Efficient Algorithms for Pseudoarboricity Computation in Large Static and Dynamic Graphs

Yalong Zhang
Beijing Institute of Technology
Beijing, China
yalong-zhang@qq.com

Rong-Hua Li
Beijing Institute of Technology
Beijing, China
lironghuabit@126.com

Qi Zhang
Beijing Institute of Technology
Beijing, China
qizhangcs@bit.edu.cn

Hongchao Qin
Beijing Institute of Technology
Beijing, China
qhc.neu@gmail.com

Lu Qin
University of Technology Sydney
Sydney, Australia
lu.qin@uts.edu.au

Guoren Wang
Beijing Institute of Technology
Beijing, China
wanggrbit@126.com

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 that 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 approximation algorithms with theoretical guarantees to approximate 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 exact pseudoarboricity computation. In addition, 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 insertion-only scenarios. 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.

1 INTRODUCTION

Arboricity is defined as the minimum number of edge-disjoint forests into which the edges of a graph can be partitioned [39, 41]. As a classic measure of graph's sparsity, arboricity has been widely used to analyze the complexity of numerous graph analysis algorithms, such as triangle counting [28, 40], k-clique listing [15, 19, 34], truss decomposition [32, 45], structural graph clustering [12, 46, 47], influential community mining [35, 36], structural diversity search [13, 29, 49], top-k ego-betweenness search [48].

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 [41], 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 [39, 41]. 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 four key applications as follows. (1) In Section 2.2, we show that pseudoarboricity can also be applied to analyze the time or space complexity of all the above-mentioned graph analysis algorithms with a tighter bound. (2) Pseudoarboricity provides a more intuitive measure of graph sparsity. It is equal to the rounded-up value of the density of the densest subgraph, which makes it easier to interpret compared to arboricity. (3) In the problem of minimizing the maximum indegree, existing studies typically rely on arboricity to achieve small indegree [4, 8]. However, pseudoarboricity can offer even smaller indegree compared to arboricity. (4) In Section 6.4, we show that the pseudoarboricity can be effectively leveraged to identify the nearly-densest subgraph in a graph, thereby emphasizing its practical value in community detection. This feature makes it particularly useful in fields such as social network analysis, where community or cluster detection is a common and fundamental task. Motivated by these observations, we focus on investigating the problem of efficiently computing the pseudoarboricity of a graph.

Previous approaches to computing 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, Bezáková proposed an algorithm that uses the re-orientation network flow and binary search techniques to compute the optimal orientation and the pseudoarboricity [9]. The key idea of Bezáková'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. Building upon Bezáková'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, 22]. 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. 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 different theoretical guarantees. 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. 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. To improve the efficiency, we propose two novel and more efficient algorithms, namely INS and DEL. 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 insertion-only scenarios. 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 community 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, 38, 49]. (2) Our best approximation algorithm consistently achieves a significantly higher level of approximation quality while requiring an order of magnitude less time than other competitors. 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 algorithms for pseudoarboricity computation on static graphs. (3) To dynamically maintain the pseudoarboricity, the proposed INS and DEL algorithms are very efficient, being 2-3 orders of magnitude faster than the basic algorithms BasicINS and BasicDEL. (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 graphs with billions of edges.

Reproducibility and full version paper. The source code and the full version of this paper can be found at: https://github.com/Flydragonet/Pseudoarboricity-Computation.

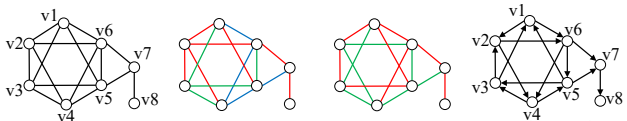
2 PRELIMINARIES

2.1 Notations and problem definition

Let $G = (V, E)$ be an unweighted and undirected graph, where V denotes the set of vertices 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 $\vec{G}$. 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 indegree is minimized, *i.e.*, the minimum k-orientation. For example, the orientation in Fig. (1d) is an optimal orientation of Fig. (1a). 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 vertex u in G (*i.e.*, the number of neighbors of u in G), and $d_u(\vec{G})$ be the indegree of u in $\vec{G}$ (*i.e.*, the number of ingoing neighbors of u in $\vec{G}$). If the context is clear, we will use d_u to denote the degree (indegree) of a u in G (in $\vec{G}$). In the directed graph $\vec{G} = (V, \vec{E})$, A *path* is a sequence of vertices $v_1 \to v_2 \to \cdots \to v_k$, 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)** [39] *The arboricity of a graph G, denoted by $a(G)$, is defined as the smallest number of edge-disjoint (*i.e.*, without overlapping edges) forests into which the edge set E of G can be partitioned, where a forest is a graph without any cycles.*

Arboricity is a classic metric in graph theory that was frequently used to measure the sparsity of a graph [15, 22]. However, arboricity is often difficult to compute on large graphs [25]. In their work



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

Figure 1: Running example.

[41], 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* [41]. A *pseudotree* is an undirected graph that is connected and contains precisely one cycle [41]. 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, pseudoarboricity can be defined as follows.

Definition 2. **(Pseudoarboricity)** [41] *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 [41], 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?

Simpler computation. The state-of-the-art algorithms [24, 25] for computing arboricity require the matroid partition algorithm, which is typically more complex than the parametric max-flow algorithm used to compute the pseudoarboricity [10].

Relationships with other concepts. In comparison to arboricity, pseudoarboricity exhibits close relationships with other concepts in graph data mining. (i) Pseudoarboricity is equal to the smallest maximum indegree among orientations of a graph [9]; (ii) Pseudoarboricity is equal to the round-up of the density of the densest subgraph of a graph [41]; (iii) Degeneracy is a 2-approximation of pseudoarboricity [11].

Analyzing the complexity. 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 [19, 34], truss decomposition [32, 45], structural graph clustering [12], structural diversity search [30, 31, 49], 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 complexity.

Algorithm 1: DEGREE (G)

Input: A simple graph $G = (V, E)$.
Output: A 2-approximate orientation $\vec{G}$.

```
1  Let Ē ← ∅, Ḡ ← (V, Ē);
2  Let core[1 ... |V|] be an array and nowcore ← 0;
3  while V is not empty do
4      u ← arg min_{u∈V} d_u(G);
5      nowcore ← max{nowcore, d_u}; core[u] ← nowcore;
6      for e = (v, u) ∈ E do
7          Ē ← Ē ∪ ⟨v, u⟩; E ← E − (v, u); d_v(G) ← d_v(G) − 1;
8      V ← V − u;
9  Let p_0 ← ⌈max d_u(Ḡ)/2⌉ and V' ← all vertices in the p_0-core of G;
10 Ḡ ← the subgraph of Ḡ induced by V';
11 return Ḡ;
```

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

Approximate algorithms. The widely-used approximate algorithm for pseudoarboricity computation is outlined in Lines 1-8 of Algorithm 1 [5, 9, 22]. 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 9-10 of Algorithm 1.

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 $p(G) = p(G')$.*

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) $\langle u, t\rangle \in A, c(u, t) = k - d_u(\vec{G}), if\ d_u(\vec{G}) < k$. 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: ReTest $(\vec{G}, k)$

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

Algorithm 2. 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 $O(|E|^{3/2} \log p(G))$.

Limitations of the existing algorithm. The efficiency of the existing algorithm is mainly constrained by the poor approximation performance of DEGREE. In our experiments, the maximum indegree of the orientations output by DEGREE often approaches twice the pseudoarboricity, which is the worst-case scenario guaranteed by its approximation ratio. Consequently, the use of DEGREE entails binary search and multiple invocations of network flow computations, which is time-consuming. Besides, since DEGREE utilizes $\lceil \max d_u(\vec{G})/2 \rceil$-core for core-pruning, the parameter $\lceil \max d_u(\vec{G})/2 \rceil$ often fails to accurately approximate p, resulting in a larger-than-desired core in the output. This makes the core-pruning method unable to fully leverage its utility.

3.2 An improved 2-approximation algorithm

To address the limitations 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 a 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 are outlined in Algorithm 3. The following example illustrates how iDEGREE works.

Example 2. *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.*, $v_1, \ldots, v_7$. 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.

Algorithm 3: iDEGREE (G)

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

Below, by Lemma 2, we show that the approximation quality of the iDEGREE algorithm is guaranteed to be no worse than DEGREE. Furthermore, by Lemma 3, we demonstrate that iDEGREE can also provide more effective pruning performance compared to DEGREE. In addition, we can also easily derive that both the time and space complexity of Algorithm 3 are $O(|E| + |V|)$, the same as DEGREE.

Lemma 2. *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 3-8 of Algorithm 1, Lines 4-11 of Algorithm 3), DEGREE and iDEGREE obtain the same $\vec{G}$, thus we focus on the second 'while' loop in iDEGREE (Lines 14-17 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. □

Lemma 3. $\rho^* \geq \max_{u \in V} d_u(\vec{G})/2$.

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

3.3 A novel approximate algorithm

We propose a novel approximate algorithm INDEGREE, which is outlined in Algorithm 4. The key idea of INDEGREE is to iteratively

Algorithm 4: INDEGREE (G)

Input: A simple graph $G = (V, E)$.
Output: An approximate orientation $\vec{G} = (V, \vec{E})$.

1 $\vec{G} \leftarrow \emptyset$;
2 Initialize an array $d_i = 0, i = 1, \ldots, |V|$;
3 **foreach** $(u, v) \in G$ **do**
4 **if** $d_u < d_v$ **then**
5 $\vec{G} \leftarrow \vec{G} \cup \langle v, u \rangle$; $d_u \leftarrow d_u + 1$;
6 **else**
7 $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$; $d_v \leftarrow d_v + 1$;
8 **repeat**
9 **foreach** $to \in V$ **do**
10 **foreach** $\langle from, to \rangle \in \vec{G}$ **do**
11 **if** $d_{to} \geq d_{from} + 2$ **then**
12 reverse $\langle from, to \rangle$; $d_{to} \leftarrow d_{to} - 1$; $d_{from} \leftarrow d_{from} + 1$;
13 **until** *Current iteration does not reduce the maximum indegree of* $\vec{G}$;
14 **return** $\vec{G}$;

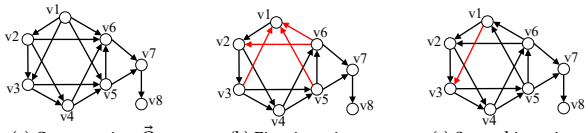
reverse the edges so that each edge is directed toward the endpoint with a smaller indegree. In particular, INDEGREE first constructs an orientation $\vec{G}$ where each edge initially points towards the endpoint with the smaller indegree (Lines 3-7). Then, the algorithm examines whether each edge points towards the vertex with smaller indegree, and reverses it if not (Lines 8-13). 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 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 8-13) will converge to a *stable orientation*.

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 13), 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 3. *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, Fig. (2b) and Fig. (2c) respectively illustrate the results of the first and second iterations. As the orientation in Fig. (2c) is already optimal, subsequent iterations will not decrease the maximum indegree, hence the iteration process terminates.*

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.

Theoretical analysis for general graphs.

Theorem 3. *Let $\vec{G}$ be a stable orientation 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 that is typically not significantly larger than $p(G)$. For instance, 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. Below, we provide another theoretical guarantee for INDEGREE, and this theorem is particularly effective when the graph G is dense (i.e., $\frac{|E|}{|V|(|V|-1)/2} \approx 1$).

Theorem 4. *Taking a graph $G = (V, E)$ as input, the maximum indegree $d_{\max}$ of the orientation outputted by* INDEGREE *satisfies $d_{\max} \leq \sqrt{2/c} \cdot p + \sqrt{c/2}$, where $c = \frac{|E|}{|V|(|V|-1)/2}$.*

Proof. According to the stop condition of iterations in the INDEGREE algorithm, the maximum indegree $d_{\max}$ of $\vec{G}$ does not decrease during the last iteration. Let u be a vertex with an

indegree of $d_{\max}$ in the output orientation. It can be deduced that the indegree of u remains unchanged in the last iteration. During the last iteration, when traversing all $\langle from, u\rangle$ edges, there are a total of $d_{\max}$ $from$ vertices, each with an indegree greater than or equal to $d_{\max}-1$. Adding the indegrees of u and these $from$ vertices results in a sum of at least $d_{\max}+d_{\max}(d_{\max}-1)=d_{\max}^2$. Since $d_{\max}^2 \le |E|$, we conclude that $d_{\max} \le \sqrt{|E|}$.

For G, we have $\sqrt{|E|} = \sqrt{c|V|(|V|-1)/2} \le |V|\sqrt{c/2}$ and $p \ge |E|/|V| = c(|V|-1)/2 \Rightarrow |V| \le 2p/c+1$. Therefore, we obtain $d_{max} \le \sqrt{|E|} \le |V|\sqrt{c/2} \le (2p/c+1)\sqrt{c/2} = \sqrt{2/c}\cdot p + \sqrt{c/2}$. □

Theorem 4 indicates that, for a dense graph with $c \approx 1$ as input, the INDEGREE algorithm can achieve an approximation factor of nearly $\sqrt{2}$, which is significantly better than the 2-approximation factors of the DEGREE and iDEGREE algorithms.

Theoretical analysis for k-plex. Here we provide theoretical guarantees for INDEGREE in a specific scenario involving a *k-plex* as the input graph. The k-plex is a classic cohesive graph model, and it is defined as an undirected graph in which the degree of each vertex is no less than $|V|-k$ [18]. Note that when the input is a complete graph, i.e., a 1-plex or $c=1$, Theorem 4 and Theorem 5 are equivalent. However, when the input is not a complete graph, these two theorems may not be equivalent.

Theorem 5. *Taking a k-plex graph G as input, the maximum indegree $d_{\max}$ of the orientation outputted by* INDEGREE *satisfies* $d_{\max} \le \sqrt{2}(p+k/2)$.

Proof. According to the proof of Theorem 4, $d_{\max} \le \sqrt{|E|}$ holds. For a k-plex, we have $p \ge |E|/|V| \ge (n-k)/2 \Rightarrow n \le 2p+k$. Therefore, we obtain

$$\begin{aligned} d_{\max} &\le \sqrt{|E|} \le \sqrt{|V|(|V|-1)/2} \\ &\le \sqrt{(2p+k)(2p+k-1)/2} \\ &\le \sqrt{(2p+k)(2p+k)/2} \\ &= \sqrt{2}(p+k/2) \end{aligned}$$

This concludes the proof. □

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 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.

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

Input: An orientation $\vec{G}=(V,\vec{E})$, the edge (u,v) to be inserted, and the pseudoarboricity p before insertion.
Output: The updated orientation $\vec{G}$ and pseudoarboricity.

```
Suppose d_v ≤ d_u, otherwise swap the input edge (u, v);
G⃗ ← G⃗ ∪ ⟨u, v⟩;
if ReTest (G⃗, p) =False then p++;
return (G⃗, p);
```

4 MAINTAINING $p(G)$ IN DYNAMIC GRAPHS

Real-graphs are typically frequently updated, such as in the social network, where each interaction between users represents an edge insertion. In the case of such dynamic graphs, a trivial method to update pseudoarboricity is to re-invoke static algorithms. However, this approach becomes excessively time-consuming when updates are frequent. Therefore, similar to maintenance algorithms for degeneracy [37] or arboricity [7], there is a need to design efficient dynamic algorithms tailored for dynamic scenarios. In this section, our goal is to devise such algorithms. We start by establishing a pseudoarboricity update theorem.

Theorem 6. ***(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 $p(\bar{G}) = \max_{G'\subseteq\bar{G}}\lceil |E'|/|V'|\rceil \ge \max_{G'\subseteq G}\lceil (|E'|-1)/|V'|\rceil \ge p(G)-1$. Therefore, the pseudoarboricity could decrease by at most one after deleting an edge. □

In the following, we first propose a basic algorithm and then we improve it to a novel and faster algorithm.

4.1 A basic maintenance algorithm

The basic algorithm for edge insertion. With Theorem 6, 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. Then, BasicINS tests whether $p(G) \le p$ using ReTest, where p is the pseudoarboricity before inserting (u,v). If not, the pseudoarboricity p increases by 1 according to Theorem 6, and p remains unchanged otherwise. It is easy to see that the worst-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 6, 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 $p(G) \le (p-1)$ 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

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

Input: An **unreversible orientation** $\vec{G} = (V, \vec{E})$, the edge (u, v) to be inserted, and the pseudoarboricity p before insertion.

Output: The updated **unreversible orientation** $\vec{G}$ and pseudoarboricity.

1 Suppose $d_v \leq d_u$, otherwise swap the input edge (u, v);
2 $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
3 **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**
4 reverse the path;
5 **if** $\max_{i \in \vec{G}} d_i > p$ **then** p++;
6 **return** $(\vec{G}, p)$;

ReTest once. Since ReTest $(\vec{G}, k)$ can test whether $p(G) \leq k$, the correctness of BasicINS and BasicDEL can be derived directly.

4.2 A novel and faster maintenance algorithm

The basic algorithms involve 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 novel pseudoarboricity maintenance algorithms based on a concept of *unreversible orientation*, which is a special kind of optimal orientation. 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, making it much faster than the basic algorithm. 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 remain 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* [44]. As shown in [44], 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.

Lemma 4. *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 $\vec{G} \cup \langle u, v \rangle$.*

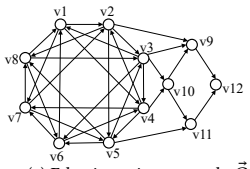
Proof. Let d_v and d_v^* denote the indegree of v before and after edge insertion, respectively. For the case of $d_v = p$, the edge insertion of $\langle u, v \rangle$ makes d_v^* equal $p + 1$. Thus, v becomes the only vertex with the maximum indegree; and therefore, any reversible path must end at 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 $d_v \leq p - 1$, 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 $d_s \leq p - 2$ 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 $d_v \leq p - 1$, 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 4, 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 3-4), 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 $\leq p - 2$ or the BFS-search queue is empty. INS can determine the pseudoarboricity by verifying $\max_{i \in \vec{G}} d_i > p$ directly (Line 5). 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 7.

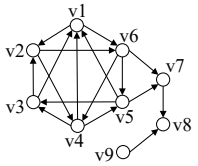
Theorem 7. *The* INS *algorithm correctly maintains the unreversible orientation and pseudoarboricity.*

Proof. If INS does not find any reversible path ending at v in $\vec{G}$, then $\vec{G}$ is an unreversible orientation by Lemma 4 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 $d_s \leq p - 2$ and $d_t = p$. If u is on the reversible path $w \rightsquigarrow v$, we have $d_u \geq p$ 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 $d_w \leq p - 2$. After reversing, since $s \rightsquigarrow t$ is a reversible path, we have $d_s \leq p - 2$ 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. □

The following example illustrates how Algorithm 6 works.



(a) Edge insertion example $\vec{G}$



(b) Edge deletion example $\vec{G}$

Figure 3: Example of dynamic graphs

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

Input: An **unreversible orientation** $\vec{G} = (V, \vec{E})$, the edge (u, v) to be deleted, and the pseudoarboricity p before deletion.

Output: The updated **unreversible orientation** $\vec{G}$ and pseudoarboricity.

1 $\vec{G} \leftarrow \vec{G} - \langle u, v\rangle$; // suppose (u, v) is oriented as $\langle u, v\rangle$

2 **if** $d_v \geq \max_{i\in\vec{G}} d_i - 2$ *and there is a reversible path starting from* v **then**

3 reverse the path;

4 **if** $\max_{i\in\vec{G}} d_i < p$ **then**

5 $p{-}{-}$; ReTest $(\vec{G}, p-1)$;

6 **return** $(\vec{G}, p)$;

Example 4. *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 $v_{10} \rightarrow v_9$ 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 2-3); 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 4). 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 5, DEL invokes the ReTest algorithm to reverse these paths, making the orientation unreversible again (Line 5).

Lemma 5. *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 $s \rightarrow y \rightsquigarrow x \rightarrow t$ 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. □

Theorem 8. DEL *can correctly maintain the unreversible orientation and pseudoarboricity.*

Proof. In Line 4 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 5, 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 2, 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 2-3). However, unlike INS, DEL may invoke ReTest (Line 5), 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

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

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

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

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

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 UPDATE 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 simpler and more efficient than the proposed INS algorithm (Algorithm 6).

5.1 A basic incremental insertion algorithm

Given an optimal orientation $\vec{G}$ and an edge (u, v) to be inserted, suppose $\langle u, v \rangle$ is directly inserted into $\vec{G}$. (i) If the maximum indegree of $\vec{G}$ remains p, *i.e.*, $d_v \leq p$ after the insertion, according to Theorem 6, the pseudoarboricity remains unchanged; (ii) If d_v becomes $p + 1$ and no reversible path is found, then $\vec{G}$ becomes an unreversible orientation and the pseudoarboricity increases by one; (iii) If d_v becomes $p + 1$ and a reversible path is found, by reversing this path, the maximum indegree remains p and the pseudoarboricity remains unchanged. Based on the aforementioned rationale, the algorithm INC can be designed, as shown in Algorithm 8, and its correctness can be derived from Theorem 6.

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 needs to consider the case when $d_v = p + 1$ (Line 3 of Algorithm 8), while INS has to consider the cases when $d_v = p$ or $d_v = p + 1$ (Line 3 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 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}. 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 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* $p(G)\}$.

Note that, by the definition of unreversible orientation, the vertices in D_{top} must have indegree no less than $p(G) - 1$. 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 maintenance of D_{top}, INS++ is not compatible with DEL in the fully-dynamic case. Besides, it is not straightforward and efficient to maintain D_{top} in the deletion case. So when maintaining D_{top}, we only devise an insertion-only algorithm INS++.

INS++ can also be faster than INC benefiting from the lower costs for finding reversible paths 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 9. 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 5 and Line 6. (i) $v \notin D_{top}$ and $d_v = p$. In this case, regardless of whether INS++ can discover a reversible path in Line 4, 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 5); (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 6-7); (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 that 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 can also efficiently and incrementally maintain it.

6 EXPERIMENTS

6.1 Experimental setup

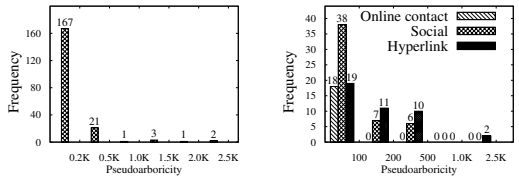
In our experiments, we implement four approximate algorithms: (i) DEGREE (the state-of-the-art approximation algorithm, *i.e.*, Algorithm 1); (i) iDEGREE (Algorithm 3); (iii) INDEGREE (Algorithm 4), which calculate the maximum indegree of orientation as an estimation of pseudoarboricity; (iv) DinicAppr [33], which is a $(1+\epsilon)$-approximate algorithm utilizing the early-stopped Dinic algorithm, can be employed to approximate arboricity and pseudoarboricity. To exactly compute the pseudoarboricity, we implement the state-of-the-art (SOTA) 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 SOTA algorithm for computing the densest subgraph [20]. For Convex, we use their original C++ implementation [20] for comparison. For fully-dynamic algorithms, we implement the basic maintenance algorithms BasicINS (Algorithm 5) and BasicDEL, as well as 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).

We collect 195 real-life graphs with various types downloaded from the Network Repository [43] 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 real-world graphs

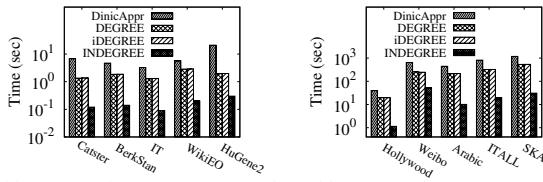
In this experiment, we systematically evaluated the pseudoarboricity of 195 real-life graphs with various types, and the results are reported in Table 1. As can be seen, the pseudoarboricities of citation graphs, online contact graphs, infrastructure graphs, technological graphs, software graphs, and lexical graphs are often very small. However, a few 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.

Fig. (4) depicts the distributions of pseudoarboricity for different types of graphs. Clearly, 167 out of the 195 real-world graphs (85%)



(a) All graphs. (b) Three types of graphs.

Figure 4: Pseudoarboricity distribution of real-world graphs.



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

Figure 5: Runtime of different approximation algorithms.

have a pseudoarboricity of less than 200 in Fig. (4a), validating that most real-world graphs have small pseudoarboricities. Fig. (4b) shows that (i) the pseudoarboricities of all online contact graphs are less than 100; (ii) over 88 percent of social graphs have small pseudoarboricities of less than 200; (iii) most hyperlink graphs have small pseudoarboricity, while a few 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, 38, 49] may be excessively optimistic for large hyperlink graphs.

6.3 Performance studies

Comparison of different approximation algorithms. Table 2 shows the maximum indegree of the approximate orientation calculated by DEGREE, iDEGREE, INDEGREE, and DinicAppr respectively. As 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 outstanding 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, which confirms the theoretical analysis in Section 3.3. For the $(1+\epsilon)$-approximation algorithm DinicAppr, its rationale is employing an early-stopped Dinic algorithm to approximate the complete Dinic algorithm. However, in our experiments, the algorithm appears to struggle to meet the condition for early stopping, even when we set ϵ to a considerably high value, such as 10. This renders DinicAppr practically equivalent to an exact algorithm. Although it computes an approximate pseudoarboricity that is typically equal to the exact value, its runtime is extremely long, as elaborated later.

The runtime of DEGREE, iDEGREE, INDEGREE, and DinicAppr is shown in Fig. (5), where, to ensure a fair comparison, we add core reduction to DinicAppr for optimizing its runtime and

Table 1: Networks statistics and the pseudoarboricity results. (1K=1,000,1M=1,000,000, and 1G=1,000,000,000)

'ρ' represents pseudoarboricity, and 'Ratio' is the acceleration ratio of the INDEGREE+ReTest algorithm compared to the DEGREE+ReTest algorithm.

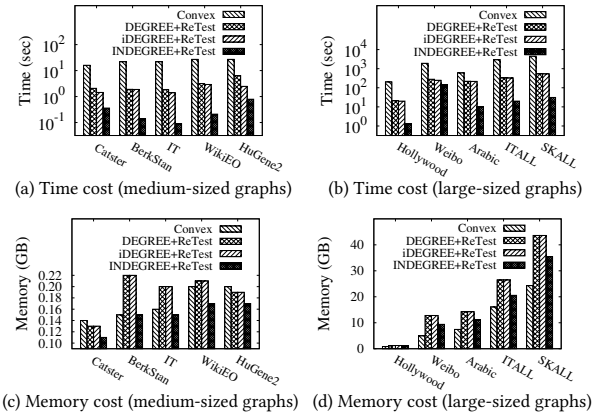
Networks	Name	$\|V\|$	$\|E\|$	ρ	Ratio	Name	$\|V\|$	$\|E\|$	ρ	Ratio	Name	$\|V\|$	$\|E\|$	ρ	Ratio
Biology	Diseas	0.5K	1.2K	6	6.1	Worm	3.5K	6.5K	8	7.7	CE-GN	2.2K	53.7K	40	2.4
	Mangwet	0.1K	1.4K	17	3.3	FisYeast	2.0K	12.6K	27	5.6	DR-CX	3.3K	84.9K	66	2.6
	Yeast	1.5K	1.9K	3	5.1	Fruitfly	7.3K	24.9K	10	2.9	HS-CX	4.4K	108.8K	65	2.0
	Celegans	0.5K	2.0K	8	4.1	Human	9.4K	31.2K	10	2.7	GYeast	6.0K	156.9K	56	2.0
	Foodweb	0.1K	2.1K	18	5.4	SC-GT	1.7K	34.0K	47	3.0	CE-CX	15.2K	246.0K	55	5.6
	Florida	0.1K	2.1K	18	3.0	SC-CC	2.2K	34.9K	36	7.3	**HuGene2**	**14.0K**	**9.0M**	**1,326**	**8.1**
	Plant	1.7K	3.1K	9	12.0	HS-LC	4.2K	39.5K	44	8.0	HuGene1	21.9K	12.3M	1,452	8.5
	DM-HT	3.0K	4.7K	6	11.4	CE-PG	1.9K	47.8K	62	5.7	MoGene	43.1K	14.5M	809	6.2
Collaboration	Netscience	1.5K	2.7K	10	7.7	HepPh	11.2K	117.6K	119	5.0	caCoCite	22.9K	2.4M	317	11.2
	caErdos	6.9K	11.9K	8	7.1	AstroPh	17.9K	197.0K	33	2.1	caIMDB	896.3K	3.8M	21	3.0
	caGrQc	4.2K	13.4K	23	11.6	caCtseer	227.3K	814.1K	43	33.0	caDBLP	540.5K	15.2M	168	27.9
	CondMat	21.4K	91.3K	14	2.2	caMath	391.5K	873.8K	12	7.2	**Hollywood**	**1.1M**	**56.3M**	**1,104**	**16.2**
Citation	ctDBLP	12.6K	49.6K	11	1.7	ctCtseer	384.1K	1.7M	14	1.8	ctHepPh	28.1K	3.1M	266	2.3
	ctCora	23.2K	89.2K	10	1.5	ctHepTh	22.9K	2.4M	317	1.7	ctPatent	3.8M	16.5M	41	34.0
Online contact	emUniv	1.1K	5.5K	8	2.2	WkTkLV	41.4K	50.6K	13	14.2	comMath	24.8K	188.0K	65	4.6
	emDNC	0.9K	10.4K	41	3.7	emEU	32.4K	54.4K	19	5.0	comEnron	87.0K	297.5K	44	3.8
	comCore	0.9K	10.4K	13	3.7	WkTkEL	40.3K	72.1K	22	11.8	emEuAll	265.0K	364.5K	33	12.2
	comUc	1.9K	13.8K	17	2.7	DIGG	30.4K	85.2K	9	2.3	comAsk	157.2K	455.7K	42	13.2
	WkTkEO	7.6K	15.6K	18	10.3	EnLarge	33.7K	180.8K	38	4.1	Super	192.4K	714.6K	54	11.9
	comPGP	10.7K	24.3K	20	13.0	FBwal	45.8K	183.4K	14	1.8	comWiki	138.6K	715.9K	51	10.7
Infrastructure	Euro	1.2K	1.4K	2	2.5	US1	129.2K	165.4K	2	1.7	Italy	6.7M	7.0M	2	1.1
	USAir97	0.3K	2.1K	18	5.5	PA	1.1M	1.5M	2	1.9	Britain	7.7M	8.2M	2	1.4
	Power	4.9K	6.6K	4	13.0	BE	1.4M	1.5M	2	1.1	DE	11.5M	12.4M	2	1.2
	Openflights	2.9K	15.7K	23	3.8	NL	2.2M	2.4M	2	1.2	Asia	12.0M	12.7M	2	1.0
	LU	114.6K	119.7K	2	2.8	CA	2.0M	2.8M	2	1.9	US2	23.9M	28.9M	2	2.0
Social	FbNIPS	2.9K	3.0K	3	6.8	WikiElec	7.1K	100.8K	47	1.7	LiveMch	104.1K	2.2M	84	3.0
	HHouse	1.6K	4.0K	11	4.7	GemRO	41.8K	125.8K	6	1.4	Buzznet	101.2K	2.8M	138	2.2
	HFriship	1.9K	12.5K	17	3.3	fbMedia	27.9K	206.0K	19	1.4	YTBsnap	1.1M	3.0M	46	6.1
	HFriend	3.0K	12.5K	17	3.7	Britkite	58.2K	214.1K	41	9.9	FourSq	639.0K	3.2M	56	8.7
	Hamster	2.4K	16.6K	18	3.0	GemHU	47.5K	222.9K	10	2.2	FlickrU	514.0K	3.2M	254	10.5
	Tvshow	3.9K	17.2K	31	7.9	Douban	154.9K	327.2K	14	5.5	Lastfm	1.2M	4.5M	62	8.9
	TwitList	23.4K	32.8K	8	7.7	Sladot1	77.4K	469.2K	43	5.2	wkTk	2.4M	4.7M	115	10.1
	Ciaodvd	4.7K	33.1K	36	6.6	GemHR	54.6K	498.2K	17	0.9	**Catster**	**149.7K**	**5.4M**	**348**	**5.8**
	Gplus	23.6K	39.2K	11	6.5	Sladot2	82.2K	504.2K	44	5.6	DIGG	770.8K	5.9M	215	5.4
	Advogt	5.2K	39.4K	23	2.7	Acade	190.2K	788.3K	17	1.1	Flixster	2.5M	7.9M	51	8.2
	fbPoli	5.9K	41.7K	25	6.1	fbArtist	50.5K	819.1K	59	1.2	Dogster	426.8K	8.5M	218	8.4
	Anybeat	12.6K	49.1K	29	4.3	TwiFol	465.0K	833.5K	26	6.7	Higgs	456.6K	12.5M	115	1.9
	fbCom	14.1K	52.1K	14	8.5	Delicious	426.4K	908.3K	18	10.2	Flickr	1.7M	15.6M	469	11.1
	PubFig	11.6K	67.0K	35	3.3	Gowalla	196.6K	950.3K	44	3.6	Pokec	1.6M	22.3M	42	1.5
	fbSport	13.9K	86.8K	20	6.3	Themker	69.4K	1.6M	144	3.8	Lvjourn	4.0M	27.9M	131	32.0
	Govern	7.1K	89.4K	37	1.8	YTB	496.0K	1.9M	44	3.0	Orkut	3.0M	106.3M	207	5.3
	Epinions	26.6K	100.1K	27	2.0	BlogCata	88.8K	2.1M	194	4.8	**Weibo**	**58.7M**	**261.3M**	**166**	**1.9**
Hyperlink	Polblogs	0.6K	2.3K	10	4.0	WikiIS	69.4K	907.4K	197	11.1	Wiki	1.9M	4.5M	55	48.6
	EPA	4.3K	8.9K	5	3.3	WikiFY	65.6K	921.6K	84	7.1	WikiTH	266.9K	4.6M	215	18.9
	Webase	16.1K	25.6K	16	10.8	Notre	325.7K	1.1M	79	18.6	WikiLT	268.2K	5.1M	158	13.8
	WkCham	2.3K	31.4K	48	14.3	WikiIA	24.0K	1.2M	280	10.9	**BerkStan**	**685.2K**	**6.6M**	**104**	**13.6**
	Spam	4.8K	37.4K	28	3.7	WikiAF	72.3K	1.5M	187	10.1	**IT**	**509.3K**	**7.2M**	**216**	**20.8**
	Idchina	11.4K	47.6K	25	8.9	IkArabic	163.6K	1.7M	51	12.3	**WikiEO**	**413.0K**	**8.2M**	**354**	**15.0**
	WikiCO	8.3K	119.8K	121	1.0	WikiAST	83.3K	2.0M	75	6.1	WikiCh	1.9M	9.0M	95	5.0
	GogInter	15.8K	149.5K	55	13.0	Stanford	281.9K	2.0M	60	11.2	UK	129.6K	11.7M	250	15.1
	WikiCroc	11.6K	170.9K	51	13.2	BaiduRe	415.6K	2.4M	183	27.6	Hudong	2.0M	14.4M	157	49.0
	WikiSqui	5.2K	198.5K	137	11.2	Italycnr	325.6K	2.7M	58	17.7	Baidu	2.1M	17.0M	73	2.2
	SK	121.4K	334.4K	42	15.1	WikiNN	215.9K	2.9M	133	19.4	WikiUK	1.2M	41.9M	459	15.1
	WikiYO	41.2K	696.4K	242	6.4	WikiLV	190.0K	2.9M	283	16.5	UKAll	39.5M	783.0M	486	7.4
	WikiCKB	60.7K	802.1K	204	12.7	WikiLA	181.2K	3.0M	140	14.0	**ITALL**	**41.3M**	**1.0G**	**2,009**	**16.4**
	WikiSW	58.8K	877.0K	157	12.4	Google	875.7K	4.3M	29	23.9	**SKALL**	**50.6M**	**1.8G**	**2,258**	**17.3**
Technological	Routers	2.1K	6.6K	12	5.4	WHOIS	7.5K	56.9K	63	4.1	RLCaida	190.9K	607.6K	26	13.4
	PGP	10.7K	24.3K	20	13.8	Internet	40.2K	85.1K	18	6.0	Skitter	1.7M	11.1M	90	25.9
	Caida	26.5K	53.4K	18	10.5	P2P	62.6K	147.9K	5	2.4	IP	2.3M	21.6M	200	3.7
Software	Jung	6.1K	50.3K	47	12.2	JDK	6.4K	53.7K	47	11.4	Linux	30.8K	213.2K	21	3.5
Lexical	EAT	23.1K	297.1K	31	3.6	Wordnet	146.0K	657.0K	17	1.8	Yahoo	653.3K	2.9M	24	7.5
Miscellaneous	HyperText	0.1K	2.2K	21	5.2	Beaflw	0.5K	45.3K	97	17.8	Amazon2	403.4K	2.4M	10	4.5
	Infectious	0.4K	2.8K	11	1.9	Orani	2.5K	87.0K	122	10.2	DBpedia	4.0M	12.6M	16	7.3
	Twin	14.3K	20.6K	13	19.2	Amazon1	334.9K	925.9K	5	3.1	Actor	382.2K	15.0M	310	9.5
	Beacxc	0.4K	42.6K	93	15.2	misFlickr	105.9K	2.3M	292	7.0	**Arabic**	**22.7M**	**553.9M**	**1,625**	**21.4**

Table 2: Approximation performance of different algorithms.

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

set $\epsilon = 1$. We can clearly see that INDEGREE 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. In comparison to INDEGREE, the runtime of DinicAppr is 12.2-69.8 times slower, and even slower by 4.4-43.4 times than the exact algorithm INDEGREE+ReTest. Therefore, while DinicAppr exhibits good approximation performance, its execution speed is significantly slow, making it replaceable by our proposed exact algorithm INDEGREE+ReTest. These results demonstrate that the proposed INDEGREE algorithm substantially outperforms the other approximation algorithms in terms of rapid running speed and outstanding approximation quality.

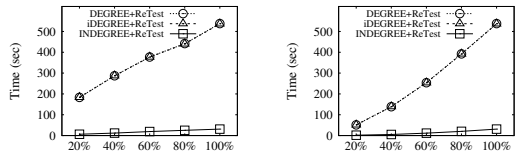
Comparison of exact algorithms. Fig. (6a) and Fig. (6b) depict the runtime of Convex, DEGREE+ReTest, iDEGREE+ReTest and INDEGREE+ReTest on 10 real-life datasets. We can see that our novel solution INDEGREE+ReTest consistently outperforms other algorithms over all datasets. The Convex algorithm has the 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



(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 6: Results of exact algorithms for pseudoarboricity computation.



(a) vary $|V|$ (b) vary $|E|$

Figure 7: Scalability of different algorithms on SKALL.

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 SKALL dataset, the runtime of Convex, DEGREE+ReTest, iDEGREE+ReTest, and INDEGREE+ReTest are 4,528, 537.84, 536.49, and 31.07 seconds respectively. To further compare our proposed INDEGREE+ReTest algorithm with the SOTA algorithm DEGREE+ReTest, we calculate the speedup ratio of INDEGREE+ReTest over DEGREE+ReTest on all datasets in Table 1. It can be observed that INDEGREE+ReTest is faster on almost all datasets, and on 31 out of 42 hyperlink graphs, it achieves a speedup ratio exceeding one order of magnitude. These results demonstrate the efficiency of the INDEGREE+ReTest algorithm on the exact computation of pseudoarboricity.

Fig. (6c) and Fig. (6d) 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. These 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. (7). We can observe 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 demonstrate the high scalability of our INDEGREE+ReTest algorithm.

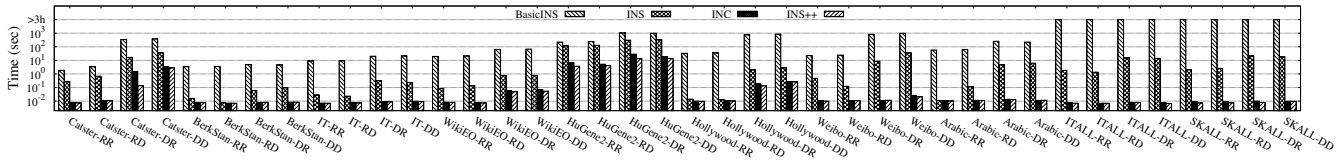
Table 3: Insertion time on temporal graphs.

Total time for the insertion of a whole graph. (Unit: second)

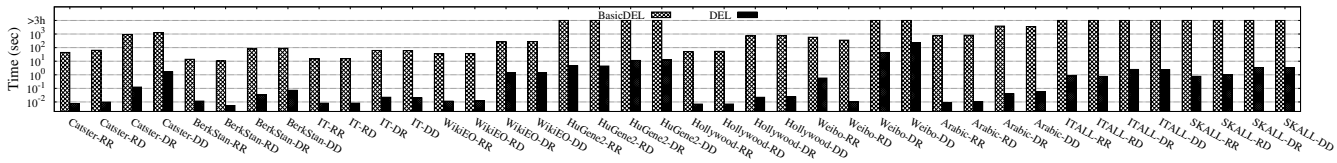
Dataset	$\|V\|$	$\|E\|$	BasicINS	INS	INC	INS++
WikiElec	8,298	100,753	188.7	10.3	8.7	2.2
Epinions	131,829	711,210	>10,000	1831.5	1156.9	655.9
HepTh	22,909	1,222,399	>10,000	2654.6	2,253.3	1,071.7

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. When selecting edges, we adopt a two-step process. First, we randomly choose a node (with equal weight or weighted by degree). Second, we randomly select a node from the neighbors of the chosen node (with equal weight or weighted by degree). The edge formed by these two nodes is then considered the selected edge. We employ four different edge selection strategies: Random-Random (RR), Random-Degree (RD), Degree-Random (DR), Degree-Degree (DD). For instance, RD signifies that we first randomly select a node with equal weight, and then, from its neighbors, select a node weighted by degree. The results are shown in Fig. (8) (results for all datasets are in full version paper). For the fully-dynamic case, the improved maintenance algorithms (INS and DEL) consistently outperform the basic ones (BasicINS and BasicDEL). For instance, on the large graph Hollywood with edge selection strategy of DR, INS and DEL consumes only 2.025 seconds and 0.022 seconds, respectively, which are 2-3 orders of magnitude faster than basic algorithms. For the incremental insertion case, the proposed algorithms INC and INS++ are incredibly fast on most datasets. Both of them take less than 0.01 seconds on datasets BerkStan, ITALL, and SKALL. For Catster and HuGene2 with edge selection strategy of DR, the runtime of INS++ is significantly lower than that of INC. Specifically, INS++ takes only 0.183 seconds and 13.366 seconds, whereas INC takes 1.525 seconds and 28.689 seconds. These results demonstrate the high efficiency of the proposed algorithms in handling dynamic graphs.

Results on temporal graphs. In this experiment, to evaluate the performance of our proposed insertion algorithms, we employ three temporal graphs: the graphs WikiElec and Epinions are social network graphs, and HepTh is a collaboration network among researchers. These datasets are sourced from the Network Repository [43], and they include timestamps. We start with an empty graph and insert edges one by one in the order of timestamps until all edges are inserted. The results are shown in Table 3. As shown, algorithms INS, INC, and INS++ exhibit significantly improved runtime compared to the basic algorithm BasicINS. Moreover, INS++ outperforms others across all datasets, with an average insertion time of only 2.29 milliseconds on the HepTh dataset. The results demonstrate the high efficiency of our proposed algorithms in real temporal graphs. Furthermore, we present the evolution of pseudoarboricity during edge insertion, as depicted in Fig. (9). For the WikiElec dataset, we observe a steady and gradual increase in pseudoarboricity over time. Conversely, for the Epinions and HepTh datasets, pseudoarboricity exhibits fluctuations, alternating

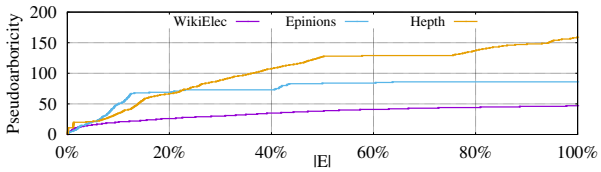


(a) Insertion time on datasets with four different edge selection strategies.



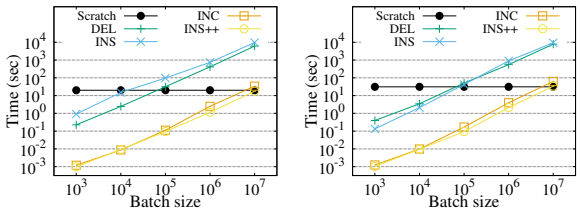
(b) Deletion time on datasets with four different edge selection strategies.

Figure 8: Results of dynamic algorithms for pseudoarboricity maintenance (total time for 10,000 edge updates).



Edges are inserted in the order of their time stamps.

Figure 9: Pseudoarboricity evolution in temporal graphs.



(a) ITALL (b) SKALL

'Scratch' represents computing from scratch.
'Batch size' represents the number of edges in a batch.

Figure 10: Results of batch update on ITALL and SKALL.

between periods of stagnation and sudden spikes. These findings demonstrate the significance of pseudoarboricity in capturing the density dynamics of temporal graphs, offering valuable insights for comprehensive temporal graph analysis.

Results of batch update. In these experiments, we evaluate the performance of our proposed algorithms in handling batch updates, *i.e.*, the insertion or deletion of multiple edges. Our baseline method is re-invoking the static algorithm INDEGREE+ReTest to recompute from scratch. The algorithms for comparison are the dynamic algorithms INS, INC, INS++, and DEL. We execute the dynamic algorithms by treating a batch update as individual edge updates. The experiments are conducted on our two largest datasets, ITALL and SKALL, and the results are shown in Fig. (10). For full-dynamic algorithms INS and DEL, their runtime is comparable to computing from scratch when the batch size is between 10^4 and 10^5. For incremental dynamic algorithms INC and INS++, it is only when the batch size reaches 10^7 that their runtime becomes comparable to computing from scratch. These results indicate that INS and DEL can efficiently handle large batches with a size less than 10^5, while INC and INS++ are highly efficient and capable of handling batches as large as 10^7.

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

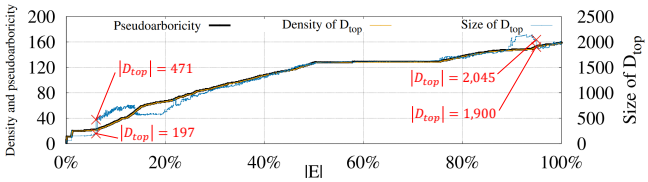
Dataset	$\rho(\delta$-core)	$\rho(D_{top})$	$\rho(G)$	Dataset	$\rho(\delta$-core)	$\rho(D_{top})$	$\rho(G)$
Catster	310.9	347.8	347.8	Hollywood	1,104.0	1,104.0	1,104.0
BerkStan	103.4	103.4	103.4	Weibo	107.6	165.4	165.4
IT	215.5	215.5	215.5	Arabic	1,623.5	1,624.5	1,624.5
WikiEO	344.5	353.2	353.2	ITALL	1,619.5	2,008.2	2,008.2
HuGene2	1,130.9	1,325.2	1,325.2	SKALL	2,256.5	2,257.5	2,257.5

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, accurate to the tenths place, of δ-core (δ is the degeneracy), the densest subgraph, and D_{top} on 10 datasets. Clearly, on all datasets, the subgraph induced by D_{top} achieves the same density as the densest subgraph when accurate to the tenth place. Conversely, the density of δ-core is significantly lower than $\rho(G)$, especially on HuGene2, Weibo, and ITALL. These results indicate that D_{top} is indeed very effective in detecting dense subgraphs in real-world graphs.

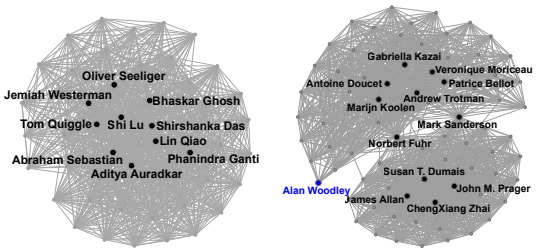
We also conduct a case study on two subgraphs extracted from the collaboration graph DBLP (https://dblp.org/), namely, DB and IR. DB consists of authors who published at least one paper in the database and data mining related conferences with 37,177 vertices and 131,715 edges. IR includes authors who 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. (12) (For clarity, only the names of authors represented by high-degree nodes are displayed). 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, 42]. 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 [2], while another group of authors has co-published a workshop report [3]. Norbert Fuhr and Mark Sanderson belong to both groups, thus connecting them. These results further confirm the effectiveness of D_{top} in identifying densely-connected communities.

D_{top} evolution in temporal graphs. For the temporal graph HepTh, we depict the evolution of its D_{top}, including the size of



Edges are inserted in the order of their time stamps.

Figure 11: D_{top} evolution in temporal graphs.



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

Figure 12: D_{top} of DB and IR.

D_{top}, $|D_{top}|$, and the density of D_{top}, $\rho(D_{top})$, as illustrated in Fig. (11). As expected, the evolution of $\rho(D_{top})$ closely resembles that of pseudoarboricity because $\rho(D_{top})$ is no less than pseudoarboricity minus 1. Additionally, the size of D_{top} changes continuously since D_{top} represents a dense community in HepTh, and dense communities are typically evolving. For example, in Fig. (11), we annotate a sudden increase in $|D_{top}|$ from 197 to 471, which results from an insertion that densifies a previously sparse region, merging it into the dense community. Another example is the insertion causing $|D_{top}|$ to drop from 2,045 to 1,900, as it densifies a part in the D_{top}, causing a relatively sparse area to detach from D_{top}. These experimental results indicate that in dynamic graphs, maintaining D_{top} can provide insights into the evolution of dense communities in dynamic graphs.

7 RELATED WORK

Arboricity computation. In [41], Picard and Queyranne showed that arboricity can be computed by Edmonds' matroid partitioning algorithm [21], and propose an improved algorithm with the time complexity of $O(|V|^2|E|\log^2|V|)$. Gabow and Westermann devised an algorithm based on the matroid partition technique with $O(a^{3/2}|V|\sqrt{|E|}) \geq O(|E|^{3/2}\sqrt{|E|/|V|})$ time complexity [25]. Subsequently, Gabow improved the time complexity to $O(|E|^{3/2}\log(|V|^2/|E|))$ using graphic polymatroid, which is the SOTA algorithm 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. The minimizing the maximum indegree problem aims to find an optimal orientation with the smallest maximum indegree, which 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 Bezáková [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 [5, 22, 33]. With these approximate algorithms, Blumenstock presented an improved algorithm based on Bezáková'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 [50], and fraud detection [27]. To compute the densest subgraph, several network flow and convex programming algorithms are devised [17, 20, 26]. However, these algorithms are often time-consuming for very 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. We believe that the proposed structure of D_{top} can be of independent interest for graph analysis applications.

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 to approximate the pseudoarboricity. With our approximation algorithms, we can significantly reduce the search space of the exact pseudoarboricity computation algorithms. For dynamic graphs, 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 insertion-only scenarios. 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 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] Niranka Banerjee, Venkatesh Raman, and Saket Saurabh. 2020. Fully dynamic arboricity maintenance. *Theor. Comput. Sci.* 822 (2020), 1–14.

[8] 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.

[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 Outdegree 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] Qiangqiang Dai, Rong-Hua Li, Hongchao Qin, Meihao Liao, and Guoren Wang. 2022. Scaling Up Maximal k-plex Enumeration. In *CIKM*. 345–354.

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

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

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

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

[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] Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao. 2022. I/O-Efficient Algorithms for Degeneracy Computation on Massive Networks. *IEEE Trans. Knowl. Data Eng.* 34, 7 (2022), 3335–3348.

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

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

[40] Mark Ortmann and Ulrik Brandes. 2014. Triangle Listing Algorithms: Back from the Diversion. In *ALENEX*. 1–8.

[41] 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.

[42] 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.

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

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

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

[46] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient Structural Graph Clustering: An Index-Based Approach. *Proc. VLDB Endow.* 11, 3 (2017), 243–255.

[47] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient structural graph clustering: an index-based approach. *VLDB J.* 28, 3 (2019), 377–399.

[48] 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.

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

[50] 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.