364.5K	33
	comUc	1.9K	13.8K	17	comDIGG	30.4K	85.2K	9	comAsk	157.2K	455.7K	42
	WikiTalkEO	7.6K	15.6K	18	emEnLarge	33.7K	180.8K	38	comSuper	192.4K	714.6K	54
	comPGP	10.7K	24.3K	20	comFBwal	45.8K	183.4K	14	comWiki	138.6K	715.9K	51
Infrastructure	Euro	1.2K	1.4K	2	US1	129.2K	165.4K	2	Italy	6.7M	7.0M	2
	USAir97	0.3K	2.1K	18	PA	1.1M	1.5M	2	Britain	7.7M	8.2M	2
	Power	4.9K	6.6K	4	Belgium	1.4M	1.5M	2	Germany	11.5M	12.4M	2
	Openflights	2.9K	15.7K	23	Netherlands	2.2M	2.4M	2	Asia	12.0M	12.7M	2
	Luxembourg	114.6K	119.7K	2	CA	2.0M	2.8M	2	US2	23.9M	28.9M	2
Social	FbNIPS	2.9K	3.0K	3	WikiElec	7.1K	100.8K	47	LiveMocha	104.1K	2.2M	84
	HamHouse	1.6K	4.0K	11	GemsecRO	41.8K	125.8K	6	Buzznet	101.2K	2.8M	138
	HamFriship	1.9K	12.5K	17	fbMedia	27.9K	206.0K	19	YTBsnap	1.1M	3.0M	46
	HamFriend	3.0K	12.5K	17	Brightkite	58.2K	214.1K	41	FourSq	639.0K	3.2M	56
	Hamster	2.4K	16.6K	18	GemsecHU	47.5K	222.9K	10	Flickr	514.0K	3.2M	254
	fbTvshow	3.9K	17.2K	31	Douban	154.9K	327.2K	14	Lastfm	1.2M	4.5M	62
	TwitList	23.4K	32.8K	8	Slashdot1	77.4K	469.2K	43	wikiTalk	2.4M	4.7M	115
	Ciaodvd	4.7K	33.1K	36	GemsecHR	54.6K	498.2K	17	**Caster**	**149.7K**	**5.4M**	**348**
	Gplus	23.6K	39.2K	11	Slashdot2	82.2K	504.2K	44	DIGG	770.8K	5.9M	215
	Advogato	5.2K	39.4K	23	Academia	190.2K	788.3K	17	Flixster	2.5M	7.9M	51
	fbPoli	5.9K	41.7K	25	fbArtist	50.5K	819.1K	59	Dogster	426.8K	8.5M	218
	Anybeat	12.6K	49.1K	29	TwiFollows	465.0K	833.5K	26	twiHiggs	456.6K	12.5M	115
	fbCom	14.1K	52.1K	14	Delicious	426.4K	908.3K	18	Flickr	1.7M	15.6M	469
	fbPubFig	11.6K	67.0K	35	Gowalla	196.6K	950.3K	44	Pokec	1.6M	22.3M	42
	fbSport	13.9K	86.8K	20	Themarker	69.4K	1.6M	144	Livejournal	4.0M	27.9M	131
	fbGovern	7.1K	89.4K	37	YTB	496.0K	1.9M	44	Orkut	3.0M	106.3M	207
	Epinions	26.6K	100.1K	27	BlogCata	88.8K	2.1M	194	**Weibo**	**58.7M**	**261.3M**	**166**
Hyperlink	Polblogs	0.6K	2.3K	10	WikiIS	69.4K	907.4K	197	Wiki	1.9M	4.5M	55
	EPA	4.3K	8.9K	5	WikiFY	65.6K	921.6K	84	WikiTH	266.9K	4.6M	215
	Webbase	16.1K	25.6K	16	Notre	325.7K	1.1M	79	WikiLT	268.2K	5.1M	158
	WikiCham	2.3K	31.4K	48	WikiIA	24.0K	1.2M	280	**BerkStan**	**685.2K**	**6.6M**	**104**
	Spam	4.8K	37.4K	28	WikiAF	72.3K	1.5M	187	**IT**	**509.3K**	**7.2M**	**216**
	Indochina	11.4K	47.6K	25	IkArabic	163.6K	1.7M	51	**WikiEO**	**413.0K**	**8.2M**	**354**
	WikiCO	8.3K	119.8K	121	WikiAST	83.3K	2.0M	75	WikiCh	1.9M	9.0M	95
	GoogleInter	15.8K	149.5K	55	Stanford	281.9K	2.0M	60	UK	129.6K	11.7M	250
	WikiCroc	11.6K	170.9K	51	BaiduRe	415.6K	2.4M	183	Hudong	2.0M	14.4M	157
	WikiSqui	5.2K	198.5K	137	Italycnr	325.6K	2.7M	58	Baidu	2.1M	17.0M	73
	SK	121.4K	334.4K	42	WikiNN	215.9K	2.9M	133	WikiUK	1.2M	41.9M	459
	WikiYO	41.2K	696.4K	242	WikiLV	190.0K	2.9M	283	UKAll	39.5M	783.0M	486
	WikiCKB	60.7K	802.1K	204	WikiLA	181.2K	3.0M	140	**ITALL**	**41.3M**	**1.0G**	**2,009**
	WikiSW	58.8K	877.0K	157	Google	875.7K	4.3M	29	**SKALL**	**50.6M**	**1.8G**	**2,258**
Technological	Routers	2.1K	6.6K	12	WHOIS	7.5K	56.9K	63	RLCaida	190.9K	607.6K	26
	PGP	10.7K	24.3K	20	Internet	40.2K	85.1K	18	Skitter	1.7M	11.1M	90
	Caida	26.5K	53.4K	18	P2P	62.6K	147.9K	5	IP	2.3M	21.6M	200
Software	Jung	6.1K	50.3K	47	JDK	6.4K	53.7K	47	Linux	30.8K	213.2K	21
Lexical	EAT	23.1K	297.1K	31	Wordnet	146.0K	657.0K	17	Yahoo	653.3K	2.9M	24
Miscellaneous	HyperText	0.1K	2.2K	21	Beaflw	0.5K	45.3K	97	misAmazon2	403.4K	2.4M	10
	Infectious	0.4K	2.8K	11	Orani	2.5K	87.0K	122	misDBpedia	4.0M	12.6M	16
	misTwin	14.3K	20.6K	13	misAmazon1	334.9K	925.9K	5	misActor	382.2K	15.0M	310
	Beacxc	0.4K	42.6K	93	misFlickr	105.9K	2.3M	292	**Arabic**	**22.7M**	**553.9M**	**1,625**

Table 2: Approximation performance of different algorithms.

Dataset	p	DEGREE	iDEGREE	INDEGREE	Iterations
Caster	348	419	380	**349**	8
BerkStan	104	201	129	**104**	7
IT	216	431	234	**216**	2
WikiEO	354	688	410	**354**	5
HuGene2	1,326	1,902	1,525	**1,326**	7
Hollywood	1,104	2,208	1,249	**1,105**	4
Weibo	166	193	179	**170**	12
Arabic	1,625	3,247	1,840	**1,625**	4
ITALL	2,009	3,224	2,327	**2,009**	7
SKALL	2,258	4,510	2,641	**2,258**	7



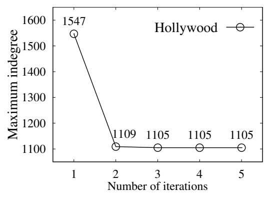
(a) Time cost (medium-sized graphs) (b) Time cost (large-sized graphs)

Figure 6: Runtime of different approximation algorithms.

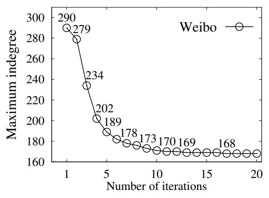
density distribution. Achieving a high pseudoarboricity requires a substantially denser subgraph, which is difficult to obtain when the density is uniform.

6.3 Performance Studies

Comparison of different approximation algorithms. Table 2 shows the maximum indegree of the approximate orientation calculated by DEGREE, iDEGREE and INDEGREE respectively. As



(a) Hollywood



(b) Weibo

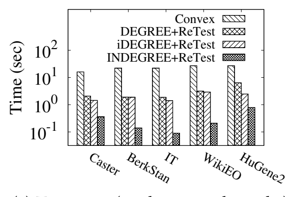
Figure 7: The maximum indegree decreasing rate of INDEGREE**.**

expected, the approximation performance of DEGREE is unsatisfactory. The proposed iDEGREE algorithm exhibits an improvement over DEGREE in terms of the maximum indegree, but it is not substantial. While our INDEGREE achieves the best approximation performance, with the discrepancy between the maximum indegree of orientation and the pseudoarboricity not exceeding 4 on all datasets. For example, on dataset Hollywood with $p = 1,104$, the maximum indegree with DEGREE is twice as the pseudoarboricity, which achieves the worst-case result. The maximum indegree of orientations yielded by iDEGREE and INDEGREE are $1,325$ and $1,105$, respectively. In addition, we also show the iteration numbers of INDEGREE in Table 2. As can be seen, INDEGREE terminates within a few iterations over all datasets.

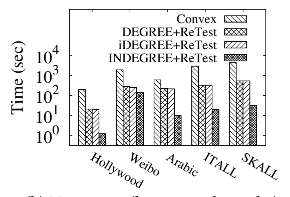
The runtime of DEGREE, iDEGREE and INDEGREE is illustrated in Fig. (6). We can clearly see that our INDEGREE algorithm achieves the lowest runtime among all approximation algorithms, and the runtime of iDEGREE is almost the same as that of DEGREE. In general, the running time of INDEGREE is around 4.8-21.3 times lower than that of DEGREE/iDEGREE on all datasets. These results demonstrate that the proposed INDEGREE algorithm substantially outperforms the other approximation algorithms in terms of both approximation performance and running time.

Iteration termination condition of INDEGREE**.** Here we study the rate at which the maximum indegree decreases as the iterations increase for INDEGREE. The results on Hollywood and Weibo are shown in Fig. (7), and similar results can also be observed on other datasets. As can be seen, the decreasing rate varies significantly across different graphs, indicating that a fixed number of iterations is not appropriate for INDEGREE. Moreover, the maximum indegree decreases rapidly in the first few iterations, and then gradually converges to the pseudoarboricity. Performing fewer iterations may cause the maximum indegree of the orientation graph to be not small enough, resulting in a poor approximation of pseudoarboricity; while the excessive number of iterations can yield a better approximation, but also increases the time overheads. To achieve the trade-off between the number of iterations and the approximate effect, we use "the maximum indegree does not decrease in current iteration" as the termination condition, which is typically efficient and effective. For example, with this termination condition, INDEGREE terminates after 4 and 12 iterations on Hollywood and Weibo respectively.

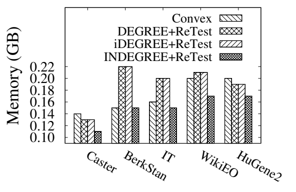
Comparison of exact algorithms. Fig. (8a) and Fig. (8b) depict the runtime of Convex, DEGREE+ReTest, iDEGREE+ReTest and INDEGREE+ReTest on 10 real-life datasets. We can see that our novel solution INDEGREE+ReTest consistently outperforms other



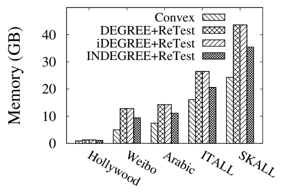
(a) Time cost (medium-sized graphs)



(b) Time cost (large-sized graphs)

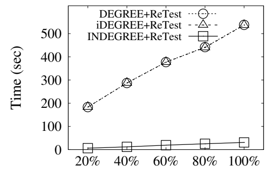


(c) Memory cost (medium-sized graphs)

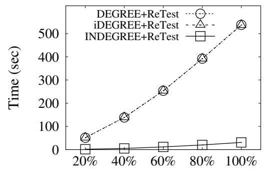


(d) Memory cost (large-sized graphs)

Figure 8: Results of eaxct algorithms for pseudoarboricity computation.



(a) vary $|V|$



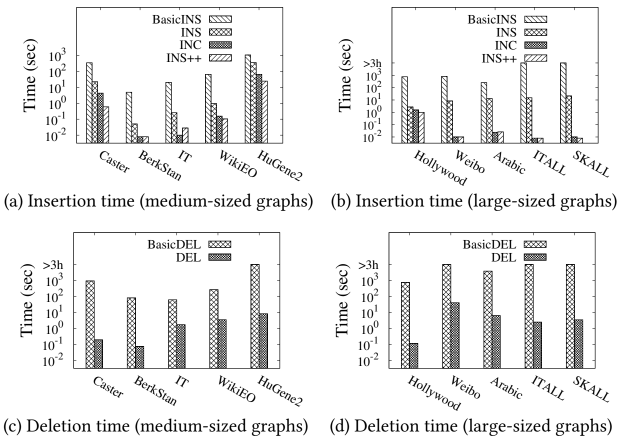
(b) vary $|E|$

Figure 9: Scalability of different algorithms on SKALL**.**

algorithms over all datasets. The Convex algorithm has the longest running time because it computes the densest subgraph to obtain pseudoarboricity, which often requires large numbers of iterations. For DEGREE+ReTest and iDEGREE+ReTest, they are slightly more complicated to implement compared to INDEGREE+ReTest, slowing down the overall computation process. Besides, the approximation performance of DEGREE is often unsatisfactory, resulting in the need to conduct a binary search. Therefore, ReTest needs to be invoked by around $\log(p)$ times, which is also time-consuming. For example, on the largest dataset SKALL, the runtime of Convex, DEGREE+ReTest, iDEGREE+ReTest, and INDEGREE+ReTest are 4,528, 537.84, 536.49, and 31.07 seconds respectively. These results demonstrate that our INDEGREE+ReTest algorithm is at least one order of magnitude faster than the state-of-the-art algorithms for exact pseudoarboricity computation on real-life graphs.

Fig. (8c) and Fig. (8d) show the memory costs of the four exact algorithms. As can be seen, the four algorithms exhibit similar memory overheads, because they all require linear space. Furthermore, the memory usage of INDEGREE+ReTest is less than that of DEGREE+ReTest and iDEGREE+ReTest. Compared to Convex, although INDEGREE+ReTest consumes slightly more memory space, the runtime of INDEGREE+ReTest is around 13-250 times faster than that of Convex. The results confirm that the proposed INDEGREE+ReTest algorithm is highly space-efficient.

Scalability testing. We evaluate the scalability of the proposed algorithms using a large dataset SKALL. The results on the other datasets are consistent. We randomly select 20%-80% of vertices (edges) to generate four subgraphs of SKALL and record the runtime of DEGREE+ReTest, iDEGREE+ReTest, and INDEGREE+ReTest on these subgraphs. The results are shown in Fig. (9). We can observe



(a) Insertion time (medium-sized graphs) (b) Insertion time (large-sized graphs)

(c) Deletion time (medium-sized graphs) (d) Deletion time (large-sized graphs)

Figure 10: Results of dynamic algorithms for pseudoarboricity maintenance.

that both the runtime of the DEGREE+ReTest and iDEGREE+ReTest increase sharply with increasing $|V|$ and $|E|$, while the runtime of INDEGREE+ReTest changes very smoothly. Moreover, INDEGREE+ReTest is significantly faster than the other algorithms on all subgraphs, which is consistent with our previous findings. These results show the high scalability of our proposed INDEGREE+ReTest algorithm.

Exp-7: Runtime of dynamic algorithms. In this experiment, we evaluate the performance of different maintenance algorithms. In the fully dynamic case, we randomly delete and insert 10,000 edges from the input graph. In the incremental insertion case (no edge deletion), we insert the same set of 10,000 edges as in the fully dynamic case. We repeatedly perform all algorithms five times on each dataset with different deleted/inserted edges, and average the results to obtain the final time cost. The results are shown in Fig. (10). For the fully-dynamic case, the improved maintenance algorithms (INS and DEL) consistently outperform the basic ones (BasicINS and BasicDEL). For instance, when processing the large graph Hollywood, INS and DEL consumes only 2.676 seconds and 0.114 seconds, respectively, which are 2-3 orders of magnitude faster than BasicINS and BasicDEL. For the incremental insertion case, the proposed two algorithms INC and INS++ are incredibly fast on most datasets. Both of them take less than 0.01 seconds to incrementally maintain the pseudoarboricity of BerkStan, Weibo, ITALL, and SKALL. For Caster and HuGene2, the runtime of INS++ is significantly lower than that of INC. Specifically, INS++ takes only 0.598 seconds and 24.224 seconds to update the pseudoarboricity for these datasets respectively, whereas INC takes 4.286 seconds and 64.32 seconds. These results demonstrate the high-efficiency of the proposed algorithms in handling dynamic graphs.

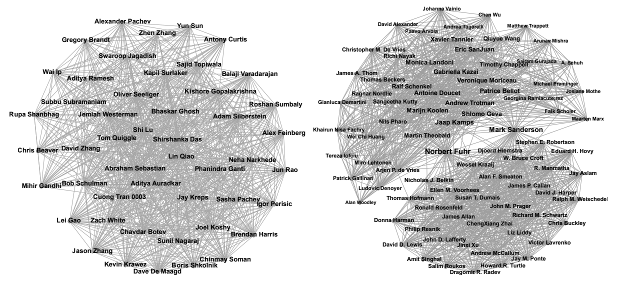
Furthermore, the experimental results reveal that the time to process edge insertion and edge deletion is not significantly related to the size of the graph. Rather, the execution time is mostly influenced by the inner structure of the graphs, which determines the possibility of whether an edge can be processed in $O(1)$ time. In Table 3, we present the statistics for the number of updates completed in $O(1)$ time by algorithms DEL, INS, INC, and INS++,

Table 3: Numbers of $O(1)$ processes for dynamic algorithms.

Dataset	DEL	INS	INC	INS++
Caster	9,255	8,436	9,351	9,573
BerkStan	9,761	9,941	9,990	9,992
IT	8,019	9,611	9,988	9,982
WikiEO	9,490	9,666	9,919	9,940
HuGene2	7,628	5,501	6,916	7,266
Hollywood	9,745	9,019	9,250	9,389
Weibo	7,126	9,987	9,514	9,987
Arabic	9,232	9,155	9,990	9,988
ITALL	9,868	9,302	9,990	9,990
SKALL	9,834	9,991	9,419	9,991

Table 4: Density of δ-core, D_{top}, and the densest subgraph

Dataset	$\rho(\delta\text{-core})$	$\rho(D_{top})$	$\rho(G)$
Caster	310.860	347.843	347.848
BerkStan	103.406	103.406	103.409
IT	215.500	215.499	215.503
WikiEO	344.499	353.179	353.182
HuGene2	**1,130.941**	**1,325.175**	**1,325.180**
Hollywood	1,104.000	1,104.000	1,104.000
Weibo	**107.618**	**165.408**	**165.418**
Arabic	1,623.500	1,624.460	1,624.460
ITALL	**1,619.461**	**2,008.185**	**2,008.186**
SKALL	2,256.499	2,257.495	2,257.495



(a) D_{top}(DB) (b) D_{top}(IR)

Figure 11: D_{top} of DB and IR.

under 10,000 random updates. This implies that no BFS algorithm or network flow algorithm was invoked for these updates. When running the INS++ algorithm, despite the size of SKALL being much larger than HuGene2, over 99.9% of insertions on SKALL can be completed in $O(1)$ time, while this percentage does not exceed 75% on HuGene2. Overall, except for BasicINS and BasicDEL, our proposed dynamic algorithms can handle over 90% insertions or deletions within $O(1)$ time on most datasets.

6.4 Application for Community Detection

As discussed in Section 5.2, the density difference between D_{top} and the densest subgraph is no more than 1. Thus, similar to the densest subgraph, D_{top} can be used to detect communities in real-life graphs. Table 4 reports the densities of δ-core (δ is the maximum core number of all vertices), the densest subgraph, and D_{top} on 10 datasets. We can clearly see that $\rho(D_{top})$ differs from $\rho(G)$ by at most 0.01 on all datasets. Moreover, on Hollywood, Arabic, and SKALL, the subgraph induced by D_{top} achieves the same density as the densest subgraph. Conversely, the density of δ-core is significantly lower than $\rho(G)$, especially on HuGene2, Weibo, and ITALL. These results confirm that D_{top} is indeed very effective to detect dense subgraphs in real-world graphs.

We also conduct case studies on two subgraphs extracted from the collaboration graph DBLP (https://dblp.org/), namely, DB and IR. DB consists of authors who published at least one paper in the database and data mining related conferences with 37,177 vertices and 131,715 edges. IR includes authors who published at least one paper in information retrieval related conferences with 13,445 vertices and 37,428 edges. We compute the D_{top} of both graphs, denoted by D_{top}(DB) and D_{top}(IR), and depict the results in Fig. (11). We also compute the densest subgraphs on both DB and IR. We find that D_{top} is exactly the densest subgraph on DB, and the densest subgraph on IR is D_{top} after deleting the vertex "Alan Woodley". As expected, the authors in D_{top}(DB) or D_{top}(IR) have a strong cooperative relationship. Specifically, all authors in D_{top}(DB) are researchers who work in LinkedIn and have co-published two papers about LinkedIn's database [6, 40]. As for authors in D_{top}(IR), they can be divided into two groups: one group of authors are members of the Initiative for the Evaluation of XML retrieval (INEX) [2], while another group of authors has co-published a workshop report about information retrieval [3]. Norbert Fuhr and Mark Sanderson belong to both groups, thus connecting them. These results confirm the effectiveness of D_{top} to identify densely-connected communities.

7 RELATED WORK

Arboricity. Arboricity, a fundamental metric of a graph, is the minimum number of edge-disjoint forests that can partition the edges of a graph. A common application of arboricity is to bound the space or time complexity of graph algorithms, such as triangle counting [28], k-clique enumeration [15], truss decomposition [32, 43], structural graph clustering [12], influential community [35, 36], structural diversity search [13, 29]. However, calculating arboricity is often challenging, and existing algorithms rely mainly on matroid theory. For example, Picard and Queyranne showed that arboricity can be computed by Edmonds' matroid partitioning algorithm [20] in polynomial time, and propose an improved algorithm with the time complexity of $O(|V|^2|E|\log^2|V|)$ [39]. Gabow and Westermann devised an algorithm based on the matroid sums technique with $O(a^{3/2}|V|\sqrt{|E|})$ time complexity [25]. Subsequently, Gabow improves the time complexity to $O(|E|^{3/2}\log(|V|^2/|E|))$ using graphic polymatroid, which is the SOTA algorithm for arboricity computation in theory [24]. Although this algorithm can achieve nearly the same time complexity as the pseudoarboricity computation algorithm, it heavily relies on the matroid related computation which is often much more expensive than the re-orientation flow algorithm in computing the pseudoarboricity.

Minimizing the maximum indegree. Our work is related to the problem of minimizing the maximum indegree, which aims to find an optimal orientation with the smallest maximum indegree. It was shown that the smallest maximum indegree is equal to pseudoarboricity [9]. The optimal orientation can be utilized to construct a graph data structure that can efficiently determine whether two vertices are adjacent [1, 4, 16]. The best-known re-orientation network for computing the optimal orientation was proposed by Bezakova [9], which is based on a binary search method with time complexity of $O(|E|^{3/2}\log p)$. Then, numerous approximate algorithms are proposed to calculate optimal orientations [5, 21, 33]. With these approximate algorithms, Blumenstock presented an improved algorithm based on Bezakova's method, which achieves the SOTA performance for exact pseudoarboricity computation [10]. In this work, we propose an exact algorithm by integrating two novel approximation techniques to further improve the efficiency. In addition, to the best of our knowledge, we are the first to investigate the problem of maintaining the optimal orientation (and also the pseudoarboricity) in dynamic graphs.

Densest subgraph mining. Our work is also closely related to the densest subgraph mining problem. As a fundamental problem in network analysis, the densest subgraph problem has a wide range of applications such as community detection [14], network visualization [46], and fraud detection [27]. To compute the densest subgraph, several network flow and convex programming algorithms are devised [17, 19, 26]. However, these algorithms are often time-consuming for large graphs. In this work, we propose a novel approximation of the densest subgraph: D_{top} (Definition 3), whose density is no smaller than the densest subgraph density minus 1.

8 CONCLUSION

In this paper, we study the problem of computing pseudoarboricity in static and dynamic graphs. For static graphs, we propose two new and efficient approximation algorithms (with various theoretical guarantees) to approximate the pseudoarboricity. With our approximation algorithms, we can significantly reduce the search space of the exact pseudoarboricity computation algorithms. For fully-dynamic graphs with both edge insertions and deletions, we first present a pseudoarboricity update theorem, based on which two novel pseudoarboricity maintenance algorithms are proposed. We have also developed two new incremental pseudoarboricity maintenance algorithms specifically for scenarios where the dynamic graph only involves edge insertions (no edge deletion). We conduct extensive experiments on 195 real-world graphs. The results suggest that most real-world graphs indeed have a small pseudoarboricity, except for a few large biological graphs, collaboration graphs and hyperlink graphs. The results also demonstrate the high efficiency and scalability of the proposed algorithms for pseudoarboricity computation in both static and dynamic graphs.

REFERENCES

[1] Oswin Aichholzer, Franz Aurenhammer, and Günter Rote. 1995. *Optimal graph orientation with storage applications.* Universität Graz/Technische Universität Graz. SFB F003-Optimierung und Kontrolle.

[2] David Alexander, Paavo Arvola, Thomas Beckers, Patrice Bellot, Timothy Chappell, Christopher M. De Vries, Antoine Doucet, Norbert Fuhr, Shlomo Geva, Jaap Kamps, Gabriella Kazai, Marijn Koolen, Sangeetha Kutty, Monica Landoni, Véronique Moriceau, Richi Nayak, Ragnar Nordlie, Nils Pharo, Eric SanJuan, Ralf Schenkel, Andrea Tagarelli, Xavier Tannier, James A. Thom, Andrew Trotman, Johanna Vainio, Qiuyue Wang, and Chen Wu. 2011. Report on INEX 2010. *SIGIR Forum* 45, 1 (2011), 2–17.

[3] James Allan, Jay Aslam, Nicholas J. Belkin, Chris Buckley, James P. Callan, W. Bruce Croft, Susan T. Dumais, Norbert Fuhr, Donna Harman, David J. Harper, Djoerd Hiemstra, Thomas Hofmann, Eduard H. Hovy, Wessel Kraaij, John D. Lafferty, Victor Lavrenko, David D. Lewis, Liz Liddy, R. Manmatha, Andrew McCallum, Jay M. Ponte, John M. Prager, Dragomir R. Radev, Philip Resnik, Stephen E. Robertson, Ronald Rosenfeld, Salim Roukos, Mark Sanderson, Richard M. Schwartz, Amit Singhal, Alan F. Smeaton, Howard R. Turtle, Ellen M. Voorhees, Ralph M. Weischedel, Jinxi Xu, and ChengXiang Zhai. 2003. Challenges in information retrieval and language modeling: report of a workshop held at the center for intelligent information retrieval, University of Massachusetts Amherst, September 2002. *SIGIR Forum* 37, 1 (2003), 31–47.

[4] Srinivasa Rao Arikati, Anil Maheshwari, and Christos D. Zaroliagis. 1997. Efficient Computation of Implicit Representations of Sparse Graphs. *Discret. Appl. Math.* 78, 1-3 (1997), 1–16.

[5] Yuichi Asahiro, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. 2007. Graph Orientation Algorithms to minimize the Maximum Outdegree. *Int. J. Found. Comput. Sci.* 18, 2 (2007), 197–215.

[6] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Boris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang. 2012. Data Infrastructure at LinkedIn. In *ICDE*. IEEE Computer Society, 1370–1381.

[7] Nikhil Bansal and Seeun William Umboh. 2017. Tight approximation bounds for dominating set on graphs of bounded arboricity. *Inf. Process. Lett.* 122 (2017), 21–24.

[8] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[9] Ivona Bezáková. 2000. Compact representations of graphs and adjacency testing. (2000).

[10] Markus Blumenstock. 2016. Fast Algorithms for Pseudoarboricity. In *ALENEX*. SIAM, 113–126.

[11] Glencora Borradaile, Jennifer Iglesias, Theresa Migler, Antonio Ochoa, Gordon T. Wilfong, and Lisa Zhang. 2017. Egalitarian Graph Orientations. *J. Graph Algorithms Appl.* 21, 4 (2017), 687–708.

[12] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang, and Shiyu Yang. 2017. pSCAN: Fast and Exact Structural Graph Clustering. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 387–401.

[13] Lijun Chang, Chen Zhang, Xuemin Lin, and Lu Qin. 2017. Scalable Top-K Structural Diversity Search. In *ICDE*. IEEE Computer Society, 95–98.

[14] Jie Chen and Yousef Saad. 2012. Dense Subgraph Extraction with Application to Community Detection. *IEEE Trans. Knowl. Data Eng.* 24, 7 (2012), 1216–1230.

[15] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[16] Marek Chrobak and David Eppstein. 1991. Planar Orientations with Low Out-degree and Compaction of Adjacency Matrices. *Theor. Comput. Sci.* 86, 2 (1991), 243–266.

[17] Nathann Cohen. 2019. Several graph problems and their linear program formulations. (2019).

[18] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. 589–598.

[19] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-friendly Graph Decomposition via Convex Programming. In *WWW*. ACM, 233–242.

[20] Jack Edmonds. 1965. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B* 69 (1965), 67–72.

[21] David Eppstein. 1994. Arboricity and Bipartite Subgraph Listing Algorithms. *Inf. Process. Lett.* 51, 4 (1994), 207–211.

[22] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM J. Exp. Algorithmics* 18 (2013).

[23] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518.

[24] Harold N. Gabow. 1998. Algorithms for Graphic Polymatroids and Parametris s-Sets. *J. Algorithms* 26, 1 (1998), 48–86.

[25] Harold N. Gabow and Herbert H. Westermann. 1992. Forests, Frames, and Games: Algorithms for Matroid Sums and Applications. *Algorithmica* 7, 5&6 (1992), 465–497.

[26] Andrew V Goldberg. 1984. Finding a maximum density subgraph. (1984).

[27] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. FRAUDAR: Bounding Graph Fraud in the Face of Camouflage. In *KDD*. ACM, 895–904.

[28] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *SIGMOD Conference*. ACM, 325–336.

[29] Jinbin Huang, Xin Huang, and Jianliang Xu. 2022. Truss-Based Structural Diversity Search in Large Graphs. *IEEE Trans. Knowl. Data Eng.* 34, 8 (2022), 4037–4051.

[30] Xin Huang, Hong Cheng, Rong-Hua Li, Lu Qin, and Jeffrey Xu Yu. 2013. Top-K Structural Diversity Search in Large Networks. *Proc. VLDB Endow.* 6, 13 (2013), 1618–1629.

[31] Xin Huang, Hong Cheng, Rong-Hua Li, Lu Qin, and Jeffrey Xu Yu. 2015. Top-K structural diversity search in large networks. *VLDB Journal.* 24, 3 (2015), 319–343.

[32] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD Conference*. ACM, 1311–1322.

[33] Lukasz Kowalik. 2006. Approximation Scheme for Lowest Outdegree Orientation and Graph Density Measures. In *ISAAC (Lecture Notes in Computer Science, Vol. 4288)*. Springer, 557–566.

[34] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* 13, 11 (2020), 2536–2548.

[35] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *Proc. VLDB Endow.* 8, 5 (2015), 509–520.

[36] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2017. Finding influential communities in massive networks. *VLDB J.* 26, 6 (2017), 751–776.

[37] Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. 2012. Arboricity, h-index, and dynamic algorithms. *Theor. Comput. Sci.* 426 (2012), 75–90.

[38] C St JA Nash-Williams. 1961. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society* 1, 1 (1961), 445–450.

[39] Jean-Claude Picard and Maurice Queyranne. 1982. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks* 12, 2 (1982), 141–159.

[40] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradkar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jagadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *SIGMOD Conference*. ACM, 1135–1146.

[41] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. AAAI Press, 4292–4293.

[42] Venkat Venkateswaran. 2004. Minimizing maximum indegree. *Discret. Appl. Math.* 143, 1-3 (2004), 374–378.

[43] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823.

[44] Qi Zhang, Rong-Hua Li, Minjia Pan, Yongheng Dai, Guoren Wang, and Ye Yuan. 2022. Efficient Top-k Ego-Betweenness Search. In *ICDE*. IEEE, 380–392.

[45] Qi Zhang, Rong-Hua Li, Qixuan Yang, Guoren Wang, and Lu Qin. 2020. Efficient Top-k Edge Structural Diversity Search. In *ICDE*. 205–216.

[46] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *Proc. VLDB Endow.* 6, 2 (2012), 85–96.