Efficient Algorithms for Density Decomposition on 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

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

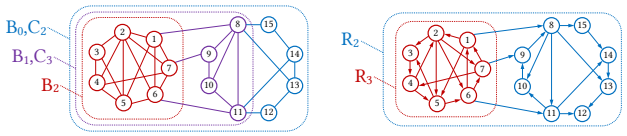
ABSTRACT

Locally-densest subgraph (LDS) decomposition is a fundamental operation in graph analysis that finds numerous applications in various domains, including community detection, fraud detection, graph querying, and graph visualization. However, the LDS decomposition is computationally challenging for both static and dynamic graphs. Furthermore, the LDS decomposition often produces an excessive number of dense subgraph layers, leading to the unnecessary separation of tightly-connected subgraphs. To address these limitations, an alternative concept called density decomposition was proposed, which can generate a more reasonable number of dense subgraph layers. However, the state-of-the-art algorithm for density decomposition requires $O(m^2)$ time (m is the number of edges of the graph), which is very costly for large graphs. In this paper, we conduct an in-depth investigation of density decomposition and propose efficient algorithms for computing it on both static and dynamic graphs. First, we establish a novel relationship between density decomposition and LDS decomposition, allowing us to leverage existing LDS algorithms for density decomposition. Second, we propose a novel algorithm to compute the density decomposition on static graphs, called Flow++, based on carefully-designed network flow and divide-and-conquer techniques. The striking feature of our Flow++ algorithm is that its time complexity is $O(m^{3/2} \log p)$ (p is often a very small constant in real-world graphs), thus significantly reducing the complexity of the previous algorithm. Third, for dynamic graphs, we prove a density decomposition updating theorem, based on which we develop three dynamic algorithms with $O(m)$ time complexity to maintain the density decomposition. Extensive experiments on several large real-world graphs demonstrate the high efficiency, scalability, and effectiveness of the proposed algorithms.

1 INTRODUCTION

Real-world graphs are typically overall sparse but contain small dense subgraphs. Discovering these dense subgraphs has a wide range of applications, including community detection [1, 19, 31, 39, 51, 67], fraud detection [6], graph querying [21, 38], and graph visualization [3, 68]. Various models have been introduced to characterize dense subgraphs, such as the clique [27], k-core [4, 41, 45, 52], k-truss [22, 37], locally densest subgraph [42, 49, 61], k-edge connected subgraph [15, 16], k-plex [24, 57], k-defective clique [23], k-club [46], and so on.

Among them, k-core perhaps stands as the most popular dense graph model due to its concise definition and linear time complexity. However, the k-core decomposition, defined solely based on degrees, falls short of effectively capturing the density structure of a graph, where density is measured as the ratio of the number of edges to the number of vertices. For example, as shown in Fig. 1a, the top layer of core decomposition C_3 contains 23 edges and 11 vertices, yielding a density of 23/11. Nevertheless, the subgraph with the highest density in this graph is B_2 with a density of 15/7. Clearly, the core decomposition is not density-based, and thus may not accurately capture the dense subgraphs.



(a) LDS decomposition and core decomposition

(b) density decomposition

Figure 1: Example graph $G = (V, E)$ and its decompositions.

To address this limitation, Tatti [61] introduced a concept of *locally-densest subgraph (LDS) decomposition*, dividing the graph into a set of LDSes with hierarchical features, i.e., a smaller subgraph must be contained in a larger subgraph, and the density of subgraphs strictly decreases as the size of the subgraph increases. LDS decomposition ensures that the smallest subgraph is the densest subgraph in the whole graph. However, despite its ability to fully reflect the density structure, LDS decomposition has three notable drawbacks: (1) The worst-case time complexity for computing it is as high as $O(n^2m)$, where n and m denote the number of vertices and edges, which is costly for large graphs; (2) The number of decomposed LDSes may be excessive and can only be bounded by $O(n)$, typically leading to unnecessary separation of densely-connected structures. For example, in Fig. 1a, B_0, B_1, and B_2 are LDSes. They separate the tightly connected subgraph $\{v_8, \ldots, v_{15}\}$, which is not necessary; (3) It is very challenging to exactly maintain the LDSes on dynamic graphs due to the strictness of its definition. To the best of our knowledge, there is currently no efficient algorithm for maintaining LDS decompositions, except for recomputing the decomposition from scratch.

On the other hand, Borradaile et al. [14] introduced an alternative density-based decomposition, known as *density decomposition*, which partitions the vertices based on their indegree and connectivity in a so-called egalitarian orientation [13]. Similar to the LDS decomposition, the density decomposition also exhibits a hierarchical structure, with the density of subgraphs decreasing as the size of layers increases. Consequently, the smallest layer of this decomposition also has extremely high density. In [14], Borradaile et al. proposed an $O(m^2)$ time algorithm to compute the density decomposition, which is clearly costly for large graphs. Moreover, Borradaile et al. did not establish a clear relationship between the density decomposition and other dense subgraph models.

Motivated by this, we for the first time conduct a thorough investigation of density decomposition, providing insights and a comprehensive understanding of its properties and applications. Specifically, we first establish a novel and interesting connection between density decomposition and LDS decomposition, which enables us to use the existing highly-optimized LDS decomposition algorithm to compute the density decomposition [42]. Then, we propose a new algorithm based on carefully-designed network flow and divide-and-conquer techniques to efficiently compute the density decomposition with reduced time complexity. Subsequently, we also develop novel algorithms with linear time complexity to efficiently maintain the density decomposition on dynamic graphs. Finally, we conduct extensive experiments to evaluate our algorithms using several large real-life graphs, and the results demonstrate the efficiency, scalability, and effectiveness of the proposed algorithms. In summary, the main contributions of this paper are as follows.

New theoretical results on density decomposition. We show that each layer of the density decomposition is exactly an LDS, making it part of the LDS decomposition. This crucial relationship reveals the fundamental similarity between these two decompositions. On the other hand, their key distinction lies in that density decomposition, through a round-up relationship, can naturally group LDSes with similar densities. This grouping significantly reduces the number of subgraphs, addressing the issue of unnecessary layers in the LDS decomposition. In particular, we show that the number of non-trivial layers generated by the density decomposition is equal to the *pseudoarboricity* of a graph, denoted as p, typically a small number. In addition, we also reveal an interesting approximation relationship between density decomposition and k-core decomposition, indicating that core decomposition can be considered as a 2-approximation of the density decomposition.

Novel algorithms for computing density decomposition. We first propose a novel algorithm, called Flow, to compute the density decomposition, based on a carefully-designed network flow technique. Compared to the state-of-the-art $O(m^2)$-time algorithm, Flow achieves a superior time complexity of $O(p \cdot m^{3/2})$. To further improve the efficiency, we propose a powerful divide-and-conquer technique, and develop a new network flow-based algorithm, called Flow++. We show that such a new algorithm further reduces the time complexity to $O(m^{3/2} \log p)$.

Novel algorithms for maintaining density decomposition. We for the first time study the problem of maintaining the density decomposition on dynamic graphs. We discover a density decomposition update theorem, revealing that the insertion or deletion of an edge requires analyzing changes in just one layer of the density decomposition. Based on this, we develop an insertion algorithm, called Insert to handle edge insertions. This algorithm only needs to invoke the Breadth-First Search (BFS) algorithm at most twice, yielding a worst-case time complexity of $O(m)$. For edge deletion, we develop the Delete algorithm along with its enhanced version, Delete++, which incorporates several carefully-designed pruning strategies. Both of these algorithms have $O(m)$ time complexity, thus they are very efficient for handling large dynamic graphs.

Extensive experiments. We conduct comprehensive experiments to evaluate the proposed algorithms using 8 real-world datasets. The results are summarized as follows: (1) Both the Flow and Flow++ algorithms are substantially faster than the state-of-the-art algorithm by at least two orders of magnitude for density composition. Moreover, even when compared to the highly-optimized LDS decomposition algorithm [42], our Flow++ can still achieve a remarkable speed improvement of one order of magnitude; (2) For density decomposition maintenance, both our insertion and deletion algorithms outperform the baselines by at least 5 orders of magnitude. We also conduct three case studies to show the effectiveness of density decomposition. The results indicate that core decomposition is likely to result in a higher density of lower-level cores than higher-level cores. Conversely, density-based decomposition ensures a monotonically decreasing trend in density for each layer, facilitating the successful discovery of dense subgraphs. Moreover, LDS decomposition can be well approximated by density decomposition while significantly reducing the number of redundant layers. Our experimental results demonstrate that density decomposition serves as a computationally efficient model for dense subgraph decomposition, accurately capturing the density structure of the graph. For reproducibility purposes, we release the source code of this paper at https://github.com/Flydragonet/Density_Decomposition_Computation.

2 PRELIMINARIES

Consider an undirected graph $G = (V, E)$, where V and E are the vertex set and edge set respectively. Let $n = |V|$ and $m = |E|$ be the number of vertices and edges in G. An *orientation* of G can be obtained by assigning a direction to every edge in E, resulting in a directed graph $\vec{G} = (V, \vec{E})$. For example, the directed graph in Fig. 1b is an orientation of the undirected graph in Fig. 1a. To eliminate confusion, we use angle brackets $\langle u, v\rangle$ to represent a directed edge in $\vec{G}$, while round brackets (u, v) denote an undirected edge in G. For a directed (resp., undirected) graph G and a vertex u, we use $d_u(G)$ (d_u for brevity) to denote the indegree (resp., degree) of u. A *path* in a directed graph $\vec{G} = (V, \vec{E})$ is a sequence of vertices $s = v_0, v_1, \ldots, v_{l-1}, t = v_l$, where $\langle v_{i-1}, v_i\rangle \in \vec{E}$, for $i = 1, \ldots, l$. We denote such a path as $s \rightsquigarrow t$, and the length of $s \rightsquigarrow t$ is l. If there exists a path $s \rightsquigarrow t$, we say that s can *reach* t. Besides, if an edge $\langle u, v\rangle$ is *reversed*, then it becomes $\langle v, u\rangle$. When a path is *reversed*, all edges in the path undergo a reversal. Before introducing the density decomposition, we first give the definitions of *reversible path* and *egalitarian orientation* as follows.

Definition 1. **(Reversible path)** *Given a graph G and its orientation $\vec{G}$, for a path $s \rightsquigarrow t$ in $\vec{G}$, if $d_t - d_s \geq 2$, we call that the path $s \rightsquigarrow t$ is a reversible path.*

Definition 2. **(Egalitarian orientation)** *Given a graph G and its orientation $\vec{G}$, if there is no reversible path in $\vec{G}$, then $\vec{G}$ is an egalitarian orientation.*

Intuitively, an egalitarian orientation distributes the indegree of vertices in the most equitable manner, i.e., minimizing the indegree difference between vertices as much as possible. Note that if reverse a reversible path $s \rightsquigarrow t$, d_s increases by 1, d_t decreases by 1, and the indegree of other vertices does not change, making the indegree difference between s and t reduced by 2. When no reversible path exists, the indegree difference can not be reduced anymore. Based on the egalitarian orientation, we give the definition of density decomposition which was originally proposed by Borradailie et al. [14] as follows.

Definition 3. **(Density decomposition)** *[14] Given an undirected graph $G = (V, E)$ and its arbitrary egalitarian orientation $\vec{G} = (V, \vec{E})$, the density decomposition of G is a set of subgraphs, denoted by $\mathcal{R} = \{R_k\}$, where for any non-negative integer k, $R_k \triangleq \{u \in V | d_u(\vec{G}) \geq k$ or u can reach a vertex v with $d_v(\vec{G}) \geq k\}$.*

Based on Definition 3, we give a definition of *integral dense number* as follows.

Definition 4. **(Integral dense number)** *For a vertex $u \in R_k \setminus R_{k+1}$, the integral dense number (IDN) of u is defined as $\bar{r}_u = k$.*

Example 1. *Taking the egalitarian orientation shown in Fig. 1b as an example, only v_5 has the indegree of 3. Consequently, R_3 contains v_5 and all vertices that can reach it, i.e., $R_3 = \{v_1, \ldots, v_7\}$. For vertices $v_1, \ldots, v_{14}$, their indegrees are larger than or equal to 2, and v_{15} can reach them, thus R_2 contains all vertices. Besides, R_1 and R_0 also contain all vertices, and for any integer $k > 3$, $R_k = \emptyset$. For the vertex $u \in R_3 \setminus R_4$, $\overline{r}_u$ is equal to 3, while for the vertex in $u \in R_2 \setminus R_3$, we have $\overline{r}_u = 2$.*

Borradaile et al. [14] shown that density decomposition exhibits a hierarchical structure, i.e., for two non-negative integers $i > j$, R_i is a subset of R_j. Let p be the largest integer such that R_p is non-empty. According to [7, 63], we can derive that p equals the *pseudoarboricity* of G, denoted by $p(G)$ or p for brevity. The pseudoarboricity is a classic metric to measure the sparsity of a graph which is defined as the minimum number of edge-disjoint pseudoforests into which the edges of a graph can be divided [48]. Here a pseudoforest is a graph in which each connected component contains at most one cycle. As indicated in [10], the pseudoarboricity of real-world graphs is often very small.

The number of layers of the density decomposition, from $R_0 = V$ to $R_{p+1} = \emptyset$, is $p + 2$, and the number of non-trivial layers, from R_1 to R_p, is p. Note that although a graph G may have many different egalitarian orientations, the density decomposition of G is unique [14]. Furthermore, Borradaile et al. [14] also proved the following result.

Lemma 1. *[14] Given an undirected graph $G = (V, E)$ and its arbitrary egalitarian orientation $\vec{G} = (V, \vec{E})$, if $u \in R_k \setminus R_{k+1}$, then $d_u(\vec{G}) \in \{k, k-1\} = \{\overline{r}_u, \overline{r}_u - 1\}$. Additionally, if $(u, v) \in E$ and $\overline{r}_u < \overline{r}_v$, then in $\vec{G}$, (u, v) must be oriented as $\langle v, u\rangle$ instead of $\langle u, v\rangle$.*

Lemma 1 shows that the indegree of each vertex is either equal to its IDN or IDN minus one, and the edges between layers must be oriented towards the endpoint with the smaller IDN. With these definitions, we formulate our problems as follows.

Problem definition. Given an undirected graph G, our goal is to efficiently compute the density decomposition of G, i.e., computing the IDNs for all vertex in G. For graphs with dynamic updates, we aim to maintain the density decomposition (i.e., maintain the IDNs for all vertex) when an edge is inserted or deleted.

By Definition 3, Definition 4, and Lemma 1, the key to compute the density decomposition is to find an egalitarian orientation. In the following sections, we will develop several more efficient algorithms to tackle this problem.

3 NEW THEORETICAL INSIGHTS

In this section, we first establish an interesting connection between density decomposition and locally-densest subgraph (LDS) decomposition, allowing us to leverage existing highly-optimized LDS decomposition algorithms, such as [42] and [35], for computing density decomposition. Then, we will show a close connection between density decomposition and core decomposition.

3.1 Connection with LDS Decomposition

Let $X \subseteq V$ be a non-empty subset of the vertex set. Define the *induced edges* of X as $E(X) = \{(x, y) \in E | x, y \in X\}$, and the *induced subgraph* of X in G is $(X, E(X))$. The *density* of X is characterized by $\rho(X) = |E(X)|/|X|$. The subset $X \subseteq V$ that maximizes the density $\rho(X)$ is recognized as the *densest subgraph* of G.

For two vertex subsets X and Y with $X \cap Y = \emptyset$, we define the *cross edges* between X and Y as $E_\times(X, Y) = \{(x, y) \in E | x \in X, y \in Y\}$, and the *additional edges* from X with respect to Y as $E_\Delta(X, Y) = E(X) \cup E_\times(X, Y)$, representing the increment in edges upon incorporating X into Y. Based on this concept, we define the outer density as follows.

Definition 5. **(Outer density)** *If $X \cap Y = \emptyset$, the* outer density *of a non-empty vertex set X with respect to a disjoint set Y is defined as $\rho(X, Y) \triangleq |E_\Delta(X, Y)|/|X|$. If $X \cap Y \neq \emptyset$, the focus is on the subset of X not included in Y, i.e., $X \setminus Y$. In this case, we define $E_\times(X, Y) \triangleq E_\times(X \setminus Y, Y)$, $E_\Delta(X, Y) \triangleq E_\Delta(X \setminus Y, Y)$, and $\rho(X, Y) \triangleq \rho(X \setminus Y, Y)$.*

With these concepts, we define LDS [61] as follows.

Definition 6. **(Locally-densest subgraph)** *Given an undirected graph $G = (V, E)$ and a vertex set $W \subseteq V$, W is an LDS if there is no $X \subseteq W$ and Y disjoint from W, such that $\rho(X, W \setminus X) \leq \rho(Y, W)$.*

To provide a more intuitive understanding, we rephrase the definition of LDS as follows: W is locally-densest if and only if $W = \emptyset$, or $W = V$, or $\min_{X \subseteq W} \rho(X, W \setminus X) > \max_{Y \cap W = \emptyset} \rho(Y, W)$. In other words, the minimum-density subgraph X within W still has a higher outer density compared to the maximum-density subgraph Y outside W. This demonstrates that W is "locally"-densest. Tatti [61] proved that all LDSes in a graph exhibit a hierarchical property, i.e., for any two LDSes X and Y, either $X \subseteq Y$ or $Y \subseteq X$. Therefore, LDSes can form a nested chain, which is the LDS decomposition.

Definition 7. **(LDS decomposition)** *[61] Given an undirected graph $G = (V, E)$, denote all of its non-empty LDSes by $\{B_0, \ldots, B_k\}$, such that $B_k \subsetneq B_{k-1} \subsetneq B_{k-2} \subsetneq \cdots \subsetneq B_0$. As $\emptyset$ is also an LDS, the LDS decomposition is defined as a set of LDSes, represented as $\mathcal{B} = \{B_{k+1} = \emptyset, B_k, \ldots, B_0\}$.*

By the LDS decomposition, we can obtain a density value for each vertex in an LDS which is referred to as the *fractional dense number*.

Definition 8. **(Fractional dense number)** *For a vertex $u \in B_i \setminus B_{i+1}$, the fractional dense number (FDN) of u is defined as $r_u = \rho(B_i, B_{i+1})$.*

Based on these definitions, Tatti [61] further proved several useful properties of the LDS decomposition, as shown in the following lemma.

Lemma 2. *For a graph G and its LDS decomposition $\mathcal{B}$, the following properties hold: (1) the smallest non-empty and largest LDSes, namely B_k and B_0, are the densest subgraph and V, respectively; (2) B_i exhibits strictly increasing density as i increases, i.e., for two vertices $u \in B_j \setminus B_{j+1}$ and $v \in B_i \setminus B_{i+1}$ with $j > i$, $r_u = \rho(B_j, B_{j+1}) > \rho(B_i, B_{i+1}) = r_v$ holds.*

Example 2. *Consider the graph shown in Fig. 1a. We can easily check that B_2, B_1, and B_0 are LDSes, and $B_{k+1} = B_3 = \emptyset$ is also locally-densest. The set $\{B_3, B_2, B_1, B_0\}$ is the LDS decomposition of G. The FDN r_u equals (1) $15/7$ for $u \in B_2 \setminus B_3$; (2) $8/4$ for $u \in B_1 \setminus B_2$; and (3) $7/4$ for $u \in B_0 \setminus B_1$. These FDNs exhibit strict monotonicity $15/7 > 8/4 > 7/4$. It is easy to verify that B_2 is the densest subgraph of G, which is consistent with the result shown in Lemma 2.*

Density decomposition and LDS decomposition are studied separately within two different research communities [14, 26, 61]. It is unclear whether there exists a direct connection between density decomposition and LDS decomposition. In this work, we fill this gap and establish, for the first time, a very close relationship between density decomposition and LDS decomposition, as shown in Theorem 1 and Theorem 2. Before introducing these theorems, we first present an important lemma. Due to the space limits, several proofs are replaced by proof sketches. All the complete proofs can be found in the full version of this paper from https://github.com/Flydragonet/Density_Decomposition_Computation.

Lemma 3. *Given an undirected graph G and an orientation $\vec{G}$, let k be a non-negative integer and T be the vertex set with $T \triangleq \{u \in V | d_u(\vec{G}) \geq k$ or u can reach a vertex v with $d_v(\vec{G}) \geq k\}$. If (1) for all $u \in T$, $d_u(\vec{G}) \geq k-1$; (2) for all $u \in V \setminus T$, $d_u(\vec{G}) \leq k-1$; (3) all edges in $E_\times(T, V \setminus T)$ point towards $V \setminus T$, then T is an LDS.*

Proof. When $T = V$ or $T = \emptyset$, T is obviously locally-densest. Next, we assume $T \neq V$ and $T \neq \emptyset$.

For all non-empty $X \subseteq T$ and non-empty Y disjoint from T, we have $\rho(Y,T) = \frac{|E_\times(Y,T)|+|E(Y)|}{|Y|} \overset{(i)}{=} \frac{\sum_{u\in Y} d_u(\vec{G})}{|Y|} \overset{(iii)}{\leq} k-1$, $\rho(X, T \setminus X) = \frac{|E_\times(X,T\setminus X)|+|E(X)|}{|X|} \overset{(ii)}{\geq} \frac{\sum_{u\in X} d_u(\vec{G})}{|X|} \overset{(iv)}{\geq} k-1$.

In a directed graph, the sum of the indegree for all vertices in a set equals the number of directed edges ending in that set. This leads to equality (i) and inequality (ii), as all edges in $E_\times(Y,T)$ are directed towards Y, while there may be edges in $E_\times(X, T \setminus X)$ not directed toward X. Further, the inequalities (iii) and (iv) can be derived by the first two conditions in Lemma 3.

Next, we examine inequalities (ii) and (iv) to demonstrate their strictness. Concerning (iv), if there exists a vertex $u \in X$ with an indegree greater than $k-1$, then (iv) can be refined to a strict inequality. Otherwise, according to the definition of T, at least one edge in $E_\times(X, T \setminus X)$ is directed towards $T \setminus X$, thereby (ii) can be tightened to a strict inequality. Therefore, inequalities (ii) and (iv) have at least one strict inequality, and we can derive that $\rho(X, T \setminus X) > k-1 \geq \rho(Y,T)$, indicating that T is an LDS. □

Based on Lemma 3, we can obtain the following results.

Theorem 1. *Given an undirected graph G and its density decomposition $\mathcal{R} = \{R_k\}$, for all $k = 0, 1, \ldots$, R_k is an LDS.*

Proof. In Lemma 3, the set R_k conforms to the definition of T, and, by Lemma 1, all three conditions in Lemma 3 are satisfied. Therefore, $R_k = T$ and thus it is an LDS. □

Theorem 2. *Given an undirected graph $G = (V,E)$ and a vertex $v \in V$, $\overline{r}_v = \lceil r_v \rceil$ holds.*

Proof. For an arbitrary vertex $v \in V$, suppose $v \in R_i \setminus R_{i+1}$ (i.e., $\overline{r}_v = i$) and $v \in B_j \setminus B_{j+1}$ (i.e., $r_v = \rho(B_j, B_{j+1})$). According to Theorem 1, R_i and R_{i+1} are LDSes, and we denote $B_{u+1} = R_{i+1}$ and $B_l = R_i$. Because of the hierarchy of LDSes, it follows that $B_u \subseteq B_j \subseteq B_l$, implying $\rho(B_u, B_{u+1}) \geq \rho(B_j, B_{j+1}) \geq \rho(B_l, B_{l+1})$ by Lemma 2. In the proof of Lemma 3, consider $T = R_i$ and $X = B_l \setminus B_{l+1}$. We can derive $\rho(B_l \setminus B_{l+1}, R_i \setminus (B_l \setminus B_{l+1})) = \rho(B_l, B_{l+1}) > i-1$. Let $T = R_{i+1}$ and $Y = B_u \setminus B_{u+1}$, we have $i \geq \rho(B_u \setminus B_{u+1}, R_{i+1}) = \rho(B_u, B_{u+1})$. In conclusion, $i \geq \rho(B_j, B_{j+1}) > i-1$ holds and thus $\overline{r}_v = i = \lceil \rho(B_j, B_{j+1}) \rceil = \lceil r_v \rceil$. □

The above theorems reveal that: (1) every layer of density decomposition is an LDS; and (2) each IDN is the round-up value of the FDN. It is worth noting that the number of LDSes can be extensive and only be bounded by $O(n)$. In contrast, the density decomposition groups LDSes based on the round-up of their densities, resulting in a number of non-trivial layers equaling the pseudoarboricity p, which is typically much smaller than n. This characteristic ensures that the density decomposition is not only computationally efficient but also retains a significant amount of information about the density structure of the graph.

3.2 Connection with Core Decomposition

Here, we reveal the relationship between the density decomposition and the core decomposition. The definitions of *k-core* and *core decomposition* are given as follows.

Definition 9. **(k-core and core decomposition)** *Given an undirected graph G and an integer k, the k-core of G, denoted by C_k, is the maximal subgraph in which every vertex has a degree of at least k. The core decomposition of G is a set of subgraphs, denoted by $\mathcal{C} = \{C_k\}$, containing all k-cores of G.*

As previously mentioned, core decomposition may not always accurately reflect the density structure of a graph. However, we find that it can serve as an approximation of density decomposition. Below, we present two theorems, one of which demonstrates the 2-approximation relationship of the outer density of these two decompositions. The other one shows that they are sandwiched by each other, also following a factor of 2.

Theorem 3. *Given an undirected graph $G = (V,E)$, for $k = 0, 1, \ldots, p$, $\rho(C_k, C_{k+1}) \geq \rho(R_k, R_{k+1})/2$.*

Proof. Let $\vec{G}$ be an arbitrary egalitarian orientation of G. Denote $C_k \setminus C_{k+1}$ as C and $R_k \setminus R_{k+1}$ as R. We have $\rho(C_k, C_{k+1}) = \frac{|E_\Delta(C_k, C_{k+1})|}{|C|} \geq \frac{\sum_{u\in C} d_u(C_k)}{2|C|} \geq \frac{k}{2} \geq \frac{\sum_{u\in R} d_u(\vec{G})}{2|R|} = \frac{\rho(R_k, R_{k+1})}{2}$. □

Theorem 4. **(Sandwich Theorem)** *Given an undirected graph G, for any non-negative integer k, we have $C_{2k} \subseteq R_k \subseteq C_k \subseteq R_{\lceil k/2 \rceil}$.*

Proof. To prove $R_k \subseteq C_k$, if $R_k = \emptyset$, then it evidently holds. If $R_k \neq \emptyset$, we show that in the subgraph of G induced by R_k, denoted as $G(R_k)$, every vertex has degree at least k. Let $\vec{G}$ be an arbitrary egalitarian orientation of G. Recall that all edges between R_k and $V \setminus R_k$ are directed toward $V \setminus R_k$ in $\vec{G}$. For each vertex u in R_k, if $d_u(\vec{G}) \geq i$, then obviously $d_u(G(R_k)) \geq d_u(\vec{G}) \geq i$. On the other hand, if $d_u(\vec{G}) = k-1$, since $u \in R_k$, u must have an out-edge enabling it to reach a vertex in R_k whose indegree is at least k. Thus $d_u(G(R_k)) \geq d_u(\vec{G}) + 1 \geq k$. In summary, $R_k \subseteq C_k$.

To prove $C_k \subseteq R_{\lceil k/2 \rceil}$, if $C_k = \emptyset$, then it evidently holds. If $C_k \neq \emptyset$, let u be an arbitrary vertex in C_k, below we show that $u \in R_{\lceil k/2 \rceil}$. Given an arbitrary egalitarian orientation $\vec{G}$ of G, let $S \triangleq \{v | v \in C_k$ and u can reach v in $\vec{G}\}$. Note that all edges belonging to $E_\times(S, C_k \setminus S)$ are directed toward S in $\vec{G}$. Therefore, we have $\sum_{v\in S} d_v(\vec{G}) \geq |E_\times(S, C_i \setminus S)| + |E(S)|$. By the definition of k-core, $|E_\times(S, C_k \setminus S)| + |E(S)| \geq \sum_{v\in S} d_v(C_k)/2 \geq k|S|/2$. That is, the sum of indegree of vertices in S is at least $k|S|/2$. Then by the pigeonhole principle, at least one vertex $t \in S$ has indegree at least $\lceil k/2 \rceil$ in $\vec{G}$. By the definition of S and density decomposition, u can reach t and thus $u \in R_{\lceil k/2 \rceil}$.

Since $C_k \subseteq R_{\lceil k/2 \rceil} \Rightarrow C_{2k} \subseteq R_k$, $C_{2k} \subseteq R_k$ also holds. This completes the proof. □

4 ALGORITHMS FOR STATIC GRAPHS

4.1 Existing Algorithms

Existing algorithms for computing the density decomposition. Borradaile [13, 14] proposed the Path algorithm, shown in Algorithm 1, which provides a basic approach to obtain an egalitarian orientation. Since the egalitarian orientation requires that there is no reversible path $s \rightsquigarrow t$, the algorithm repeatedly searches for such paths until none can be found. Once the algorithm terminates, an egalitarian orientation can be obtained, and then R_k

Algorithm 1: Path(G)

Input: An undirected graph G.
Output: An egalitarian orientation $\vec{G}$.

Arbitrarily orient the edges in G to obtain $\vec{G}$;
while *there is a reversible path* $s \rightsquigarrow t$ *in* $\vec{G}$ **do**
  Reverse $s \rightsquigarrow t$;
return $\vec{G}$;

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

Input: An orientation $\vec{G} = (V, \vec{E})$ and an integer k.
Output: The updated $\vec{G}$ and the layer R_k of the density decomposition.

$V' \leftarrow V \cup \{s, t\}$, where s is source and t is sink;
$d \leftarrow k - 1$;
foreach $\langle u, v\rangle \in \vec{E}$ **do**
  Add arc $\langle u, v\rangle$ to A and let $c(u, v) \leftarrow 1$;
foreach $u, d_u(\vec{G}) < d$ **do**
  Add arc $\langle s, u\rangle$ to A and let $c(s, u) \leftarrow d - d_u(\vec{G})$;
foreach $u, d_u(\vec{G}) > d$ **do**
  Add arc $\langle u, t\rangle$ to A and let $c(u, t) \leftarrow d_u(\vec{G}) - d$;
Compute the maximum flow value f_{max} of (V', A, c);
foreach $\langle u, v\rangle \in \vec{E}$ **do** `// Copy the residual network to` $\vec{G}$
  if $\langle u, v\rangle \in A$ *is saturated* **then** reverse the edge $\langle u, v\rangle \in \vec{E}$;
$R_k \leftarrow \{u \in V | d_u(\vec{G}) \geq d$ or u can reach a vertex v with $d_v(\vec{G}) \geq d\}$;
return $(\vec{G}, R_k)$;

can be identified layer by layer based on the definition of density decomposition [13]. The time complexity of the Path algorithm is $O(m^2)$ as shown in [13]. Clearly, such a high time complexity of the Path algorithm poses a significant challenge when handling large graphs.

Existing algorithms for LDS decomposition. As shown in Theorem 2, each IDN is a round-up value of FDN. Thus, we can obtain the density decomposition by computing the LDS decomposition first and then rounding up the FDNs. Tatti [61] proposed a network flow algorithm to compute LDS decomposition with a time complexity of $O(n^2m)$. Then, Danisch et al. [26] proposed the current state-of-the-art FW algorithm. The algorithm initially employs convex optimization to obtain an approximate LDS decomposition and then invokes Tatti's algorithm to calculate the exact LDS decomposition. Subsequently, Harb et al. [35] introduced a new approximate algorithm based on the proximal gradient method, which can be integrated into the FW algorithm, resulting in the Fista algorithm. Both of these two highly-optimized algorithms can significantly improve the practical performance, but the worst-case time complexity remains $O(n^2m)$. As a consequence, these LDS-based algorithms may still perform poorly on large graphs due to their high worst-case time complexities, as confirmed in our experiments.

4.2 A Re-orientation Network Flow Algorithm

As discussed above, existing algorithms for density decomposition suffer from the high complexity issue. To overcome this problem, we develop a novel algorithm based on a so-called re-orientation network flow technique. The re-orientation network flow technique was originally devised to compute the pseudoarboricity of a graph [7]. In this paper, we discover that such a powerful network flow technique can also be used to compute the density decomposition. Below, we introduce the concept of the re-orientation network.

Definition 10. **(Re-orientation network)** *[7] Given an orientation $\vec{G} = (V, \vec{E})$ and an integer d, the re-orientation network is a weighted network with an additional source vertex s and sink vertex t where the weight of each edge denotes the capacity of the edge. Specifically, the re-orientation network with parameter d is defined as $(V \cup \{s, t\}, A, c)$ where (1) $\langle u, v\rangle \in A, c(u, v) = 1$, if $\langle u, v\rangle \in E$; (2) $\langle s, u\rangle \in A, c(s, u) = d - d_u(\vec{G})$, if $d_u(\vec{G}) < d$; and (3) $\langle u, t\rangle \in A, c(u, t) = d_u(\vec{G}) - d$, if $d_u(\vec{G}) > d$.*

By Definition 10, the re-orientation network uses a parameter d to separate vertices based on their indegree. The source s is connected to vertices with an indegree less than d, while the sink t is linked to vertices with indegree greater than d. Based on this, we can perform a maximum flow computation on this re-orientation network. When the maximum flow algorithm terminates, there does not exist a path from s to t (i.e., no augmentation path exists in the residual network). That is to say, in this case, the vertices with indegree less than d cannot reach the vertices with indegree larger than d. Let T be the set of vertices that can reach t. When the maximum flow algorithm terminates, only the vertices in T have degrees larger than d. As a consequence, the set T satisfies the three conditions in Lemma 3, indicating that T is an LDS. Below, we further prove that the set T defined in Lemma 3 is exactly R_{d+1}.

Lemma 4. *The subgraph T defined in Lemma 3 is R_k.*

Proof. By the proof of Lemma 3, when $T \neq \emptyset$ and $T \neq V$, for all $X \subseteq T$ and Y disjoint from T, we have $\rho(X, T \setminus X) > k - 1 \geq \rho(Y, T)$. Suppose that T is the LDS B_i. Then, according to the above inequalities, $\rho(B_i \setminus B_{i+1}, B_{i+1}) > k - 1 \geq \rho(B_{i-1} \setminus B_i, B_i)$. Thus, the FDNs of vertices in T is strictly greater than $k - 1$, while the FDNs of vertices in $V \setminus T$ is less than or equal to $k - 1$. By Lemma 2, the round-up FDN is equal to IDN. As a result, all vertices in T have IDNs no less than k, and all vertices in $V \setminus T$ have IDNs less than k, which proves $T = R_k$. When $T = \emptyset$ or $T \neq V$, only the inequality $k - 1 \geq \rho(Y, T)$ or $\rho(X, T \setminus X) > k - 1$ holds, respectively. Using the same argument, the lemma can be easily established. □

By Lemma 4, we devise the re-orientation network flow algorithm, namely GetLayer, which is outlined in Algorithm 2. Below, we show that each invocation of GetLayer($\vec{G}, k$) can correctly output one layer of density decomposition, i.e., R_k.

Theorem 5. *Algorithm 2 correctly outputs R_k.*

Proof. First, an important observation is that in the output orientation $\vec{G}$, there is no path $s \rightsquigarrow t$ such that $d_s(\vec{G}) < k - 1$ and $d_t(\vec{G}) > k - 1$. If such a path exists, then the flow value can be increased by filling this path. As the flow reaches its maximum value, such paths no longer exist.

To avoid confusion, we use T to denote the output R_k of the GetLayer algorithm. Next, we prove that T equals the layer R_k in the density decomposition. T contains all vertices with indegree greater than $k - 1$ or reachable to a vertex with indegree greater than $k - 1$. Since vertices with indegree less than $k - 1$ cannot reach vertices with indegree greater than $k - 1$, they are not in T. As a result, the vertices in T all have indegree greater than or equal to $k - 1$, the vertices in $V \setminus T$ have indegree less than or equal to $k - 1$, and the edges in $E_\times(T, V \setminus T)$ all point to $V \setminus T$. By Lemma 4, $T = R_k$. □

The time complexity of Algorithm 2 depends on computing the maximum flow in the re-orientation network. It was shown that

Algorithm 3: Flow(G)

Input: An undirected graph G.
Output: Density decomposition $\mathcal{R} = \{R_k\}$.

1 Invoke a 2-approximation algorithm to obtain an approximate orientation $\vec{G}$;
2 **foreach** $k = 0, 1, 2, \ldots$ **do**
3 $\quad(\vec{G}, R_k) \leftarrow \text{GetLayer}(\vec{G}, k)$;
4 $\quad$**if** $R_k = \emptyset$ **then break**;
5 $\quad\vec{G} \leftarrow$ the induced subgraph of R_k in $\vec{G}$; `// pruning strategy`
6 **return** $\mathcal{R} = \{R_k\}$;

the re-orientation network maximum flow algorithm can be implemented with time complexity of $O(m^{3/2})$ and space complexity of $O(m)$ [10], and thus Algorithm 2 has the same complexity.

4.3 The Proposed Flow Algorithm

As shown in Theorem 5, each invocation of the GetLayer algorithm outputs one layer of density decomposition. Consequently, simply invoking GetLayer $p + 2$ times can yield all layers from $R_0 = V$ to $R_{p+1} = \emptyset$. Based on this, we propose the Flow algorithm for density decomposition, as shown in Algorithm 3.

Algorithm 3 first invokes an existing linear-time 2-approximation algorithm to obtain an orientation [28] (Line 1). Note that with such a 2-approximation initial orientation, the maximum flow computation can be faster than that with arbitrary initial orientation, because this approach is expected to reduce the number of augmentation paths in the maximum flow computation as indicated in [10]. Then, the Flow algorithm iteratively performs GetLayer from the bottom layer R_0 to yield each layer one by one (Lines 2-5). Once R_k is computed, the algorithm can directly assign $\vec{G}$ as the induced subgraph of R_k in $\vec{G}$ by discarding the vertices in $V \setminus R_k$ (Line 5). Since the indegree of vertices in $V \setminus R_k$ is no larger than $k - 1$, they cannot reach vertices in R_k. In the following 'foreach' loop, the vertices in $V \setminus R_k$ are all connected to the source s, but the flow cannot enter R_k from $V \setminus R_k$. Thus, Flow only needs to compute R_{k+1} within R_k in the subsequent computation, which can significantly improve the efficiency. When k equals $p + 1$, R_k becomes an empty set, indicating that all layers of the density decomposition are obtained, thus the Flow algorithm terminates. The correctness of Flow can be guaranteed by Theorem 5. Below, we analyze the complexity of Flow.

Theorem 6. *The worst-case time and space complexity of Algorithm 3 is $O(p \cdot m^{3/2})$ and $O(m + n)$ respectively.*

Proof. Algorithm 3 invokes the GetLayer algorithm $p+2$ times, and each invocation consumes $O(m^{3/2})$ time. Therefore, the total time complexity of Flow is $O(p \cdot m^{3/2})$. For the space complexity, it is easy to see that the algorithm only consumes $O(m)$ space. □

Note that since $p \leq \sqrt{m}$ [20], the worst-case time complexity of Algorithm 3 is lower than that of Algorithm 1. Furthermore, p is often small in real-world graphs [10], thus the practical performance of Algorithm 3 is typically much better than the worst-case time complexity suggests. Indeed, as shown in our experiments, Algorithm 3 is around two orders of magnitude faster than Algorithm 1 on most datasets.

4.4 The Proposed Flow++ Algorithm

In this subsection, we propose an improved algorithm Flow++. By using a newly-developed divide-and-conquer technique, Flow++ improves the time complexity of Flow from $O(p \cdot m^{3/2})$ to $O(\log p \cdot m^{3/2})$. The key idea of Flow++ is that it partitions the graph into

Algorithm 4: Flow++(G)

Input: An undirected graph G.
Output: The egalitarian orientation $\vec{G}$ and density decomposition $\{R_k\}$.

1 Invoke a 2-approximation algorithm to obtain an approximate orientation $\vec{G}$ and a 2-approximation of pseudoarboricity $\bar{p}$;
2 $R_{\bar{p}+1} \leftarrow \emptyset$; $R_0 \leftarrow V$;
3 Divide($R_{\bar{p}+1}, R_0$);
4 **return** $(\vec{G}, \{R_k\})$;
5 **Function** Divide(R_u, R_l)
6 $\quad$**if** $u - l \leq 1$ *or* $R_u = R_l$ **then return**;
7 $\quad k_u \leftarrow u, k_l \leftarrow l$;
8 $\quad$**while** $k_u > k_l$ **do**
9 $\quad\quad k \leftarrow \lfloor (k_u + k_l + 1)/2 \rfloor$;
10 $\quad\quad(\vec{G}, R_k) \leftarrow \text{GetLayer}(\vec{G}, k)$;
11 $\quad\quad$**if** $|E_\Delta(R_l, R_k)| < |E_\Delta(R_l, R_u)|/2$ **then** $k_l \leftarrow k$;
12 $\quad\quad$**else** $k_u \leftarrow k - 1$;
13 $\quad k \leftarrow k_l$;
14 $\quad$Divide(R_k, R_l);
15 $\quad$Divide(R_u, R_{k+1});

almost equal parts at each step and performs maximum flow computation recursively in each part. Recall that in Flow, we compute R_k within R_{k-1} instead of the whole graph G due to the hierarchical structure of density decomposition. Similar to this strategy, an improved strategy is to compute R_k between R_u and R_l, where $u > k > l$, which can intuitively further reduce the data scale. To construct such R_u and R_l, a divide-and-conquer manner can be employed to recursively compute all layers of density decomposition. Specifically, the recursive function, denoted as Divide(R_u, R_l), is designed to find all R_k for $u > k > l$. It first selects a parameter k where $u > k > l$ and divides the graph into two parts: $R_l \setminus R_k$ and $R_{k+1} \setminus R_u$. Then, Divide(R_k, R_l) and Divide(R_u, R_{k+1}) are separately invoked for deeper recursions.

Equipped with such a divide-and-conquer approach, the improved algorithm Flow++ is proposed to compute the density decomposition, which is outlined in Algorithm 4. Specifically, Algorithm 4 starts by invoking an approximate algorithm to calculate a 2-approximate pseudoarboricity $\bar{p} \geq p$ (Line 1). Then, we can obtain two trivial layers: $R_{\bar{p}+1} = \emptyset$ and $R_0 = V$ (Line 2). The recursion begins with these two initial layers by calling Divide($R_{\bar{p}+1}, R_0$) (Line 3). During this invocation, a parameter k is selected to divide the graph (Lines 7-13). We use a binary search (Lines 7-13) to find the maximum k satisfying $|E_\Delta(R_l, R_k)| < |E_\Delta(R_l, R_u)|/2$. Thus, for $k + 1$, we have $|E_\Delta(R_l, R_{k+1})| \geq |E_\Delta(R_l, R_u)|/2$, suggesting that $|E_\Delta(R_{k+1}, R_u)| = |E_\Delta(R_l, R_u)| - |E_\Delta(R_l, R_{k+1})| < |E_\Delta(R_l, R_u)|/2$. Therefore, both the edges $E_\Delta(R_l, R_k)$ and the edges $E_\Delta(R_{k+1}, R_u)$ are less than half of the edges $E_\Delta(R_l, R_u)$. This ensures the number of edges being divided by 2 in each depth of recursion. When $u - l \leq 1$ or $R_u = R_l$, there are no further subdivided layers between R_u and R_l, thus Divide can return directly. The correctness of the Flow++ algorithm can be easily derived by induction because in each recursion we can identify a R_k and no layer will be missed by the divide-and-conquer procedure. Thanks to our divide-and-conquer technique, the graph size is halved with each recursion. As a result, the computation costs exponentially decrease for the higher-level layers. This is why our Flow++ algorithm can achieve a lower time complexity compared to the Flow algorithm. Below, we use an example to illustrate how Flow++ works.

Example 3. *Assume the input of* Flow++ *is the* Citeseer *dataset* ($|V| = 384,054, |E| = 1,736,145$). Flow++ *first computes the 2-approximation pseudoarboricity $\bar{p} = 16$. Subsequently, the* Divide *algorithm is invoked for recursive computation. Fig. 2 illustrates the*

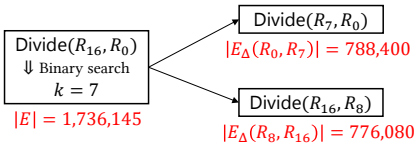


Figure 2: Illustration of the Flow++ algorithm on the Citeseer dataset ($|V| = 384,054$, $|E| = 1,736,145$).

first recursion Divide(R_{16}, R_0). *The algorithm initially employs binary search to determine $k = 7$ and then divides the graph $R_0 \setminus R_{16}$ into two parts: $R_8 \setminus R_{18}$ and $R_0 \setminus R_7$. Both $|E_\Delta(R_0, R_7)|$ and $|E_\Delta(R_8, R_{16})|$ are less than half of $|E| = |E_\Delta(R_0, R_{16})|$, achieving the partition of the graph into two smaller subgraphs for divide-and-conquer. Subsequently, deeper recursion is performed independently on the two smaller subgraphs.*

The remaining challenge in the above divide-and-conquer algorithm lies in implementing the computation within $R_l \setminus R_u$. We find that we can slightly modify the re-orientation network flow algorithm to achieve this goal. Specifically, we need to consider not only the subgraph induced by $R_l \setminus R_u$ in $\vec{G}$ but also the cross edges $E_\times(R_l, R_u)$. In GetLayer, arcs from the source and sink to vertices in only $R_l \setminus R_u$ are connected (the other vertices are pruned). However, for computing the indegree of the vertices in $R_l \setminus R_u$, the edges in $E_\times(R_l, R_u)$ are also considered. The reasons why these modifications of GetLayer can ensure the correctness are as follows. Because R_l and R_u are either previously computed by the GetLayer algorithm or are the trivial subgraphs $\emptyset$ or V. According to the properties of GetLayer, the edges in $E_\times(V \setminus R_u, R_u)$ all point towards $V \setminus R_u$, and the edges in $E_\times(V \setminus R_l, R_l)$ all point towards $V \setminus R_l$. This indicates that at this point when the GetLayer algorithm is called again for further computation, there will be no flow from $V \setminus R_u$ to R_u, and no flow from $V \setminus R_l$ to R_l. As a result, we can safely ignore $V \setminus R_l$ and R_u, which do not have any flow passing through them, and only connect the vertices in $R_l \setminus R_u$ to the source and sink.

Clearly, with this divide-and-conquer implementation, the size of the network can be bounded by $E_\Delta(R_l, R_u)$ in each recursion. Thus, by [10], the time complexity of GetLayer on subgraph $R_l \setminus R_u$ is $O(|E_\Delta(R_l, R_u)|^{3/2})$. Based on this, we can derive the total time and space complexity of Flow++ as follows.

Theorem 7. *The time complexity of Algorithm 4 is $O(m^{3/2} \cdot \log p)$, and the space complexity is $O(p \cdot n)$.*

Proof. The time complexity of Divide(R_l, R_u) consists of two parts: the complexity of the binary search procedure in each recursion and the complexity of two deeper recursions Divide(R_k, R_l) and Divide(R_u, R_{k+1}). Since the binary search range is $O(p)$, it only requires $O(\log p)$ rounds of binary search. Therefore, the time complexity of the binary search in Divide(R_u, R_l) is $O(|E_\Delta(R_l, R_u)|^{3/2} \cdot \log p)$. For deeper recursions, the binary search in Divide ensures that k is the largest integer satisfying $|E_\Delta(R_l, R_k)| < |E_\Delta(R_l, R_u)|/2$, which guarantees that both $|E_\Delta(R_l, R_k)|$ and $|E_\Delta(R_{k+1}, R_u)|$ are less than $|E_\Delta(R_l, R_u)|/2$. Thus, the data scale is divided by 2 in each recursive layer. According to the master theorem for divide-and-conquer algorithms [5], the time complexity of deeper recursions in Divide(R_u, R_l) is no higher than the time complexity of the binary search. Hence, it is easy to show that the total time complexity of Flow++ is $O(m^{3/2} \cdot \log p)$.

For the space complexity, GetLayer takes $O(m)$ space, and each recursion depth needs to store information for R_u and R_l, occupying $O(n)$ space. Since the number of layers does not exceed $O(p)$, the space complexity for storing R_u and R_l is $O(p \cdot n)$. Because $p \geq m/n \Rightarrow O(m) \subseteq O(p \cdot n)$ [48], the space complexity of Flow++ is $O(p \cdot n)$. □

Note that since p is small in real-world graphs, $\log p$ is a very small constant. For example, for most real-world graphs where $p \leq 1024$, we have $\log_2 p \leq 10$. Consequently, the complexity of our algorithm is near to $O(m^{3/2})$, making it highly efficient for processing real-world graphs. Indeed, as shown in our experiments, Flow++ is extremely efficient, and it can further achieve a significant speedup compared to the Flow algorithm.

5 DYNAMIC MAINTENANCE ALGORITHMS

When the graph is updated by an edge insertion or deletion, a straightforward algorithm to maintain the density decomposition is to invoke the Flow++ algorithm to re-compute the density decomposition from scratch. Clearly, such an approach is costly. To improve the efficiency, in this section, we develop several novel algorithms to maintain the density decomposition when the graph is updated by an edge insertion or deletion. To avoid confusion, we assume that the notations, including IDN $\bar{r}$, indegree d, and R_k, all denote their values *before* edge updated unless otherwise specified.

5.1 Density Decomposition Update Theorem

In this subsection, we establish a density decomposition update theorem, based on which we can devise the maintenance algorithms.

Theorem 8. **(Density decomposition update theorem)** *When inserting (resp. deleting) an edge (u, v) in the undirected graph G, assuming $\bar{r}_v \leq \bar{r}_u$, only the IDNs of vertices whose IDN equals $\bar{r}_v$, i.e., vertices in $R_{\bar{r}_v} \setminus R_{\bar{r}_v+1}$, may change, and it can only increase by 1 (resp. decrease by 1).*

Proof. We first analyze the case of edge insertion. Prior to the insertion operation, based on Theorem 1 and Lemma 3, we can conclude that the inequality $\rho(X, R_k \setminus X) > k - 1$ holds for any R_k and non-empty $X \subseteq R_k$. The subsequent insertion of (u, v) does not decrease $\rho(X, R_k \setminus X)$, thus $\rho(X, R_k \setminus X) > k - 1$ is satisfied. As a result, the FDNs of vertices in R_k remain larger than $k - 1$. By Theorem 2, the IDNs are no less than k, indicating that inserting (u, v) does not decrease the IDNs.

Next, we show that only the IDNs of vertices within $R_{\bar{r}_v} \setminus R_{\bar{r}_v+1}$ may increase by at most 1. Let $\vec{G}$ be an egalitarian orientation of G before the insertion of (u, v). For any non-negative integer $k \neq \bar{r}_v + 1$, suppose that we directly insert $\langle u, v \rangle$ into $\vec{G}$. The condition $\bar{r}_v \leq \bar{r}_u$ ensures that the edges in $E_\times(R_k, V \setminus R_k)$ still point to $V \setminus R_k$. Moreover, the insertion of $\langle u, v \rangle$ only alters the indegree of v, increasing by 1. Therefore, the indegree of vertices in R_k continues to be no less than $k - 1$, and the indegree of vertices in $V \setminus R_k$ remains no larger than $k - 1$. The analysis confirms that R_k obeys the three conditions in Lemma 3 and Lemma 4, and thus we can conclude that R_k remains unchanged after inserting (u, v). However, when $k = \bar{r}_v + 1$, the indegree of the vertex $x \in V \setminus R_k$ may reach k. Since the IDN of each vertex does not decrease, it can be derived that after inserting (u, v), only vertices in $R_{\bar{r}_v} \setminus R_{\bar{r}_v+1}$ might be merged into $R_{\bar{r}_v+1} \setminus R_{\bar{r}_v+2}$, leading to an increment of 1 in the IDNs.

For edge deletion, using a similar argument, we can easily derive that: (1) the removal of (u, v) does not increase the IDNs of vertices; (2) for any non-negative integer $k \neq \bar{r}_v$, R_k remains unchanged; (3) only the vertices in $R_{\bar{r}_v} \setminus R_{\bar{r}_v+1}$ might be merged into $R_{\bar{r}_v-1} \setminus R_{\bar{r}_v}$, resulting in a decrement of 1 in the IDNs. This completes the proof. □

Algorithm 5: Insert$(\vec{G}, \overline{r}, (u, v))$

Input: The egalitarian orientation $\vec{G}$, the IDNs of all vertices $\overline{r}$, and the edge (u, v) to be inserted.

Output: The updated egalitarian orientation $\vec{G}$ and IDNs $\overline{r}$.

1 Suppose $\overline{r}_v \leq \overline{r}_u$, otherwise swap the input edge (u, v);
2 **if** $d_v = \overline{r}_v - 1$ **then** $\vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
3 **else** // $d_v = \overline{r}_v$
4 $\quad \vec{G} \leftarrow \vec{G} \cup \langle u, v \rangle$;
5 $\quad$ **if** *there is a reversible path* $s \rightsquigarrow v$, *where* $d_s = \overline{r}_v - 1$ **then**
6 $\quad\quad$ Reverse the path;
7 $\quad$ **else**
8 $\quad\quad$ **foreach** $w \in \{w | \overline{r}_w = \overline{r}_v$ *and* w *can reach* $v\}$ **do** $\overline{r}_w \leftarrow \overline{r}_w + 1$;
9 $\quad\quad \overline{r}_v \leftarrow \overline{r}_v + 1$;
10 **return** $(\vec{G}, \overline{r})$;

Theorem 8 indicates that maintaining density decomposition is highly localized. That is, we only need to consider the vertices with IDNs equal to $\overline{r}_v$. Moreover, given that the density decomposition is defined based on an egalitarian orientation, the key to updating the density decomposition lies in maintaining the egalitarian orientation, building upon which we develop several efficient density decomposition maintenance algorithms in the following.

5.2 The Proposed Insertion Algorithm

Based on Theorem 8, we propose an insertion algorithm, called Insert, to handle the case of edge insertion. The detailed implementation of our algorithm is shown in Algorithm 5. The main idea of Insert is as follows. By Theorem 8, Insert needs to consider three different cases: (1) $d_v = \overline{r}_v - 1$ (Line 2); (2) $d_v = \overline{r}_v$ and there exists $s \rightsquigarrow v$ (Lines 3-6); (3) $d_v = \overline{r}_v$ and there is no $s \rightsquigarrow v$ (Lines 3-4 and Lines 7-9). In case (1), the Insert algorithm directly inserts $\langle u, v \rangle$. Following this operation, $\vec{G}$ is still an egalitarian orientation, and the IDNs do not need to be updated. In case (2), Insert also inserts $\langle u, v \rangle$, but a reversible path $s \rightsquigarrow v$ is created, making $\vec{G}$ non-egalitarian. Consequently, Insert promptly reverses this path, restoring $\vec{G}$ to an egalitarian state. At this point, the IDNs also do not change. Note that for case (2), we can perform Breadth-First Search (BFS) in the reversed graph from the vertex v to find the reversible path $s \rightsquigarrow v$ (Lines 5-6). In case (3), where no reversible path is found, $\vec{G}$ remains an egalitarian orientation, but the IDNs of some vertices need to be increased by 1 according to Theorem 8. Similarly, in this case, we can also invoke a reverse BFS algorithm from v to identify the set $\{w | \overline{r}_w = \overline{r}_v$ and w can reach $v\}$ (Line 8). Below, we prove that Insert correctly maintains the density decomposition (i.e., maintaining all IDNs for all vertices).

Theorem 9. *Algorithm 5 correctly maintains an egalitarian orientation and all IDNs for all vertices.*

Proof. We prove the theorem by considering three cases, namely: (1) $d_v = \overline{r}_v - 1$; (2) $d_v = \overline{r}_v$ and there exists $s \rightsquigarrow v$; (3) $d_v = \overline{r}_v$ and there is no $s \rightsquigarrow v$.

We first prove that Insert correctly updates the egalitarian orientation, i.e., proving that the output orientation has no reversible path. For case (1), since $d_v + 1 = \overline{r}_v \leq \overline{r}_u$, even if we directly insert $\langle u, v \rangle$ into $\vec{G}$, $\vec{G}$ still satisfies the properties in Lemma 1, and it is clear that $\vec{G}$ has no reversible path. In the case of (2), because for any k, the vertices in $V \setminus R_k$ cannot reach the vertices in R_k, combining that all vertices on the path $s \rightsquigarrow v$ can reach v, all these vertices also must all be in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. Reversing the path $s \rightsquigarrow v$ reduces d_v by 1, and the situation becomes the same as case (1). In case (3), there is no reversible path ending at v, and because of $\overline{r}_v \leq \overline{r}_u$ and Lemma 1, there is also no reversible path ending at other vertices.

Next, we prove that Insert correctly maintains the IDNs of vertices. As the output $\vec{G}$ is an egalitarian orientation, we can directly analyze the indegree and reachability of vertices. According to Theorem 8, only vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$ need to be considered. Based on the proof of egalitarian orientation, it can be deduced that in the updated $\vec{G}$, vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$ cannot reach the vertices in $R_{\overline{r}_v+1}$ for cases (1) and (2). Thus, the indegree of vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$ does not increase to $\overline{r}_v + 1$, meaning that the IDNs remain $\overline{r}_v$. In case (3), only the indegree of vertex v in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$ increases to $\overline{r}_v + 1$. Therefore, the vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$ that can reach vertex v, as well as vertex v itself, have their IDNs increased to $\overline{r}_v + 1$. □

The following example illustrate how Insert works.

Example 4. *Suppose the graph in Fig. 1b is the input egalitarian orientation of* Insert. *For the three aforementioned cases: (1) When* $d_v = \overline{r}_v - 1$, *consider the example of inserting* (v_3, v_6). *In this case, the algorithm only needs to directly insert* $\langle v_3, v_6 \rangle$ *or* $\langle v_6, v_3 \rangle$. *(2) When* $d_v = \overline{r}_v$ *and there exists a path* $s \rightsquigarrow v$, *an example is inserting* (v_5, v_{12}). *Since* $\overline{r}_{v_5} \geq \overline{r}_{v_{12}}$, *the algorithm first inserts* $\langle v_5, v_{12} \rangle$ *directly. Then it identifies a reversible path* $v_{15} \rightarrow v_{14} \rightarrow v_{13} \rightarrow v_{12}$, *and reverse this path. (3) When* $d_v = \overline{r}_v$ *and there is no path* $s \rightsquigarrow v$, *consider the example of inserting* (v_6, v_{10}). *The algorithm inserts* $\langle v_6, v_{10} \rangle$ *into* $\vec{G}$. *However, it cannot find a reversible path, and* $\vec{G}$ *remains egalitarian. Since the indegree of* v_{10} *increases to* $\overline{r}_{v_{10}} + 1$, *the IDNs of* v_{10} *and other vertices that have the same IDN and are reachable to* v_{10} *should be incremented by one. Therefore, the IDNs of* $\{v_8, \ldots, v_{11}\}$ *are equal to 3 after inserting* (v_6, v_{10}).

Below, we analyze the time complexity of Algorithm 5 as follows.

Theorem 10. *The worst-case time complexity of Algorithm 5 is* $O(\sum_{u \in R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}} d_u(G))$.

Proof. Finding the path $s \rightsquigarrow v$ and obtaining the set $\{w | \overline{r}_w = \overline{r}_v$ can reach $v\}$ both can be accomplished using the reverse Breadth-First Search (BFS) algorithm once, and it only needs to search for vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. Therefore, the time complexity of Insert is $O(\sum_{u \in R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}} d_u(G)) \subseteq O(m)$. □

As shown in Theorem 10, the time complexity of Algorithm 5 is clearly bounded by $O(m)$. Furthermore, its complexity only depends on the size of the subgraph where the vertices' IDNs might be updated, indicating that the algorithm is local and only needs to explore a small portion of the graph. Indeed, as shown in our experiments, Algorithm 5 can be 5-7 orders of magnitude faster than the baseline algorithm that relies on re-computation.

5.3 The Proposed Deletion Algorithm

In this subsection, we proposed the Delete algorithm to maintain the density decomposition for handling edge deletion. The detailed implementation of Delete is outlined in Algorithm 6. Suppose that the deleted edge (u, v) is oriented as $\langle u, v \rangle$ (Line 1). According to Lemma 1, the inequality $\overline{r}_v \leq \overline{r}_u$ holds. The Delete algorithm addresses two cases for maintaining the egalitarian orientation: (1) $d_v = \overline{r}_v$ (not satisfying the 'if' condition of Line 2); (2) $d_v = \overline{r}_v - 1$ (satisfying the 'if' condition). In case (1), Delete directly deletes $\langle u, v \rangle$ (Line 5), and $\vec{G}$ is still an egalitarian orientation. In case (2), by the definition of density decomposition, a reversible path $v \rightsquigarrow t$ must exist (Line 3). This ensures that Delete does not encounter the scenario of being unable to find a reversible path, as opposed to

Algorithm 6: Delete($\vec{G}, \overline{r}, (u, v)$)

Input: The egalitarian orientation $\vec{G}$, the IDNs of all vertices $\overline{r}$, and the edge (u, v) need to be deleted.

Output: The updated egalitarian orientation $\vec{G}$ and IDNs $\overline{r}$.

1 Suppose (u, v) is oriented as $\langle u, v\rangle$, otherwise swap the input edge (u, v);
2 **if** $d_v = \overline{r}_v - 1$ **then**
3 There must be a reversible path $v \rightsquigarrow t$, where $d_t = \overline{r}_v$;
4 Reverse the path;
5 $\vec{G} \leftarrow \vec{G} - \langle u, v\rangle$;
6 Let $\overline{r}_0 \leftarrow \overline{r}_v$;
7 Let $S \leftarrow \{w | \overline{r}_w = \overline{r}_0$, and $d_w = \overline{r}_0$ or w can each an $\overline{r}_0$-indegree vertex$\}$;
8 **forall** w *where* $\overline{r}_w = \overline{r}_0$ *and* $w \notin S$ **do**
9 $\overline{r}_w \leftarrow \overline{r}_0 - 1$;
10 **return** $(\vec{G}, \overline{r})$;

Insert. After reversing $v \rightsquigarrow t$, case (2) transforms into case (1). Then, $\langle u, v\rangle$ can be deleted and $\vec{G}$ is restored to the egalitarian orientation.

With the egalitarian orientation, the Delete algorithm starts maintaining the IDNs of vertices. Unfortunately, unlike Insert, which can update IDNs directly, there is no direct method to update IDNs in the deletion case. Therefore, Delete needs to perform Lines 6-9 to update IDNs. Since $\overline{r}_v$ may change, to avoid ambiguity, we use $\overline{r}_0$ to denote the value of $\overline{r}_v$ before updating (Line 6). By the definition of density decomposition, Delete employs a single BFS algorithm to determine the set of vertices S that do not require IDNs updates (Line 7). It then decreases the IDNs of vertices that were originally equal to $\overline{r}_0$ but are not in the set S (Lines 8-9). Finally, the Delete algorithm terminates, outputting the egalitarian orientation and the IDNs of all vertices. The following theorem shows the correctness of Algorithm 6.

Theorem 11. *Algorithm 6 correctly updates the egalitarian orientation and the IDNs of all vertices.*

Proof. To prove the theorem, we need to consider two cases: (1) $d_v = \overline{r}_v$; (2) $d_v = \overline{r}_v - 1$. We first prove that Delete correctly updates the egalitarian orientation. In case (1), Delete directly deletes $\langle u, v\rangle$. For vertex v, before the edge deletion, it is impossible for v to reach any vertex with an indegree no less than $\overline{r}_v + 1$. Therefore, after deleting the edge, a reversible path starting from v cannot exist. For other vertices, the deletion operation does not expand their reachability to other vertices, thus no reversible path from them appears. For case (2), we can easily derive that all vertices in the path $v \rightsquigarrow t$ are also in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. Therefore, after reversing $v \rightsquigarrow t$, $\vec{G}$ remains an egalitarian orientation, and case (2) becomes (1). Thus, the proof is established. Then, we show the correctness of Delete for updating IDNs which is straightforward. By Theorem 8, only the vertices having IDNs of $\overline{r}_0$ need to be updated, while the set S contains all vertices whose IDNs remain unchanged. Therefore, the IDNs of vertices can be correctly updated by Delete. □

The following example illustrates how Algorithm 6 works.

Example 5. *Consider the graph in Fig. 1b. When deleting the edge (v_1, v_8), we have $d_v = \overline{r}_v$. The* Delete *algorithm directly removes $\langle v_1, v_8\rangle$. Then, it computes the set S to ensure no IDN should be updated. While in the case of deleting (v_1, v_2), due to $d_v = \overline{r}_v - 1$, the* Delete *algorithm finds the path $v_2 \rightarrow v_5$ and reverses it. After that,* Delete *confirms that the IDNs of vertices $v_1, \ldots, v_7$, which all do not belong to $S = \emptyset$, need to be decreased, so their IDNs are decreased to 2.*

Theorem 12. *The worst-case time complexity of Algorithm 6 is $O(\sum_{u \in R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}} d_u(G))$.*

Proof. Similar to Insert, all computation in Delete can be carried out within $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. Furthermore, the search for S can be done by performing the BFS algorithm starting from all vertices with indegree equal to $\overline{r}_0$ in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. Therefore, the time complexity of the Delete algorithm is the same as Insert, which is $O(\sum_{u \in R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}} d_u(G)) \subseteq O(m)$. □

Similar to the Insert algorithm, the Delete algorithm is also local, and the worst-case time complexity is clearly bounded by $O(m)$. Our experiments also show that Delete is very efficient and can be 3-5 orders of magnitude faster than the baseline which is based on re-computation.

5.4 An Improved Deletion Algorithm

Despite exhibiting the same time complexity, in our experiments, the Delete algorithm runs around three orders of magnitude slower than Insert. The main issue is, to update the IDNs of vertices, Insert performs BFS only within $\{w | \overline{r}_w = \overline{r}_v$ and w can reach $v\} \subseteq R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$, whereas Delete necessitates considering the entire $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. In this subsection, we devise an improved algorithm, called Delete++ (Algorithm 7), to further enhance the performance of the deletion algorithm.

The rationale and intuition behind Delete++ are as follows. If the IDN of a vertex w decreases due to the deletion of $\langle u, v\rangle$, by Definition 3, w must have been capable of reaching v prior to the deletion. Otherwise, w would have possessed the ability to reach another vertex with an $\overline{r}_v$-indegree distinct from v before the deletion, thereby preventing its IDN from decreasing. Therefore, we can calculate the set $P \subseteq R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$, which includes the vertices capable of reaching v (Line 2). The vertices where the IDNs may be reduced are specifically within the set P, allowing us to narrow down the scope for updating the IDNs to P rather than $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$.

Subsequently, we turn our focus to updating the IDNs of the vertices in P. Recall that the Delete algorithm identifies all vertices capable of reaching the vertices with $\overline{r}_v$-indegree, and then reduces the IDNs of vertices rather than these vertices. Such an approach, however, cannot be directly applied to update the IDNs within P. This is because, if a vertex in P cannot reach $\overline{r}_v$-indegree vertices within P, it does not necessarily imply that its IDN needs to be decreased since it might still have the possibility of reaching $\overline{r}_v$-indegree vertices outside of P.

A basic idea is performing a BFS individually on each vertex in P to determine its reachability to vertices with an indegree of $\overline{r}_v$. But it is impractical due to massive redundant re-computations. The key to solving this problem lies in reducing the number of BFS operations instead of "individually" searching for each vertex. To achieve this, we leverage a classic concept of Strongly Connected Component [60] (SCC). Once whether a vertex s can reach $\overline{r}_v$-indegree vertices is confirmed, vertices within the same SCC as s also share the same reachability, eliminating the need to invoke the BFS algorithm on these vertices again.

In the following, we show the correctness of the Delete++ algorithm. Since Line 1 and Line 3 of Delete++ is the same as the method for maintaining egalitarian orientation in Delete, Delete++ can correctly update egalitarian orientation. Next, we demonstrate that Delete++ can correctly update the IDNs of vertices.

Theorem 13. *Algorithm 7 correctly updates the IDNs of vertices.*

Proof. First, an important observation is that in an egalitarian orientation $\vec{G}$, vertices within the same SCC share identical IDNs. According to the definition of density decomposition, the IDN of a vertex depends on its reachability in the egalitarian orientation.

Algorithm 7: Delete++($\vec{G}, \overline{r}, (u, v)$)

Input: The egalitarian orientation $\vec{G}$, the IDNs of all vertices $\overline{r}$, and the edge (u, v) to be deleted.

Output: The updated egalitarian orientation $\vec{G}$ and IDNs $\overline{r}$.

1 Perform Lines 1-4 of Delete;
2 Let $P \leftarrow \{w | w \text{ can reach } v\} \cap (R_{\overline{r}_v} \setminus R_{\overline{r}_v+1})$;
3 $\vec{G} \leftarrow \vec{G} - \langle u, v \rangle$;
4 **foreach** $s \in P$ *and* $d_s = \overline{r}_v - 1$ **do**
5 **if** *the SCC containing* s *has not been computed* **then**
6 Let $S \leftarrow \{s\} \cup \{w | s \text{ can reach } w\} \cap (R_{\overline{r}_v} \setminus R_{\overline{r}_v+1})$;
7 Compute all SCCs in the set S, while recording whether they can reach a $\overline{r}_v$-indegree vertex;
8 **if** *the SCC containing* s *cannot reach a vertex with* $\overline{r}_v$ *indegree* **then**
9 $\overline{r}_s \leftarrow \overline{r}_s - 1$;
10 **return** $(\vec{G}, r)$;

Given that vertices within the same SCC share an identical set of reachable vertices, their IDNs are identical.

In accordance with Theorem 8, the necessity to update IDNs is confined to vertices in $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$. In Line 2 of Algorithm 7, the vertices with $(\overline{r}_v - 1)$-indegree in the set $(R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}) \setminus P$ are guaranteed to reach another vertex having an $\overline{r}_v$-indegree distinct from v. It further follows that the deletion of $\langle u, v \rangle$ will not affect their IDNs. Thus, the key consideration lies in determining whether there is a necessity to reduce the IDNs of $(\overline{r}_v - 1)$-indegree vertices in set P. In the 'foreach' loop, consider each $(\overline{r}_v - 1)$-indegree vertex s in P one by one. By Definition 3, the IDN of vertex s ought to decrease if and only if there are no vertices with an $\overline{r}_v$-indegree in S. This condition is equivalent to the SCC containing s being unable to reach a $\overline{r}_v$-indegree vertex. Therefore, the IDN of vertex s can be updated correctly. □

We employ the linear-time Tarjan algorithm [60] to compute SCCs, and thus the complexity of Delete++ is equivalent to that of Delete. However, in contrast to Delete, which accounts for the entire set $R_{\overline{r}_v} \setminus R_{\overline{r}_v+1}$, Delete++ focuses solely on the neighborhood P of vertex v. Consequently, the actual running time of Delete++ proves notably faster than that of Delete in our experiments.

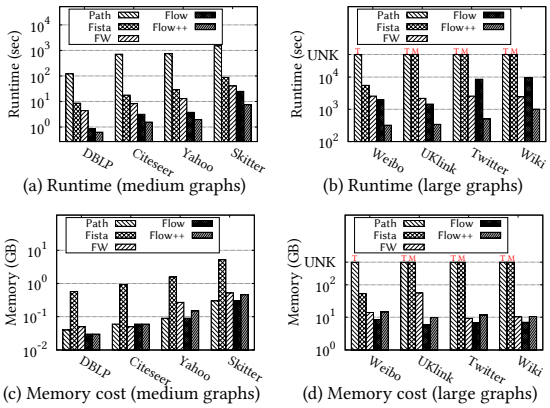
5.5 Discussions

As shown in Theorems 10 and 11, both our insertion and deletion algorithms only need to traverse a small portion of the graph and their time complexity can clearly be bounded by $O(m)$, revealing the advantage of relatively-easy maintenance of density decomposition. In contrast, LDS decomposition appears to be challenging to maintain. To our knowledge, there is not an efficient algorithm to exactly maintain the LDS decomposition, except for re-computation from scratch. The potential reasons are summarized as follows.

First, for the LDS decomposition, we find that the insertion or deletion of an edge may generate many new layers, resulting in high updating costs. For example, consider a graph G consists of three subgraphs, A, B, and C, where A is connected to both B and C, but B and C are not interconnected. Assume that $\rho(A) < \rho(B, A) < \rho(C, A)$, and the entire graph $A \cup B \cup C$ forms the densest subgraph. Now, we insert an edge within A, leading to $\rho(B, A) < \rho(C, A) < \rho(A)$, and A becomes the new densest subgraph. Given that B and C are not connected to each other, A, B, and C are assigned to different layers in the LDS decomposition, indicating the division of one layer into three layers. As a result, to maintain it, we need to identify all these divided layers, which is clearly costly, because identifying any one layer of the LDS decomposition may need to invoke the parameterized network flow technique once [34].

Table 1: Datasets statistics

Name	Type	n	m
DBLP	co-authorship network	317,081	1,049,866
Citeseer	citation network	384,414	1,736,145
Yahoo	lexical network	653,261	2,931,698
Skitter	internet	1,694,617	11,094,209
Weibo	social network	58,655,850	261,321,033
UKlink	web graph	18,483,187	261,787,258
Twitter	social network	20,826,113	294,585,816
Wiki	web graph	13,593,033	334,591,525



(a) Runtime (medium graphs) (b) Runtime (large graphs)
(c) Memory cost (medium graphs) (d) Memory cost (large graphs)

Figure 3: Results of density decomposition on static graphs.

Second, whenever an edge is inserted or deleted, the outer density must be changed, and therefore the FDNs also need to be updated. In contrast, each IDN is a round-up value of FDN, thus it is updated infrequently. Indeed, updating a single edge in a graph certainly changes the FDNs, but it typically has a small impact on the round-up values of FDNs (i.e., IDNs). For example, as shown in the Insert algorithm, IDNs only needs to change during insertion when $d_v = \overline{r}_v$ and no reversible path can be found, further confirming that IDNs changes are infrequent. This result suggests that maintaining the LDS decomposition is often much more difficult than maintaining the density decomposition, further highlighting the superiority of density decomposition.

6 EXPERIMENTS

Different algorithms. For static graphs, we compare our proposed algorithms Flow (Algorithm 3) and Flow++ (Algorithm 4) with three baselines: Path (Algorithm 1) [14], FW [26], and Fista [35]. Here, Flow, Flow++, and Path are implemented by us. FW [26] and Fista [35] are the state-of-the-art algorithms for LDS decomposition. By our results established in Section 3, these algorithms can also be used to compute the density decomposition, thus we also include them as baselines. For dynamic graphs, we implement three proposed algorithms Insert (Algorithm 5), Delete (Algorithm 6), and Delete++ (Algorithm 7). Since there is no algorithm that can maintain density decomposition, we use the Flow++ algorithm for re-computing the density decomposition as a baseline.

All algorithms are implemented using C++ and compiled with the GCC compiler, employing the O3 optimization. All experiments are conducted on a PC operating a Linux system, equipped with a 2.2GHz AMD 3990X 64-Core CPU and 128GB memory.

Datasets. We use 4 medium graphs and 4 large graphs in our experiments which are downloaded from the Network Repository [50] and the Koblenz Network Collection (http://konect.cc/). The detailed information is provided in Table 1.

Table 2: Number of layers of different decompositions

"DD" denotes density decomposition.
"LDS" represents locally-densest subgraph decomposition.

Datasets	DD	LDS	Datasets	DD	LDS
DBLP	59	1,088	Weibo	168	5,609
Citeseer	16	1,435	UKlink	474	40,875
Yahoo	26	1,376	Twitter	840	11,949
Skitter	92	3,493	Wiki	602	23,203

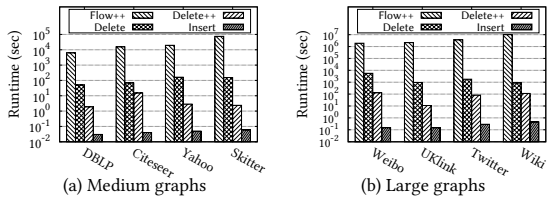
6.1 Performance Studies

Exp-1: Runtime of various density decomposition algorithms. Fig. 3a and Fig. 3b depict the results of different algorithms for computing the density decomposition on medium graphs and large graphs. In Fig. 3b, there are two cases designated as 'UNK' (unknown) for runtime: (1) the algorithm's runtime exceeds our time constraint of 50,000 seconds, labeled as 'T'; (2) the algorithm's memory usage exceeds 128GB, labeled as 'M'. From Fig. 3a and Fig. 3b, Path unsurprisingly exhibits the longest execution time, and even fails to complete the computation within the time constraint on all large graphs. In contrast, our proposed algorithms, Flow and Flow++, not only successfully compute the density decomposition for all datasets but are also at least 1 and 2 orders of magnitude faster than Path respectively, exhibiting significant superiority. For instance, on Citeseer, the algorithms Path, Flow, and Flow++ consume 712.02, 2.5, and 1.56 seconds, respectively, indicating that Flow and Flow++ can be 284x and 456x faster than Path. In addition, Flow++ shows remarkable enhancements, particularly on large graphs, where efficiency is improved by about an order of magnitude compared to Flow. These results confirm our theoretical analysis in Section 3.

When comparing the two LDS decomposition algorithms, we find that FW outperforms Fista over all datasets. The reason could be that Fista was originally designed as an approximate LDS decomposition algorithm, relying on a peeling technique [35], and it may not be well-suited for exact LDS decomposition. Compared to the LDS decomposition algorithms, our best algorithm Flow++ is consistently faster on all datasets, and can still achieve one order of magnitude speedup on large datasets. For instance, on Weibo, the runtime of Flow++ and FW are 190.42 seconds and 2,575.96 seconds respectively, representing a 13x acceleration. This result further demonstrates the high efficiency of our Flow++ algorithm.

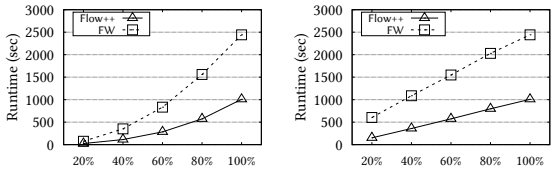
Exp-2: Memory overheads of different algorithms. The memory usages of different algorithms are shown in Fig. 3c and Fig. 3d. Both the Path and Flow algorithms have lower memory costs, which are consistent with the theoretical analysis of the space complexity as $O(m)$. The algorithms Flow++, FW, and Fista all involve a recursive procedure, which requires auxiliary space usage. Among them, Flow++ needs to store each layer R_k, and since the number of layers is small as shown in Table 2, the memory cost of Flow++ remains modest. On all datasets, Flow++ requires no more than 15GB of memory. In the case of FW and Fista, their memory costs depend to a large extent on the effectiveness of the approximation algorithms that they employ. The approximation algorithm in FW is more effective, thus it often consumes less memory than Fista. Nevertheless, on UKlink, FW algorithm uses significantly higher memory compared to Flow and Flow++, indicating lower robustness in terms of memory cost for the FW algorithm. These results confirm that the memory usage of our algorithms is acceptable with efficient computational performance.

Exp-3: Number of layers of different decompositions. The number of layers of density decomposition and LDS decomposition are presented in Table 2. For density decomposition, the number of layers is equal to the pseudoarboricity plus 2, which is typically small in real-world graphs [40]. However, the LDS decomposition yields a considerably larger number of layers, reaching 40,875 layers on UKlink. This multitude of layers often leads to high computation time. Moreover, such a fine partition of layers may inevitably lead to unnecessary separation of tightly connected subgraphs, thus reducing the performance of identifying dense subgraphs for practical applications. These results confirm our analysis in Sections 1 and 3.



(a) Medium graphs (b) Large graphs

Figure 4: Runtime of various maintenance algorithms



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

Figure 5: Scalability test results on Wiki**.**

Exp-4: Runtime of the various maintenance algorithms. We randomly delete and insert 10,000 edges from each graph dataset to evaluate different maintenance algorithms. The total time required to process these 10,000 deleted or inserted edges is shown in Fig. 4. For the baseline algorithm, since both insertion and deletion require invoking Flow++ to recompute, the runtime for deletion and insertion is the same, thus we only plot one bar in Fig. 4. As can be seen, both Delete and Delete++ are around 3-5 orders of magnitude faster than the baseline algorithm (Flow++), respectively. Furthermore, Delete++ significantly outperforms Delete on all datasets as expected. For example, on UKlink, Delete consumes 943.62 seconds to maintain the density decomposition, while Delete++ only takes 10.98 seconds, achieving an 85x acceleration. These results confirm our analysis in Section 5. For edge insertion, Insert is substantially faster than Flow++, improving by about 5-7 orders of magnitude. When applied to large graphs such as Wiki with hundreds of millions of edges, handling the insertion of 10,000 edges only requires 0.4 seconds. These results further demonstrate the high efficiency of our algorithm. Additionally, it is worth noting that the runtime of our insertion algorithm is often orders of magnitude faster than that of the deletion algorithms. This significant efficiency stems from the fact that (as analyzed in Section 5) updating the IDNs during insertion is much easier compared to the deletion case.

Exp-5: Scalability test. In this experiment, we evaluate the scalability of Flow++ and FW, which are the best algorithms for density decomposition and LDS decomposition respectively. We randomly sample the vertex set V and edge set E to generate 8 subgraphs for each dataset to conduct the scalability test. The results on the Wiki dataset are shown in Fig. 5, and similar results can also be observed on the other datasets. In Fig. 5, we can see that Flow++ outperforms FW at all data scales, consistent with previous observations. Furthermore, the runtime of FW exhibits a rapid increase as the data scale grows, whereas the runtime of Flow++ increases more smoothly. These results demonstrate the high scalability of the Flow++ algorithm.

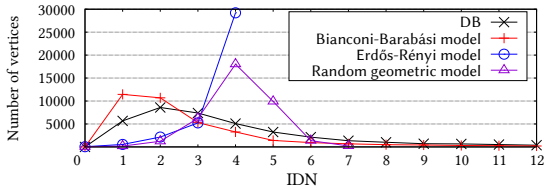


Figure 6: The IDN distribution of DB.

Exp-6: The IDN distribution. We use a subgraph DB of DBLP, consisting of 37,177 vertices and 131,715 edges, to illustrate the IDN distribution. DB is constructed from the DBLP by extracting authors who have published at least one paper in a conference related to databases and data mining, along with the collaborations between them. For comparison, various synthetic graphs are meticulously generated to match DB's scale in terms of vertices and edges using different models. Specifically, the models include Erdős–Rényi model (ER) [29], power-law Bianconi –Barabási model (BB) [9], random geometric model (RG) [30], power-law Barabási–Albert model (BA) [2], and small-world Watts–Strogatz model (WS) [65]. The IDN distributions of DB and these synthetic graphs are depicted in Fig. 6, and the results on other datasets are consistent. Given that the pseudoarboricity of DB and the BB graph are 19 and 40, we only draw distributions with IDNs no larger than 12 in Fig. 6 for clarity. When IDN exceeds 12, their distributions still exhibit a gradual decline, ultimately converging to zero. In addition, in the BA graph and the WS graph, all vertices share the same IDNs. This phenomenon is not explicitly depicted in Fig. 6 and will be elucidated later.

As shown in Fig. 6, similar to the degree distribution, the IDN distribution of DB is highly uneven. The number of vertices is higher when IDN is not greater than 2 but exhibits a declining trend when IDN exceeds 2. This suggests that represented by DB, real-world graphs are typically sparse with small dense regions. For the ER graph, which has a lower pseudoarboricity of 4, most vertices cluster at the top layer of the density decomposition, and its IDN distribution monotonically increases. This differs significantly from real-world graphs that adhere to the power-law principle. In the case of the BB graph, it can be observed that the IDN distribution closely resembles that of DB. However, its pseudoarboricity is 40, significantly higher than DB's pseudoarboricity 19. Thus, the graph generated by this model can effectively reflect the density structure of real-world graphs, but its densest region could be excessively dense. Regarding the RG graph, the IDN distribution also exhibits a trend of initially increasing and then decreasing. But its pseudoarboricity is only 7, and more than half of the vertices have IDNs greater than half of the pseudoarboricity. This indicates that most vertices are located in denser regions within the RG graph, which does not align with real-world graphs. For both the BA graph and WS graph, all vertices have the same IDN of 4. This is because for almost all subgraph $H \subsetneq G$, $\rho(H, G) \approx 4$ holds, resulting in a highly uniform density throughout the graph. These results indicate that the BB graph can relatively better match the density characteristics of real-world graphs.

6.2 Case Studies

Harry Potter characters relationship network. Here we conduct a case study on the Harry Potter characters network[1], denoted by HPC. We evaluate the effectiveness of three graph decompositions, including density decomposition, LDS decomposition, and core decomposition. The IDNs, FDNs, and core numbers of all characters in HPC are illustrated in Fig. 8, with vertex sizes corresponding to their degrees. From Fig. 7a and Fig. 7b, we can see that the density decomposition and LDS decomposition on HPC exhibit a high similarity, with the only distinction being that in the LDS decomposition, Quirinus Quirrell, one of the four characters with an IDN of 4, is separated from the other three characters. This result indicates that the density decomposition can well approximate the LDS decomposition. Moreover, in this case study, there seems no need to excessively differentiate the characters in fine detail. Whether a character with FDN of 4.00 or 3.33 can both be considered as similar important secondary characters. This result suggests that a moderately coarse-grained partition achieved by density decomposition often suffices, eliminating the need for an excessively fine-grained partition. Additionally, decomposing the graph into finer layers generally requires more computational time. On HPC, invoking Flow++ for density decomposition takes 0.048ms, while performing FW to compute LDS decomposition consumes 0.918ms. In summary, density decomposition can achieve similar results as the LDS decomposition, while significantly reducing computational time and the number of redundant dense subgraph layers.

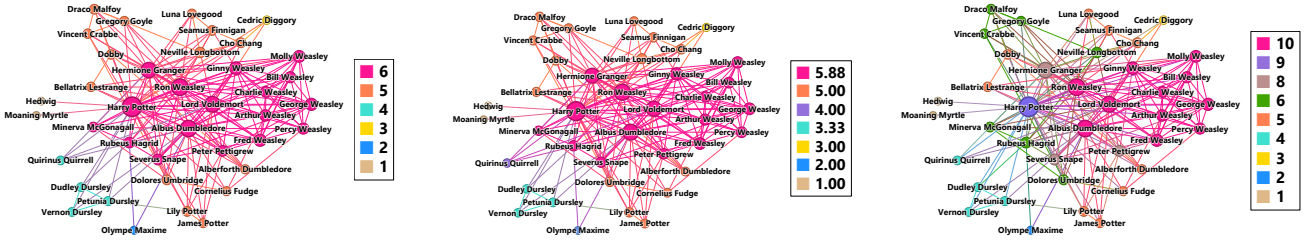
When comparing the density decomposition (Fig. 7a) and the core decomposition (Fig. 7c), a noticeable disparity emerges. The most notable difference is that the top layer of density decomposition is divided into four layers in the core decomposition. The main character, Harry Potter, is not included in the top layer (i.e., 10-core), which also exhibits a clearly lower density in comparison to the other two density-aware decomposition methods. These results confirm our analysis in Sections 1 and 3.

DBLP collaboration subgraph. We also perform a case study on the DB dataset (with 37,177 vertices and 131,715 edges), which is a subgraph of the DBLP co-authorship network that only contains database researchers and their co-authorships. We depict DB's four subgraphs $C_{35} \subseteq R_{18} \subseteq C_{18}$ and R_{19} as shown in Fig. 8a. The full version of the figure, with the author represented by each vertex, is available at https://github.com/Flydragonet/Density_Decomposition_Computation.

As shown by our proposed Sandwich Theorem (Theorem 4), C_{35}, R_{18}, and C_{18} show a hierarchical structure, with their numbers of vertices and edges being $|C_{35}| = 72$, $|E(C_{35})| = 1,260$, $|R_{18}| = 131$, $|E(R_{18})| = 2,339$, $|C_{18}| = 529$, and $|E(C_{18})| = 7,398$. Compared to the original graph with $|V| = 37,177 \gg |C_{18}|$ vertices, R_{18} is tightly sandwiched between C_{35} and C_{18}. Furthermore, we illustrate the subgraph R_{19} with the dashed line in Fig. 8a. It can be observed that R_{18} and R_{19} are adjacent layers, but $C_{35} \subseteq R_{18}$ and $C_{35} \nsubseteq R_{19}$, indicating the tightness of $C_i \subseteq R_{\lceil i/2 \rceil}$ in the Sandwich Theorem. Moreover, within DB, C_{35} and R_{19} serve as the top non-empty layer for core decomposition and density decomposition, respectively. It can be seen that C_{35} fails to separate two loosely-connected communities, whereas R_{19} contains a densely-connected community. Their respective densities are $\rho(C_{35}) = 17.5 < \rho(R_{19}) = 18.3$, and R_{19} is the densest subgraph. This again highlights the weakness of the core decomposition in capturing dense structures, particularly when contrasted with the density-aware decomposition.

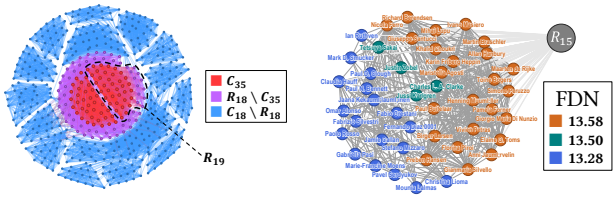
To further confirm the unnecessary fine separation generated by LDS decomposition, we present the subgraph R_{14} of DB in Fig. 8b, using a single large vertex to represent R_{15} for clarity. In density decomposition, vertices in $R_{14} \setminus R_{15}$ are grouped into the same layer, while LDS decomposition divides them into three different layers with FDNs of 13.58, 13.50, and 13.28, respectively. These three layers have nearly the same outer density and form a densely connected community, suggesting that they should not be partitioned into three distinct layers. Furthermore, the four vertices with FDNs of

[1]Data source: https://github.com/efekarakus/potter-network, from which we extract a subgraph containing only main characters and main relationships for clarity.



(a) IDN (density decomposition) (b) FDN (LDS decomposition) (c) Core number (core decomposition)

Figure 7: Dense subgraph decompositions of Harry Potter character relationship network HPC.



(a) C_{18}, R_{18}, C_{35} and R_{19} (b) R_{14}

Figure 8: Dense subgraphs of decompositions of DB **network.**

13.50 are connected with each other by only four edges, indicating a loose connection. It is evident that there is no need to separate these four vertices into an individual layer. All these observations suggest that LDS decomposition may overly separate the dense subgraph layers, while density decomposition, which places $R_{14} \setminus R_{15}$ in the same layer, is more reasonable.

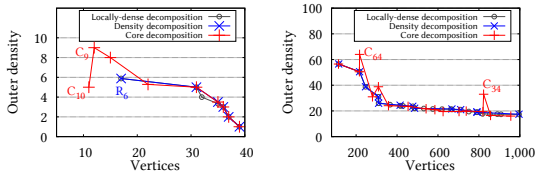
Outer density comparison. In this case study, we evaluate the ability of core decomposition, LDS decomposition, and density decomposition to reflect the density of subgraphs. For measuring the density of a specific layer R_k in a decomposition, utilizing the outer density $\rho(R_k, R_{k+1})$ is considered superior to the alternatives $\rho(R_k \setminus R_{k+1})$ or $\rho(R_k)$. This is because: (1) the computation of $\rho(R_k \setminus R_{k+1})$ disregards the contributions of $E_\times(R_k, R_{k+1})$, which represents the closeness of the connection between R_k and R_{k+1}. Contrarily, $\rho(R_k, R_{k+1})$ incorporates these edges into the computation; (2) $\rho(R_k)$ considers the entirety of R_k instead of $R_k \setminus R_{k+1}$. However, our goal is to measure the density of R_k while excluding the influence of R_{k+1}. When k is small, $\rho(R_k)$ typically fails to highlight the impact of $R_k \setminus R_{k+1}$ on the density of R_k.

Fig. 9 depicts the outer densities of subgraphs of different decompositions on HPC and DBLP. In Fig. 9, each point along the horizontal axis is the number of vertices contained in a layer, and the vertical axis is the outer density of that layer. For example, the point of 'C_9' in Fig. 9a represents the 9-core in HPC, and its horizontal and vertical axes are $|C_9|$ and $\rho(C_9, C_{10})$ respectively. From Fig. 9a, we can see that on HPC, the line chart for core decomposition starts from 5 and goes up to 9 initially because of $\rho(C_{10}, \emptyset) = 5$ and $\rho(C_9, C_{10}) = 9$. This increase indicates that the lower layer (i.e., C_9) is even denser than the higher layer (i.e., C_{10}), suggesting that core decomposition does not perform well in extracting high-density subgraphs. In contrast, density decomposition guarantees a monotonically decreasing trend in the outer density of each layer, ensuring that higher layers are denser than lower ones. Similar results are presented for DBLP, as shown in Fig. 9b, where, for clarity, only the layers with fewer than 1,000 vertices are plotted. Clearly, the outer densities of C_{64} and C_{34} are higher than those of C_{65} and C_{35}, again indicating that core decomposition cannot stably obtain denser graphs. For density decomposition and LDS decomposition, as expected, their line charts overlap perfectly. This confirms that density decomposition can well approximate LDS decomposition while greatly reducing the number of redundant layers.

7 RELATED WORK

Densest subgraph discovery. Our work is close to the problem of finding the densest subgraph in a graph. A well-known algorithm for computing the densest subgraph is based on a parameterized network flow technique [34] which needs to invoke the maximum flow algorithm for $O(\log n)$ times. To improve the efficiency, Danisch et al. proposed a convex programming approach to identify the densest subgraph which is considered as the state-of-the-art [26]. Several approximate algorithms have also been devised to compute the densest subgraph, such as the linear-time 2-approximation algorithm [17] $O(m+n)$ and the $(1+\epsilon)$-approximation iterative algorithm [12, 18], with near-linear time for each iteration. Recently, the techniques for finding the densest subgraph were also extended to the higher-order case [36, 59, 62, 66] as well as the directed graph case [43, 44]. Additionally, there are several approximation algorithms available for maintaining the densest subgraph [8, 47, 56], but none of them can achieve exact maintenance of the densest subgraph. Moreover, all the techniques mentioned above are mainly designed for solving the densest subgraph problem and cannot be directly applied to compute the density decomposition.

Cohesive subgraph decomposition. Except for the density decomposition, there also exist several other cohesive subgraph decompositions that aim to decompose a graph into a set of nested cohesive subgraphs. Notable examples include the k-core decomposition [4, 41, 52], k-truss decomposition [22, 37, 64], nucleus decomposition [53–55, 58], k-edge connected subgraph decomposition [15, 16], locally densest subgraph (LDS) decomposition [49, 61], distance-generalized core decomposition [11, 25] and colorful h-star k-core decomposition [32, 33]. The core decomposition [4, 41, 52] and their variants [11, 25, 32, 33] is a degree-aware decomposition which iteratively decomposes the graph based on the degrees of the vertices. The truss decomposition is a triangle-aware decomposition that is based on the number of triangles in which an edge participates [22, 37, 64]. The nucleus decomposition can be considered as a higher-order core or truss decomposition which is based on the number of cliques in which a smaller clique participates [53–55, 58]. The k-edge connected subgraph decomposition is a connectivity-aware decomposition that aims to decompose the graph into a set of highly connected subgraphs [15, 16]. The LDS decomposition, however, is a density-aware decomposition that can fully capture the density of the decomposed subgraphs [49, 61]. Note that the density decomposition studied in this paper is also a density-aware decomposition. In this paper, we reveal an interesting inclusion relationship between the density decomposition and LDS decomposition, and we also propose several novel and more efficient algorithms to compute the density decomposition on both large static and dynamic graphs.



(a) HPC (b) DBLP

Figure 9: Outer density of different decompositions.

8 CONCLUSION

In this paper, we conduct an in-depth investigation of density decomposition on both static and dynamic graphs. First, we establish an inclusion relationship between density decomposition and LDS decomposition, as well as the approximate relationship between density decomposition and core decomposition. For the computation of density decomposition, we develop the Flow++ algorithm using carefully-designed network flow and divide-and-conquer techniques, achieving a notable reduction in time complexity to $O(m^{3/2} \log p)$. To handle dynamic graphs, we prove a density decomposition update theorem, which shows that the maintenance of density decomposition is highly localized. Building upon this result, we develop the insertion and deletion algorithms, i.e., Insert, Delete, and Delete++, all with linear time complexity. Extensive experiments on 8 real-world datasets demonstrate the efficiency, scalability, and effectiveness of our solutions.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: a Truss-equivalence Based Indexing Approach. In *Proc. VLDB Endow.*, Vol. 10. 1298–1309.

[2] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of modern physics* 74, 1 (2002), 47.

[3] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*. 41–50.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[5] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. 1980. A general method for solving divide-and-conquer recurrences. *SIGACT News* 12, 3 (1980), 36–44.

[6] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. CopyCatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*. 119–130.

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

[8] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. 2015. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *STOC*. 173–182.

[9] Ginestra Bianconi and A-L Barabási. 2001. Competition and multiscaling in evolving networks. *Europhysics letters* 54, 4 (2001), 436.

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

[11] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.

[12] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos E. Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting Densest Subgraphs Without Flow Computations. In *WWW*. 573–583.

[13] Glencora Borradaile, Jennifer Iglesias, Theresa Migler, Antonio Ochoa, Gordon Wilfong, and Lisa Zhang. 2017. Egalitarian Graph Orientations. *Journal of Graph Algorithms and Applications* 21, 4 (2017), 687–708.

[14] Glencora Borradaile, Theresa Migler, and Gordon T. Wilfong. 2019. Density decompositions of networks. *J. Graph Algorithms Appl.* 23, 4 (2019), 625–651.

[15] Lijun Chang and Zhiyi Wang. 2022. A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition. In *Proc. VLDB Endow.*, Vol. 15. 1146–1158.

[16] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216.

[17] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX (Lecture Notes in Computer Science, Vol. 1913)*. 84–95.

[18] Chandra Chekuri, Kent Quanrud, and Manuel R. Torres. 2022. Densest Subgraph: Supermodularity, Iterative Peeling, and Flow. In *SODA*. SIAM, 1531–1555.

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

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

[21] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[22] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008), 1–29.

[23] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, and Guoren Wang. 2023. Maximal Defective Clique Enumeration. *Proc. ACM Manag. Data* 1, 1 (2023), 77:1–77:26.

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

[25] Qiangqiang Dai, Rong-Hua Li, Lu Qin, Guoren Wang, Weihua Yang, Zhiwei Zhang, and Ye Yuan. 2021. Scaling Up Distance-generalized Core Decomposition. In *CIKM*. 312–321.

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

[27] Luce R. Duncan and Perry Albert D. 1949. A method of matrix analysis of group structure. *Psychometrika* 14, 2 (1949), 95–116.

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

[29] Paul Erdős and Alfréd Rényi. 1959. On random graphs I. *Publ. math. debrecen* 6, 290-297 (1959), 18.

[30] Abraham D. Flaxman, Alan M. Frieze, and Juan Vera. 2007. A Geometric Preferential Attachment Model of Networks. *Internet Math.* 3, 2 (2007), 187–205.

[31] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. In *Proceedings 14th International Conference on Intelligent Systems for Molecular Biology*. 156–157.

[32] Sen Gao, Rong-Hua Li, Hongchao Qin, Hongzhi Chen, Ye Yuan, and Guoren Wang. 2022. Colorful h-star Core Decomposition. In *ICDE*. 2588–2601.

[33] Sen Gao, Hongchao Qin, Rong-Hua Li, and Bingsheng He. 2023. Parallel Colorful h-star Core Maintenance in Dynamic Graphs. In *Proc. VLDB Endow.*, Vol. 16. 2538–2550.

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

[35] Elfarouk Harb, Kent Quanrud, and Chandra Chekuri. 2022. Faster and scalable algorithms for densest subgraph and decomposition. *NeurIPS* 35 (2022), 26966–26979.

[36] Yizhang He, Kai Wang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2023. Scaling Up k-Clique Densest Subgraph Detection. *Proc. ACM Manag. Data* 1, 1 (2023), 69:1–69:26.

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

[38] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*. 813–826.

[39] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. 1999. Trawling the Web for Emerging Cyber-Communities. *Comput. Networks* 31, 11-16 (1999), 1481–1493.

[40] 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.

[41] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[42] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. Finding Locally Densest Subgraphs: A Convex Programming Approach. In *Proc. VLDB Endow.*, Vol. 15. 2719–2732.

[43] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *SIGMOD*. 845–859.

[44] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On Directed Densest Subgraph Discovery. *ACM Trans. Database Syst.* 46, 4 (2021), 13:1–13:45.

[45] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[46] Robert J Mokken et al. 1979. Cliques, clubs and clans. *Quality & Quantity* 13, 2 (1979), 161–173.

[47] Muhammad Anis Uddin Nasir, Aristides Gionis, Gianmarco De Francisci Morales, and Sarunas Girdzijauskas. 2017. Fully Dynamic Algorithm for Top-k Densest Subgraphs. In *CIKM*. 1817–1826.

[48] 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.

[49] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*. 965–974.

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

[51] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense Subgraphs with Restrictions and Applications to Gene Annotation Graphs. In *RECOMB (Lecture Notes in Computer Science, Vol. 6044)*, Bonnie Berger (Ed.). Springer, 456–472.

[52] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. In *Proc. VLDB Endow.*, Vol. 6. 433–444.

[53] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. In *Proc. VLDB Endow.*, Vol. 10. 97–108.

[54] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. In *Proc. VLDB Endow.*, Vol. 12. 43–56.

[55] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*. 927–937.

[56] Saurabh Sawlani and Junxing Wang. 2020. Near-optimal fully dynamic densest subgraph. In *STOC*. 181–193.

[57] Stephen B Seidman and Brian L Foster. 1978. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology* 6, 1 (1978), 139–154.

[58] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Theoretically and Practically Efficient Parallel Nucleus Decomposition. In *Proc. VLDB Endow.*, Vol. 15. 583–596.

[59] Bintao Sun, Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. In *Proc. VLDB Endow.*, Vol. 13. 1628–1640.

[60] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[61] Nikolaj Tatti. 2019. Density-Friendly Graph Decomposition. *ACM Trans. Knowl. Discov. Data* 13, 5 (2019), 54:1–54:29.

[62] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*. 1122–1132.

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

[64] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. In *Proc. VLDB Endow.*, Vol. 5. 812–823.

[65] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[66] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2023. Efficient and Effective Algorithms for Generalized Densest Subgraph Discovery. *Proc. ACM Manag. Data* 1, 2 (2023), 169:1–169:27.

[67] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k, r)-Core Computation on Social Networks. In *Proc. VLDB Endow.*, Vol. 10. 998–1009.

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