Scalable Algorithms for Densest Subgraph Discovery

Wensheng Luo[1], Zhuo Tang[1], Yixiang Fang[2], Chenhao Ma[2], Xu Zhou[1]

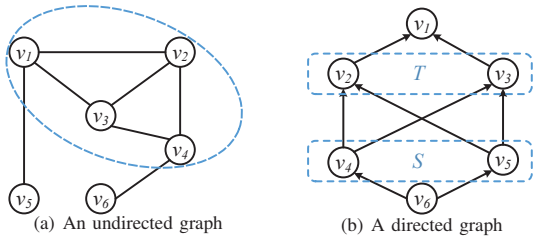
[1]College of Computer Science and Electronic Engineering, Hunan University, China
[2]School of Data Science, The Chinese University of Hong Kong, Shenzhen, China
Email:{luowensheng, ztang}@hnu.edu.cn, {fangyixiang, machenhao}@cuhk.edu.cn, zhxu@hnu.edu.cn

***Abstract*—As a fundamental problem in graph data mining, *Densest Subgraph Discovery (DSD)* aims to find the subgraph with the highest density from a graph. It has been studied for several decades and found a large number of real-world applications, such as network community detection, regulatory motif discovery in DNA, graph index construction, and fake follower detection. Although there are many existing DSD algorithms, they are often not scalable or efficient to process large-scale graphs, since most of them are serial algorithms and can only leverage the computing resource of a single CPU core. To tackle these issues, in this paper we propose efficient parallel algorithms for solving the DSD problems on both undirected and directed graphs at scale. Our main idea is to use the k-cores (a kind of dense subgraph) to approximate the densest subgraph in the undirected graphs, and then propose efficient parallel algorithms for computing the cores by optimizing the iterative process and also reducing the number of iterations. We further extend this idea for directed graphs by introducing a novel concept, named w-induced subgraph, to avoid unnecessary enumerations of x or y when searching $[x, y]$-cores (a kind of directed dense subgraph to approximate the densest). To verify the scalability and efficiency of the proposed algorithms, we have conducted extensive experiments on 12 large real-world graphs, and four of them are billion-scale. The experimental results show that our proposed algorithms outperform the state-of-the-art algorithms on both undirected and directed graphs, in terms of scalability and efficiency.**

I. Introduction

As a fundamental problem in graph data mining, *Densest Subgraph Discovery (DSD)* aims to find the subgraph with the highest density from a graph. It has been studied for several decades [1]–[9] and found a large number of real-world applications in various areas. For example, in network science area, DSD can be used for mining community [10]; in biology area, it can be leveraged to identify regulatory motifs in genomic DNA network [11], [12]; in web mining area, it can be applied to link spam detection [13]; in system optimization area, DSD has been used in social piggybacking for improving the throughput of social networking systems [14], [15]; in social media area, it is exploited for fake follower detection [7] and fraud detection [16], [17]; in graph database area, it is used to support many operations such as reachability query [18], and distance queries [19], graph visualization [20], [21], and large near-clique detection [22]. Actually, the exact solutions to the DSD problem are time-consuming, and the more efficient approximation approaches can provide high-quality results to meet the actual needs. Thus, approximation solutions are more widely used in these applications.

Most of the existing DSD works focus on undirected graphs and directed graphs, and they often use the number of edges "allocated" by each vertex to measure the density of a subgraph. More precisely, for an undirected graph G=(V, E), its density is measured by the number of edges over the number of vertices, i.e., $\rho(G)$=$\frac{|E|}{|V|}$. For example, in the undirected graph of Fig. 1(a), the density of the subgraph in the dashed ellipse is 5/4, since there are five edges and four vertices, and this subgraph is densest because its density is the highest among all possible subgraphs. For the directed graph D=(V, E), its density is defined over two vertex sets; that is, given two vertex sets (not necessarily disjoint) $S, T \in V$, the density of the subgraph induced by S and T is the number $|E(S,T)|$ of edges linking vertices from S to the vertices in T over the geometric mean of their sizes, i.e., $\rho(S,T)$=$\frac{|E(S,T)|}{\sqrt{|S||T|}}$. For instance, in the directed graph of Fig. 1(b), for the two vertex sets S=$\{v_4, v_5\}$ and T=$\{v_2, v_3\}$, the density of the directed subgraph induced by S and T is $\rho(S,T)$=$\frac{4}{\sqrt{2\times2}}$ = 2, since there are four edges linking from S to T, and this subgraph is the densest subgraph because no other two vertex sets are having a higher density. We can observe that when S = T, the density of the induced directed graph reduces to the above notion of the density of undirected graphs. In other words, the density of directed graphs naturally generalizes the notion of the density of undirected graphs [7].



(a) An undirected graph (b) A directed graph

Fig. 1: Illustrating DSD on undirected and directed graphs.

Prior works. Despite its importance, DSD is technically challenging on both undirected graphs and directed graphs. The exact DSD algorithms, which often involve solving the maximum flow problem or linear programming [1], [2], [6], [7], [9], [23], are very inefficient even on moderate-sized graphs [6], [7]. Thus, to achieve higher efficiency, many approximation algorithms are developed by trading accuracy with theoretical guarantee on the approximation ratio. Note that the approximation ratio is often defined as the density of the densest subgraph over the density of the subgraph found by an approximation algorithm, which is at least 1.0.

Specifically, for undirected graphs, Charikar [3] proposed the first 2-approximation algorithm, by iteratively peeling the vertex with the lowest degree and returning the subgraph with the highest density during this process. Bahmani et al. [5]

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00029

presented a parallel $2(1+\epsilon)$-approximation algorithm based on a streaming model (ϵ >0). Fang et al. [6] proved that the k-core with the maximum core number of the graph, also called k^*-core, is a 2-approximation solution, where a k-core is the largest subgraph in which each vertex's degree is at least k, and developed efficient algorithms. For directed graphs, Charikar [3] also proposed a 2-approximation algorithm following the peeling paradigm above. Khuller and Saha [4] developed an algorithm with linear time cost, but its approximation ratio is larger than 2 [7]. Bahmani et al. [5] presented a parallel $2(1+\epsilon)$-approximation algorithm based on a streaming model. Ma et al. [7], [9] introduced the $[x, y]$-core on the directed graph, or a subgraph in which the out-degrees of vertices in S are at least x and the in-degrees of vertices in T are at least y, and proved that the core with the largest $x \cdot y$ among all $[x, y]$-cores, i.e., $[x^*, y^*]$-core, offers a 2-approximation solution.

Although there are many existing DSD algorithms, they often scale poorly as the size of graphs grows, because most of them are serial algorithms and can only leverage the computing resource of a single CPU core, except the parallel algorithms developed by Bahmani et al. [5] which solve the DSD problems on both undirected and directed graphs in parallel using MapReduce. For example, the peeling-based approximation algorithms in [3] have a strong dependency in their steps; that is, when a vertex is peeled, all its neighbors' information needs to be updated synchronously before subsequent computation can be performed, making it hard to work in parallel. Besides, as shown by our later experiments in Section VI, the parallel algorithms in [5] are still inefficient for processing large-scale graphs, and their approximation ratios are larger than 2. Hence, it is desirable to develop more scalable and efficient DSD algorithms on both undirected and directed graphs at scale.

Our technical contributions. To alleviate the issues above, in this paper we aim to develop scalable and efficient DSD algorithms for both undirected and directed graphs. Our main idea is to develop efficient parallel approximation algorithms based on the shared-memory model that can work on massive graphs at scale. Specifically, for the undirected graph, we propose an efficient parallel algorithm by computing the k^*-core, which offers a 2-approximation solution [6]. We follow the h-index-based core decomposition approaches in [24], [25], which iteratively compute the h-indices of neighbor vertex degrees to obtain the core number of each vertex in a local and parallel manner. Instead of computing the core numbers of all vertices like these approaches, we only focus on computing the core numbers of vertices in the k^*-core, and thus avoid such redundant computation for all the other k-cores, thereby improving the efficiency significantly.

For the directed graph, we also develop an efficient parallel algorithm by computing the [x^*, y^*]-core, which provides a 2-approximation solution [7]. To facilitate the computation, we introduce a novel subgraph model, namely w-induced subgraph, by using an edge weight threshold w (w >0). More precisely, we first define the weight of each directed edge (u, v) as the product of the out-degree of u and in-degree of v, and then the w-induced subgraph can be induced by using all the edges with weights being at least w. After that, we present an approach to computing the w-induced subgraph with the maximum edge weight, also called w^*-induced subgraph, and theoretically prove that the [x^*, y^*]-core can be easily derived through the w^*-induced subgraph. Since our approach only needs to compute a single w^*-induced subgraph, it runs much faster than the original algorithm of computing the [x^*, y^*]-core which computes a list of [x, y]-cores [7]. Besides, the approach does not require any synchronization among vertices during computation. Thus, it can be easily parallelized.

In addition, we have implemented all the proposed algorithms using OpenMP[1], a popular parallel computing framework based on the shared-memory model. To evaluate the scalability and efficiency of our algorithms, we have conducted extensive experiments using 12 real large graphs, and four of them are billion-scale. Our experiments show that compared to state-of-the-art algorithms on undirected and directed graphs, our proposed parallel algorithms could achieve up to 20× and 30× speedup with 32 threads, respectively.

In summary, our principal contributions are listed as follows:

- For undirected graphs, we present an efficient parallel algorithm to compute the k^*-core as an approximate solution, by avoiding computing the core numbers of all the vertices.
- For directed graphs, we devise a fast parallel algorithm to compute the $[x^*, y^*]$-core by introducing the w-induced subgraph and also establishing its theoretical connection with $[x^*, y^*]$-core.
- We implement our algorithms using OpenMP and conduct extensive experiments on 12 real large graphs to test the scalability and efficiency of our proposed algorithms.

Outline. Section II reviews the existing DSD works. We introduce the preliminaries with formal problem definitions in Section III. Sections IV and V present our proposed parallel DSD algorithms on undirected graphs and directed graphs respectively. We report the experimental results in Section VI, and conclude the paper in Section VII.

II. Related Work

We mainly review the existing DSD works as shown in Table 1. We also briefly discuss other related works.

• **DSD on undirected graphs.** Goldberg [1] first introduced the problem of DSD on undirected graphs, and designed an exact DSD algorithm based on flow network. Charikar [3] introduced a linear programming (LP) approach for solving the DSD problem. Fang et al. [6] improved the efficiency by locating the densest subgraph in some specific k-cores. These exact algorithms can process small graphs in reasonable time cost, but they are inefficient and scales poorly on large graphs. To alleviate the issues above, many approximation algorithms have been proposed. Charikar [3] proposed a peeling-based 2-approximation algorithm with $O(m+n)$ time cost. Bahmani et al. [5] proposed a $2(1+\epsilon)$-approximation algorithm based on MapReduce and streaming model ($\epsilon > 0$). Boob et al. [26] proposed a 2-approximation approach based on a greedy strategy. Fang et al. [6] devised a fast 2-approximation algorithm based on the k-core. Bahmani et al. [27] and Su et al. [28] proposed $(1+\epsilon)$-approximation algorithms based on dual LP respectively. Chekuri et al. [29] designed a flow-based $(1+\epsilon)$-approximation algorithm with $\tilde{O}(\frac{m}{\epsilon})$ time cost.

[1] https://www.openmp.org/

TABLE 1: Classification of existing DSD algorithms.

Graph type	Exact algorithms	Approx. algorithms
Undirected graph	[1], [3], [6]	2-approximation: [3], [6], [8], [26] 2(1+ϵ)-approximation: [5]
Directed graph	[2]–[4]	2-approximation: [3], [7], [9] 2(1+ϵ)-approximation: [5]

Besides, there are many variants of DSD on undirected graphs. Tsourakakis et al. introduced the k-clique-density by extending classic graph density, and the DSD problem using k-clique-density has been extensively studied recently [6], [8], [22], [30], [31]. Sawlani and Wang [32] presented a $(1+\epsilon)$-approximation algorithm for dynamic graphs. Qin et al. [33] studied the top-k locally DSD problem. Tatti et al. [34] and Danisch et al. [23] studied the density-friendly decomposition problem to obtain a series of nested dense subgraphs. Galbrun et al. [35] and Dondi et al. [36], [37] studied the top-k densest subgraphs with minimum overlapping. Another variant is the densest k-subgraph search problem, which finds the densest subgraph with only k vertices [38]–[42].

• **DSD on directed graphs.** Kanan and Vinay [2] first introduced the notion of the density of a directed graph. Charikar [3] proposed an LP-based exact algorithm. Khuller and Saha [4] proposed a max-flow-based exact algorithm. Ma et al. [7], [9] introduced the $[x, y]$-core on directed graphs, which can be computed by iteratively peeling vertices whose in-degrees and out-degrees are less than x and y respectively, and proposed a fast exact core-based algorithm with divide-and-conquer strategy. Similar to exact algorithms for undirected graphs, the exact algorithms above also suffer from the issues of low efficiency and poor scalability, so many approximation algorithms have been developed. Kanan and Vinay [2] proposed a $O(\log n)$-approximation algorithm. Charikar [3] proposed a peeling-based 2-approximation algorithm. Khuller and Saha [4] proposed an approximation algorithm with linear time cost, but its approximation ratio is misclaimed to be 2 [7]. Bahmani et al. [5] proposed a parallel $2(1+\epsilon)$-approximation algorithm based on MapReduce and the streaming model ($\epsilon > 0$). Ma et al. [7], [9] proved that the $[x, y]$-core that maximizes the value of $x \cdot y$ is a 2-approximation solution to the DDS problem, which can be identified by enumerating all the $[x, y]$-cores that achieve the maximum values of x and y respectively.

In addition, the DSD problem has been studied on bipartite graphs [22], [43], [44], uncertain graphs [45], [46], and multilayer graphs [47]–[49]. Due to the different attributes and structures of these graphs, the solutions of these works cannot be applied to the problems studied in this paper.

• **Other dense subgraphs.** Recently, many other dense subgraph models have been studied [50], such as k-core [25], [51], k-truss [52], clique, quasi-clique [53], (α, β)-core [54], bitruss [55], biclique [56], [57], and quasi-bicliques [58]. More details can be found in [59], [60]. These dense subgraphs are different from UDS and DDS, which attain the highest density. Among them, the k-core is closely related to UDS, since the k-core with the largest k is a 2-approximation solution to the UDS problem [6]. Thus, existing parallel k-core decomposition approaches [25], [50], [61] can be used for finding UDS. In [25], Sariyuce et al. developed a local core decomposition algorithm on the multi-core platform, which iteratively updates the h-index of each vertex until it converges to its core number in a parallel manner. In [61], Kabir et al. divided all the vertices into k^* levels according to their degrees, and then computed the core numbers of vertices by processing vertices level by level in parallel.

In summary, most of the existing DSD algorithms scale poorly on large graphs except the parallel DSD algorithms developed in [5], since they are serial algorithms and can only leverage the computing resource of a single CPU core, thus calling for more scalable and efficient DSD algorithms.

III. Preliminaries

In this section, we formally introduce the DSD problems on undirected and directed graphs, also called *undirected densest subgraph* (UDS) and *directed densest subgraph* (DDS) problems respectively. Table 2 summarizes the frequently-used notations and their meanings.

TABLE 2: Frequently-used notations.

Notation	Definition
G=(V, E)	An undirected graph with vertex set V and edge set E
D=(V, E)	A directed graph with vertex set V and edge set E
ρ	Graph density
$N_G(v)$	The neighbors of v in G
$d_G(v)$	The degree of vertex v in G
$N_D^+(v), N_D^-(v)$	The out/in-neighbors of v in D, respectively
$d_D^+(v), d_D^-(v)$	The out/in-degree of v in D, respectively
$h(v)$	The h-index of v in G
k^*-core	The k-core of G with the maximum k
$[x^*, y^*]$-core	The $[x, y]$-core of D with the maximum $x \cdot y$
$N_D^+(v), N_D^-(v)$	The out/in-neighbors of v in D, respectively

A. Problem statement

We denote an undirected graph by $G = (V, E)$, where $|V| = n$ and $|E| = m$ are the numbers of vertices and edges of G, respectively. For each vertex $v \in V$, $N_G(v)$ is the neighbors of v and $d_G(v)$ denotes the degree of v, i.e., $d_G(v) = |N_G(v)|$. Given a vertex subset $S \subseteq V$, the subgraph induced by S, or $G[S]$, includes the vertex set S and the edge set $E(S) = E \cap (S \times S)$. The density, UDS, and UDS problem are formally defined as follows.

Definition 1 (Density of undirected graphs [1])**.** *Given an undirected graph $G = (V, E)$ and a vertex subset $S \subseteq V$, the density of the subgraph $G[S]$ induced by S is defined as $\rho(G[S]) = \frac{|E(S)|}{|S|}$.*

Definition 2 (Undirected densest subgraph (UDS) [1])**.** *Given an undirected graph $G = (V, E)$, an UDS of G is the subgraph of G with the maximum density among all the possible subgraphs.*

Problem 1 (UDS problem [1], [3], [5], [6], [26])**.** *Given an undirected graph G, return a subgraph of G with the maximum density.*

Let $D = (V, E)$ be a directed graph. For each vertex $v \in V$, denote by $N_D^+(v)$ and $N_D^-(v)$ the out and in-neighbors of v, respectively, and correspondingly denote by $d_D^+(v)$ and $d_D^-(v)$ the out and in-degree of v, respectively. Given two vertex subsets $S, T \subseteq V$ that are not necessarily disjoint, $E(S, T) = E \cap (S \times T)$ denotes the set of all edges from S to T in the graph D. The (S, T)-induced subgraph of D contains the vertex sets S, T and the edge set $E(S, T)$. The density of the (S, T)-induced subgraph, directed densest subgraph (DDS), and DDS problem are defined as follows.

Definition 3 (Density of directed graphs [2])**.** *Given a directed graph $D = (V, E)$ and two vertex subset $S, T \in V$,*

the density of the (S,T)-induced subgraph is defined as $\rho(S,T) = \frac{|E(S,T)|}{\sqrt{|S||T|}}$.

Definition 4 (Directed densest subgraph (DDS) [2])**.** *Given a directed graph $D = (V,E)$, a DDS of D is the (S,T)-induced subgraph of D with the maximum density among all the possible (S,T)-induced subgraphs.*

Problem 2 (DDS problem [2], [4], [5], [7], [9])**.** *Given a directed graph D, return an (S,T)-induced subgraph of D having the maximum density.*

Approximation ratio. Denote by $\rho*$ the density of the densest subgraph. An algorithm is called an α-approximation solution to the UDS/DDS problem if the ratio of ρ^* over the density of the returned subgraph is not greater than α ($\alpha \geq 1.0$). This paper focuses on the 2-approximation solutions to the UDS and DDS problems.

OpenMP. Open Multi-Processing (or OpenMP in short) is an application programming interface that supports shared-memory multiprocessing programming on a wide range of Symmetrical Multi-Processing architectures (e.g., multi-core CPUs). It shares most of the data in the parallel region by default, implying that all the branch threads can access the data at the same time. Recently, OpenMP has been widely used for implementing many graph algorithms, such as PageRank [62], k-core decomposition [61], clique enumeration [63], and graph partitioning [64].

IV. Algorithms for undirected densest subgraph discovery

In this section, we first review a state-of-the-art 2-approximation UDS algorithm [6] based on k-core, then show our proposed *parallel* 2-approximation algorithm, and finally present the parallel implementation using OpenMP.

A. State-of-the-art approximation algorithm

Before introducing the state-of-the-art 2-approximation UDS algorithm [6], we first give the definitions of k-core and core number.

Definition 5 (k-core [51], [65])**.** *Given an undirected graph $G = (V,E)$, the k-core of G is the largest induced subgraph $G[S] = (S, E(S))$ of G, where the degree of each vertex in $G[S]$ is not less than k, i.e., $\forall v \in S, d_{G[S]}(v) \geq k$.*

Definition 6 (Core number [51])**.** *The core number of a vertex is the largest value of k such that there is a k-core containing it.*

Among all the k-cores of G, the k-core with the largest k value is called k^*-core. Fang et al. [6] proved that the k^*-core provides a 2-approximation solution to the UDS problem.

Lemma 1 [6]. *Given an undirected graph G, the k^*-core of G is a 2-approximation solution to the UDS problem.*

To obtain the k^*-core, the straightforward way is to perform the core decomposition on the given graph to compute the core number for each vertex. Then, the vertices with the largest core number (i.e., k^*) comprise the k^*-core. The conventional method of k-core decomposition is to iteratively delete the vertex with the minimum degree in G, which can be done in $O(m)$ time via binsort [66].

However, the approach to remove the vertex with the smallest degree each time is not suitable for parallel computing, because every time the vertex is removed, it needs to wait for the degree update of the remaining vertices. Sariyuce et al. [25] proposed a parallel k-core decomposition algorithm, named `Local`, which calculates the core number of each vertex only based on the information of its neighbors. The main idea of `Local` is similar to the computation of h-index [67], which was initially used to measure the citation impact of a scholar or a journal. For a scholar or journal, its h-index is defined as the maximum value h such that there exist at least h papers, each with a citation count at least h. Analogously, the core number k of a vertex u can be viewed as the largest k such that u has at least k neighbors whose core numbers are also at least k.

As a result, the core number of each vertex can be obtained by iteratively updating the h-indexes of all vertices in the graph, until their values converge to their core numbers. For example, as shown in Fig. 2, the h-index of each vertex is initialized to its degree (i.e., $h^{(0)}$). We first update the h-index values of vertices in non-ascending order of degrees (i.e., v_4, v_3, v_2, v_1, v_5, v_6, v_7, v_8). After the first iteration, $h^{(1)}(v_7)$ will be reduced from 2 to 1, because v_7 does not have 2 neighbors whose degrees are at least 2. By analogy, after four rounds of iterations, the h-indexes of all vertices do not change, which means that their values converge to their core numbers.

Algorithm 1: `Local` [25]

Input: An undirected graph $G = (V,E)$
Output: The core number of all the vertices of G

1 **foreach** $v \in V$ **do** $h^{(0)}(v) = d_G(v)$;
2 $t \leftarrow 0$, $F \leftarrow$ **True**;
3 **while** F **do**
4 $F \leftarrow$ **False**;
5 **for** $v \in V$ **in parallel do**
6 $h^{(t+1)}(v) \leftarrow$ the maximum k satisfying $|\{h^{(t)} \geq k | u \in N_G(v)\}| \geq k$;
7 **if** $h^{(t+1)}(v) < h^{(t)}(v)$ **then** $F \leftarrow$ **True** ;
8 $t \leftarrow t + 1$;
9 **return** $\{h^{(t)}(v) | v \in V\}$;

Algorithm 1 presents the steps of the algorithm `Local` [25]. Specifically, the h-index of each vertex in G is initialized to its degree (line 1), and then each vertex updates its h-index iteratively according to the h-index values of its neighbors (lines 3-8). If the h-index values of all vertices remain unchanged in two consecutive iterations, the h-index of each vertex converges to its core number (line 7). Clearly, the h-index value of a vertex provides an upper bound of its core number. Since the h-index value of a vertex only depends on its neighbors' information, the h-index values of all the vertices can be updated independently without synchronization.

Note that the k^*-core may have multiple connected components, and any one of them can be regarded as a 2-approximation solution to the UDS problem.

Complexity. `Local` completes in $O(t \cdot m)$ time, where $t \leq n$ is the number of iterations, and in practice $t \ll k^*$ [25].

B. Our optimized k^*-core computation

Although the k^*-core can be obtained by using the core decomposition algorithms above, these algorithms involve much unnecessary computation, since they compute the core numbers of all vertices, while for the UDS problem, we only need the k^*-core by Lemma 1. In other words, for

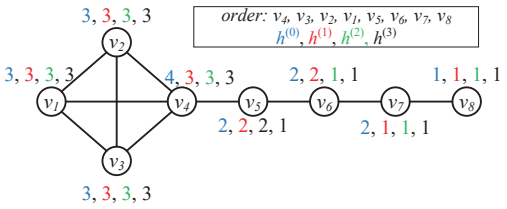


Fig. 2: Illustrating the optimized `Local` algorithm.

many vertices that are not in the k^*-core, their core numbers have also been precisely computed. Therefore, to improve the efficiency of solving the UDS problem, the main challenge is how to reduce the computation of core numbers for vertices that are not in the k^*-core, given that k^* is unknown in advance.

A simple method of computing the k^*-core without computing all vertices' core numbers is to perform core decomposition with binary search on the possible values of k^*. Specifically, for each guessed $\widehat{k^*}$ of k^*, we first select vertices with degrees of $\widehat{k^*}$ or more, and then do the core decomposition on the induced subgraph. If the maximum core number obtained is at least $\widehat{k^*}$, then it must be k^* for the entire graph G; otherwise, we need to reduce the value of $\widehat{k^*}$ and repeat the above process. Nevertheless, the number of binary search iterations could be $O(\log n)$ in the worst case, making the overall time cost be $O((m+n)\log n)$. As a result, this method may be even slower than the algorithms above.

To alleviate the above issues, we first show some interesting properties of k^*-core and `Local` (Algorithm 1), and then use them to optimize `Local`, by significantly reducing the number of its iterations and also trying to avoid computing the core numbers for the vertices outside the k^*-core.

Proposition 1. *There are at least k^*+1 vertices in k^*-core.*

Proposition 2. *Given an undirected graph G, the degrees/h-indices of vertices outside the k^*-core have no effect on the core number of its internal vertices.*

Lemma 2. *For the t-th iteration in Algorithm 1, if $h^{(t)}(v)$ equals to the core number of v, then $\forall q > t$, $h^{(q)}(v) = h^{(t)}(v)$, i.e., the h-index value of v will not change in the following iterations.*

Through the above properties, we can make the following conclusion: in an iteration of Algorithm 1, the k^*-core is obtained if the h-index values of all vertices in k^*-core are k^* and the maximum h-index value of other vertices is less than k^*. However, in practice, the value of k^* and the number of vertices in k^*-core are unknown in advance, so we cannot get the k^*-core directly by using the conclusion. Luckily, this conclusion inspires us to derive an interesting theorem which provides an effective early stop criterion, allowing us to obtain the k^*-core from the first a few iterations of `Local` without running all the iterations. In other words, equipped with this theorem, the optimized `Local` can report the k^*-core immediately once it has been derived, so the computation of other k-cores will be reduced significantly.

Theorem 1. *In the t-th iteration of Algorithm 1, let $h_{max}^{(t+1)} = \max_{v\in V}\{h^{(t+1)}(v)\}$ and $h_{max}^{(t)} = \max_{v\in V}\{h^{(t)}(v)\}$. If $h_{max}^{(t+1)} = h_{max}^{(t)}$ and $|\{v \in V | h^{(t+1)}(v) = h_{max}^{(t+1)}\}| = |\{v \in V | h^{(t)}(v) = h_{max}^{(t)}\}|$, then $k^* = h_{max}^{(t)}$ and the subgraph induced by $\{v \in V | h^{(t)}(v) = h_{max}^{(t)}\}$ is the k^*-core of the input graph.*

Proof. We prove the theorem by contradiction. Assuming that $k^* \neq h_{max}^{(t)}$ if $h_{max}^{(t+1)} = h_{max}^{(t)}$ and $|\{v \in V | h^{(t+1)}(v) = h_{max}^{(t+1)}\}| = |\{v \in V | h^{(t)}(v) = h_{max}^{(t)}\}|$. That is, $k^* < h_{max}^{(t)}$, because for each vertex $v \in V$, $h^t(v)$ is the upper bound of the core number of v. In the subgraph induced by $S = \{v \in V | h^{(t)}(v) = h_{max}^{(t)}\}$, each vertex has a degree larger than or equal to $h_{max}^{(t)}$, according to the definition of h-index and line 7 of Algorithm 1. Hence, the subgraph $G[S]$ is a $h_{max}^{(t)}$-core, which contradicts $k^* < h_{max}^{(t)}$. □

By Theorem 1, we can optimize `Local` by stopping its iteration process directly, whenever we find that the maximum values of h-index values in two adjacent iterations are the same and meanwhile the number of corresponding vertices remains unchanged, because the k^*-core has been already derived from these iterations.

Since the original `Local` is a highly parallel algorithm and we only use its first several iterations for computing the k^*-core, the optimized algorithm is still a highly parallel algorithm, allowing us to find a 2-approximation solution to the UDS problem in parallel. We illustrate the optimized `Local` algorithm by Example 1.

Example 1. *We illustrate Theorem 1 by running `Local` on an example graph in Fig. 2. Initially, the h-index values of all vertices are their degrees, so $h_{max}^{(0)}$=4 and there is 1 vertex with h-index values equal to $h_{max}^{(0)}$ is 1. Next, we update the h-index values of vertices in non-ascending order of degrees, (i.e., $v_4, v_3, v_2, v_1, v_5, v_6, v_7, v_8$), and then get $h_{max}^{(1)}$ = 3 with 4 vertices whose h-index values equal to $h_{max}^{(1)}$. In the next iteration, $h_{max}^{(2)}$ is still 3 and there are still 4 vertices with h-index values equal to $h_{max}^{(2)}$. By Theorem 1, the k^*-core has been obtained at this moment, which includes vertices $\{v_1, v_2, v_3, v_4\}$.*

Notice that the h-index values of other vertices still need two more iterations to converge to their corresponding core numbers, when there is no h-index update in the entire graph. Hence, the optimized `Local` only needs 2 iterations, while the original `Local` needs 4 iterations.

C. The overall algorithm

In this section, we present the optimized `Local` algorithm with the parallel implementation using OpenMP, which essentially is a parallel k^*-core computation (also abbreviated as `PKMC`) algorithm. We show the pseudocodes of `PKMC` in Algorithm 2.

Specifically, we first initialize the h-index value of each vertex by its degree (line 1). $h_{\max}^{(t)}$ records the maximum value of $h^{(t)}(v)$ in the t-th iteration, which is initialized to the maximum degree of all vertices (line 2). $s^{(t)}$ denotes the number of vertices whose h-index values equal to $h_{\max}^{(t)}$ in the t-th iteration (line 3). Next, we update the h-index values of all vertices iteratively (lines 6-9). Particularly, the h-index update of each vertex can be performed independently without synchronization (line 7). After each iteration, we obtain the maximum value $h_{max}^{(t+1)}$ of the $(t+1)$-iteration, and the number of corresponding vertices $s^{(t+1)}$ (lines 10-11). By proposition 1, if $s^{(t+1)} \leq h_{max}^{(t+1)}$, then continue

the while-loop (line 12). Afterwards, if $h_{max}^{(t)} = h_{max}^{(t+1)}$ and $s^{(t+1)}{=}s^{(t)}$, we can stop the loop directly by setting F to **False** according to Theorem 1. Otherwise, we need to continue the h-index update iterations (lines 13 – 14). Finally, the subgraph induced by $\{v \in V | h^{(t)}(v) = h_{max}^{(t)}\}$ is the k^*-core, which gives a 2-approximation solution to the UDS problem.

Algorithm 2: Parallel k^*-core computation (PKMC)

```
   Input: An undirected graph G = (V, E)
   Output: The 2-approximation UDS, i.e., k*-core
1  foreach v ∈ V do h^(0)(v) ← d_G(v);
2  h_max^(0) ← max_{v∈V}{h^(0)(v)};
3  s^(0) ← |{v ∈ V | h^(t)(v) = h_max^(t)}|;
4  t ← 0, F ← True;
5  while F do
6      F ← False;
7      for v ∈ V in parallel do
8          h^(t+1)(v) ← the maximum k satisfying
             |{h^(t) ≥ k | u ∈ N_G(v)}| ≥ k;
9          if h^(t+1)(v) < h^(t)(v) then F ← True ;
10     h_max^(t+1) ← max_{v∈V}{h^(t+1)(v)};
11     s^(t+1) ← |{v ∈ V | h^(t+1)(v) = h_max^(t+1)}|;
12     if s^(t+1) ≤ h_max^(t+1) then continue;
13     if h_max^(t) = h_max^(t+1) then
14         if s^(t+1)=s^(t) then F ← False ;
15     t ← t + 1;
16 return the subgraph by {v ∈ V | h^(t)(v) = h_max^(t)};
```

Time complexity. The worst case time complexity of PKMC (Algorithm 2) is the same as that of Local (Algorithm 1), i.e., $O(t \cdot m)$. As analyzed by Sariyuce et al. [25], there is no dependency on the update of the h-index of vertices; that is, the order of the update of the h-index of the vertices does not affect the correctness of the result. Consequently, the span (depth) of each iteration is $O(1)$, and the span of PKMC is $O(t)$. In practice, since the number of iterations needed by PKMC is often much less than that of Local, PKMC runs much faster than Local as shown by our later experimental results in Section VI.

V. Algorithms for directed densest subgraph discovery

In this section, we first review the state-of-the-art 2-approximation DDS algorithm [7] based on the $[x, y]$-core. Then, we propose a new subgraph model, called w-induced subgraph, and a novel parallel 2-approximation DDS algorithm based on the w-induced subgraph. Finally, we show how to implement it using OpenMP.

A. State-of-the-art approximation algorithm

The state-of-the-art 2-approximation DDS algorithm is mainly based on the concept of $[x, y]$-core [7]:

Definition 7 ($[x, y]$-core [7])**.** *Given a directed graph $D = (V, E)$, an (S, T)-induced subgraph $H = G[S, T]$ is called an $[x, y]$-core, if it satisfies:*

1) *$\forall u \in S, d_H^+(u) \geq x$ and $\forall v \in T, d_H^-(v) \geq y$;*
2) *H is maximal, i.e., $\nexists H'$, s.t., H is a subgraph of H' and H' also satisfies (1).*

The integer pair $[x, y]$ is also called the *cn-pair* of the $[x, y]$-core. Among all the possible $[x, y]$-cores, the cn-pair with the largest value of $x \cdot y$ is called the maximum cn-pair, and the corresponding $[x, y]$-core is denoted by $[x^*, y^*]$-core.

Lemma 3 [7]. *Given a directed graph D, the $[x^*, y^*]$-core of D is a 2-approximation solution to the DDS problem.*

To obtain the $[x^*, y^*]$-core, Ma et al. [7] developed an algorithm by enumerating $O(\sqrt{m})$ cn-pairs. Specifically, for each possible $x \in [1, \sqrt{m}]$ (resp. $y \in [1, \sqrt{m}]$), it iteratively peels the vertices with the smallest in-degree (resp. out-degree) to obtain the maximum y (resp. x) such that there exists an $[x, y]$-core. Afterwards, the $[x^*, y^*]$-core is obtained by selecting the largest value of all $x \cdot y$. Therefore, its overall time complexity is $O(\sqrt{m}(m+n))$.Similar to k^*-core, there may exist several connected components in an $[x^*, y^*]$-core, any of which can be regarded as a 2-approximation solution to the DDS problem.

To achieve higher scalability, we can easily adapt the above algorithm to run in parallel and the adapted algorithm is termed as PXY. The main idea of PXY is to dynamically assign each x to a specific thread for computing the corresponding cn-pairs. After that, we gather all the cn-pairs and get the maximum cn-pair. Although PXY can accelerate the computation of $[x^*, y^*]$-core, it still needs to decompose the graph to get $O(\sqrt{m})$ $[x, y]$-cores. Besides, it may suffer from the load imbalance issue since different cn-pairs have different computational cost. To tackle these issues, in the following we propose a novel subgraph model, called w-induced graph, for the directed graph and then show that the $[x^*, y^*]$-core can be easily computed in parallel by establishing its relationship with the w-induced subgraph. As a result, we obtain a novel scalable and efficient 2-approximation algorithm to the DDS problem.

B. The w-induced graph and its computation

Recall that when computing an $[x, y]$-core, all the peeling operations are restricted on the vertices. Then, a natural question is that *can we obtain a similar dense subgraph by performing operations on edges?* In the following, we show that this is not only possible, but also can be done efficiently in parallel.

Remember that for each edge $e = (u, v)$ in an $[x, y]$-core (denoted by H), we always have $d_H^+(u) \geq x$ and $d_H^-(v) \geq y$. Consequently, the product of degrees on both sides of each directed edge in an $[x, y]$-core must be at least $x \cdot y$. Inspired by this, for each edge, we can assign it a weight via the degrees of its two endpoints.

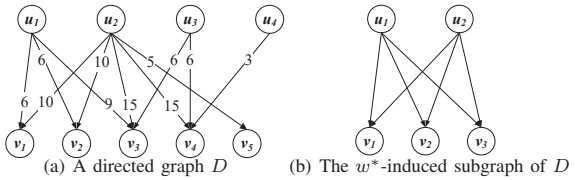
Definition 8 (Edge weight)**.** *Given a directed graph $D = (V, E)$, for each edge $e = (u, v) \in E$, its assigned edge weight w.r.t. D is defined as $w_D(e) = d_D^+(u) \cdot d_D^-(v)$.*

Based on the definition of edge weight, we introduce the concepts of w-induced subgraph and induce-number.

Definition 9 (w-induced subgraph)**.** *Given a directed graph $D = (V, E)$ and an integer w, the w-induced subgraph is the maximal subgraph $H = (V_H, E_H)$ of D, such that the weight of each edge w.r.t. H is at least w, i.e., $\forall e = (u, v) \in E_H$, we have $w_H(e) = d_H^+(u) \cdot d_H^-(v) \geq w$.*

Definition 10 (Induce-number)**.** *Given a directed graph $D = (V, E)$ and an edge $e \in E$, the induce-number of e is the largest value of w, such that that there exists a w-induced subgraph containing it. We denote the maximum induce-number of edges in D by w^*.*

Example 2. *Consider the directed graph D in Fig. 3(a). The value on each edge denotes its assigned edge weight. For instance, the weight of edge $e = (u_1, v_3)$ is $w_D(e) = d_D^+(u_1) \times d_D^-(v_3) = 3 \times 3 = 9$. After initializing the weights of all edges, we iteratively remove the edge with the smallest weight from the graph. First, we remove edge (u_4, v_4) and record its induced number as 3. Since the in-degree of v_4 decreases from 3 to 2, we update the weights of (u_3, v_4) and (u_2, v_4) to 4 and 10 respectively. Afterward, there is no edge with a weight of 3 in the remaining graph. Then, we continue to find the smallest weight and the corresponding edges in the graph, and repeat the above steps until the graph is empty, and the induced numbers of all edges in the graph are shown in Table 3. Clearly, the maximum induce-number is $w^* = 6$, and the w^*-induced subgraph contains vertices $\{u_1, u_2, v_1, v_2, v_3\}$ as depicted in Fig. 3(b).*



(a) A directed graph D (b) The w^*-induced subgraph of D

Fig. 3: An example of w-induced subgraph.

TABLE 3: Induce-numbers of edges of the graph in Fig. 3(a).

Edge(s)	Induce-number
(u_4, v_4)	3
$(u_3, v_3), (u_3, v_4)$	4
$(u_2, v_4), (u_2, v_5)$	5
$(u_1, v_1), (u_1, v_2), (u_1, v_3)$, $(u_2, v_1), (u_2, v_2), (u_2, v_3)$	6

Next, we show that the w-induced subgraphs have some interesting properties:

Proposition 3 (Nested Property)**.** *Given a directed graph D, a w-induced subgraph is contained by a w'-induced subgraph, if $w \geq w' \geq 0$.*

Proposition 4. *Given a directed graph D and its w^*-induced subgraph, removing all edges in w^*-induced subgraph with weights being exactly w^* will collapse the w^*-induced subgraph.*

Similar to k-core decomposition, to decompose the graph for getting all the possible w-induced subgraphs, we can compute the induce-numbers of all the edges. This can be done efficiently in parallel by following the k-core decomposition algorithm; that is, we iteratively peel the edges with the smallest weights, record their induce-numbers, and in the meantime update the weights of their adjacent edges, until the graph is empty.

Algorithm 3 presents the parallel algorithm for decomposing the w-induced subgraphs. Specifically, we first use a set $N(v)$ to keep the out-neighbors of each vertex v, use another set A to keep all vertices having out-neighbors, and initialize a Boolean value F to be false (lines 1-2). Afterwards, we iteratively remove the edges with the smallest weight by the outer while-loop (lines 3-15). In each iteration t, we first induce a subgraph by using edges which connect vertices of A (line 4), then get the minimum edge weight $w^{(t)}$ (line 5),

Algorithm 3: Parallel w-induced subgraph decomposition

Input: A directed graph $D = (V, E)$
Output: Induce-numbers of all the edges of D

```
1  foreach v ∈ A do N(v) ← N_D^+(v);
2  A ← {v ∈ V | d_D^+(v) > 0}, F ← True, t ← 0;
3  while A is not empty do
4      H ← the subgraph induced by the edges implied
         by neighbors of each v ∈ A;
5      w^(t) ← the minimum weight of edges in H;
6      while F do
7          F ← False;
8          for u ∈ A in parallel do
9              foreach v ∈ N(u) do
10                 if d_H^-(v) ≤ w^(t)/d_H^+(u) then
11                     record the induce-number of (u, v)
                         as w^(t);
12                     remove v from N(u);
13                     update d_H^+(u), d_H^-(v) atomically;
14                     F ← True;
15             if d_H^+(u) = 0 then remove u from A;
16     t ← t + 1;
17 return the induce-number of all edges of D;
```

and finally peel all the edges with such a weight using the inner while-loop (lines 6-15). In the inner while-loop, when removing an edge (lines 9-14), we record its induce-number and meanwhile decrease the weights of its adjacent edges by 1. Finally, the induce-numbers of all the edges are returned (line 17).

Time complexity. Let d_{max} be the maximum in-degree/out-degree of vertices in D. In the worst case, Algorithm 3 takes $O(m \cdot d_{max})$ time, since the weight of each edge will be decreased d_{max} times, but in practice it is much faster.

Remark. Note that we aim to obtain the w^*-induced subgraph of D, and for the directed graph D, there must be an induced subgraph consisting of the vertex with the largest out/in-degree and its neighbors. Therefore, $w* \geq d_{max} = max_{u \in D}\{d^+(u), d^-(u)\}$. Then we can set $w^{(0)}$ to d_{max} to speed up the w^*-induced subgraph computation.

C. Deriving $[x^*, y^*]$ from w^*-induced subgraph

In this subsection, we theoretically establish the relationship between the maximum cn-pair $[x^*, y^*]$ and the w^*-induced subgraph, and then discuss how to derive $[x^*, y^*]$ from the w^*-induced subgraph.

Intuitively, an $[x, y]$-core is a subgraph of a w-induced subgraph. For example, as shown in Fig. 4, a $[4, 3]$-core is contained in a 12-induced subgraph. However, there may be some edges in the w-induced subgraph that do not belong to the $[x, y]$-core.

Example 3. *Take the directed graph D in Fig. 4 as an example. Note that the whole graph D is a w^*-induced subgraph with w^* = 12. The $[x^*, y^*]$-core of D is the (S, T)-induced subgraph with vertex sets $S = \{u_1, u_2, u_3\}$ and $T = \{v_1, v_2, v_3, v_4\}$, which means that x^*=4 and y^*=3. In the 12-induced subgraph, the weights of four edges (u_2, v_6), (u_4, v_6), (u_3, v_7), and (u_4, v_7) are 12, but they do not belong to $[x^*, y^*]$-core, since the in-degrees of v_6 and v_7 are less than 3.*

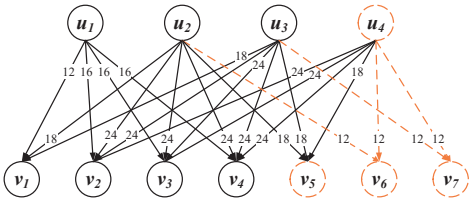


Fig. 4: An example of w^*-induced subgraph and $[x^*, y^*]$-core, where w^*=12, x^*=4, and y^*=3.

To establish a theoretical connection between the two subgraph models, we begin with two interesting lemmas about the $[x, y]$-core:

Lemma 4. *Given a directed graph D, for any $[x, y]$-core in D, it must be contained by the xy-induced subgraph, but an xy-induced subgraph may include multiple $[x, y]$-cores.*

Lemma 5. *Given a directed graph D, removing all the edges, whose weights are exactly $x^* \cdot y^*$, from its $[x^*, y^*]$-core will collapse the entire $[x^*, y^*]$-core.*

Proof. We prove the lemma by contradiction. Assuming that there exists a subgraph H after removing all edges with weights being exactly $x^* \cdot y^*$ from $[x^*, y^*]$-core, which is an (S, T)-induced subgraph. Essentially, removing all the edges of the $[x^*, y^*]$-core with weights being exactly $x^* \cdot y^*$ is equivalent to removing all the vertices with out-degrees being exactly x^* and in-degrees being exactly y^* from S and T, respectively. This implies that H is an $[x', y']$-core with $x' \cdot y' > x^* \cdot y^*$, which contradicts the fact that $x^* \cdot y^*$ is the largest value among all $[x, y]$-cores of D. Hence, the lemma holds. □

By Lemma 4, we know that the $[x^*, y^*]$-core is a subgraph of $(x^* \cdot y^*)$-induced subgraph. According to Lemma 5, the minimum weight of edges in the $[x^*, y^*]$-core is exactly $x^* \cdot y^*$, which also implies that the induce-number of each edge in $[x^*, y^*]$-core is at least $x^* \cdot y^*$. Next, we formally establish the relationship between $[x^*, y^*]$-core and w^*-induced subgraph:

Theorem 2. *Given a directed graph D, the w^*-induced subgraph of D, and the $[x^*, y^*]$-core of D, we have $w^* = x^* \cdot y^*$.*

Proof. We prove the theorem by contradiction. If $w^* \neq x^* \cdot y^*$, then there are two cases: $w^* > x^* \cdot y^*$ and $w^* < x^* \cdot y^*$.

(1) If $w^* > x^* \cdot y^*$, then according to Proposition 3, the w^*-induced subgraph must be a subgraph of $[x^*, y^*]$-core. That is, the w^*-induced subgraph can be obtained by removing the edges with weights less than w^* in $[x^*, y^*]$-core. According to Lemma 5, $[x^*, y^*]$-core will collapse when removing the edges with weights of $x^* \cdot y^*$. Therefore, there is no subgraph with higher induce-number, which however contradicts the assumption that $w^* > x^* \cdot y^*$.

(2) If $w^* < x^* \cdot y^*$, then the $[x^*, y^*]$-core must be a subgraph of w^*-induced subgraph. Removing all the edges with weights less than $x^* \cdot y^*$ in the w^*-induced subgraph can obtain $[x^*, y^*]$-core, which contradicts Proposition 4.

Therefore, we conclude $w^* = x^* \cdot y^*$. □

According to Theorem 2 and Proposition 4, we can conclude that the $[x^*, y^*]$-core must be a subgraph of w^*-induced subgraph. Besides, in the w^*-induced subgraph H, there may exist some edges $e = (u, v)$, which satisfy $d_H^+(u) \cdot d_H^-(v) \geq w^* = x^* \cdot y^*$, but $d_H^+(u) < x^* \wedge d_H^-(v) > y^*$, or $d_H^+(u) > x^* \wedge d_H^-(v) < y^*$, as illustrated in Example 3. Hence, it is a key issue to extract the edges related to the $[x^*, y^*]$-core from the w^*-induced subgraph.

Given the w^*-induced subgraph of a graph, to obtain the maximum cn-pair $[x^*, y^*]$, a simple method is to check each possible cn-pair $[x, y]$ such that $x \cdot y = w^*$ and then select the largest $x \cdot y$. Since we still need to enumerate all the $x \in [1, w^*]$ and check the existence of corresponding $[x, y]$-cores, this method is costly, especially when the value of w^* is very large. To improve the efficiency, we make an important observation: *among all the edges whose weights are exactly w^* in the w^*-induce subgraph, there must be some edges, whose endpoints' out-degrees and in-degrees are exactly x^* and y^* respectively.* If we remove all such edges, the w^*-induce subgraph will collapse, as indicated by Lemma 6.

Lemma 6. *Given a directed graph D, removing all edges, whose endpoints' out-degrees and in-degrees are exactly x^* and y^* respectively, from its w^*-induced subgraph will lead to the collapse of the entire w^*-induced subgraph immediately.*

Proof. According to Theorem 2, there must be edges in D whose endpoints' out-degrees and in-degrees are exactly $x^*$and y^* respectively. These edges are also in the $[x^*, y^*]$-core, and according to Lemma 5, removing them will lead to the collapse of the $[x^*, y^*]$-core. Note that the $[x^*, y^*]$-core is a subset of the w^*-induced subgraph, so the collapse of the $[x^*, y^*]$-core will result in the collapse of the w^*-induced subgraph. Thus, Lemma 6 holds. □

Inspired by the observation above, we propose a fast algorithm to obtain the maximum cn-pair directly from the w^*-induced subgraph, without computing any $[x, y]$-core. The key idea is that for all edges in the w^*-induced subgraph H whose weights are exactly w^*, we iteratively consider each degree pair $[x, y]$ satisfying $x \cdot y = w^*$. For each $[x, y]$, we delete all the edges whose endpoints have out-degrees and in-degrees being x and y respectively, and then update the weights of their adjacent edges. If the remaining subgraph cannot form a w^*-induced subgraph, the current degree pair $[x, y]$ is the maximum cn-pair, i.e., $x^* = x$ and $y^* = y$.

After obtaining $[x^*, y^*]$, we can extract the $[x^*, y^*]$-core from the w^*-induced subgraph directly as the 2-approximation DDS. It is worth noting that in the above algorithm, the subgraph revealing the maximum cn-pair $[x^*, y^*]$ may not be the $[x^*, y^*]$-core, since it does not impose any constraint on vertices' in-degrees and out-degrees. Therefore, to get the $[x^*, y^*]$-core, we have to continue removing all the vertices whose in-degrees and out-degrees are less than x^* and y^* respectively from the w^*-induced graph, until the remaining graph is an $[x^*, y^*]$-core.

Example 4. *Recall that in the graph of Fig. 4, we have w^*=12. To derive $[x^*, y^*]$, we first locate the five edges whose weights are exactly 12, i.e., (u_1, v_1), (u_2, v_6), (u_4, v_6), (u_3, v_7), and (u_4, v_7), where the first edge's degree pair is $[4, 3]$ and the remaining four edges' degree pairs are $[6, 2]$. Suppose we first remove all the edges whose degree pairs are $[6, 2]$, which mean that the vertices v_6 and v_7 will be removed. After that, the remaining graph is still a 12-induced graph. Hence, we continue removing all edges whose degree pairs are $[4, 3]$, which directly leads to the collapse of the remaining graph, so the maximum cn-pair must be $[4, 3]$. Note that if we first remove the edge whose degree pair*

is $[4, 3]$*, the graph will collapse immediately, which also indicates that the maximum cn-pair is* $[4, 3]$.

D. The overall algorithm

To improve the parallelizability, we develop a novel approach. Instead of focusing on deleting edges in parallel, we iterate on the vertices having out-neighbors in parallel. Recall that each out-neighbor of a vertex corresponds to an out-going edge adjacent to the vertex, and the product of the two degrees (i.e., an out-degree and an in-degree) is the weight of the edge. For example, the out-neighbor v_1 of u_1 in Figure 3(b) corresponds to the edge (u_1, v_1), and the product of the two degrees is its weight 6. Consequently, when an edge $e = (u, v)$ is removed, we can simply remove v from the out-neighbor set of u and update the degrees of the two endpoints. In this way, each vertex only needs to take actions based on its neighbor information, thereby improving the parallelizability.

Algorithm 4: Parallel $[x^*, y^*]$-core computation (PWC)

```
   Input: A directed graph D = (V, E)
   Output: The [x*, y*]-core
1  H ← invoke Algorithm 3 to compute w*-induced
     subgraph;
2  A ← {v ∈ H | d_H^+(v) > 0}, P ← ∅, F ← True;
3  foreach v ∈ A do N(v) ← N_H^+(v);
4  for u ∈ A in parallel do
5      foreach v ∈ N(u) do
6          if d_H^-(v) = w*/d_H^+(u) then add d_H^-(v) to P ;
7  while F do
8      F ← False, d* ← P.pop();
9      for u ∈ A in parallel do
10         foreach v ∈ N(u) do
11             if d_H^-(v) < w*/d_H^+(u) then
12                 delete v from N(u), update d_H^+(u), d_H^-(v)
                     atomically;
13                 F ← True;
14             if d_H^-(v)=d* and d_H^+(u) · d* =w* then
15                 x* ← d_H^+(u), y* ← d*;
16                 delete v from N(u), update d_H^+(u), d_H^-(v)
                     atomically;
17                 F ← True;
18         if d_H^+(u) = 0 then remove u from A;
19     if A = ∅ then F ← False;
20 extract [x*, y*]-core from H;
21 return [x*, y*]-core;
```

Algorithm 4 presents the steps of our parallel implementation for the algorithm above, which is denoted by PWC. It first computes the w^*-induced subgraph and collects a set A of vertices with out-degrees larger than 0 (lines 1-2). For each vertex v in A, we initialize its neighbor set $N(v)$ as the outgoing neighbors of v in H (line 3). Next, for each vertex u with out-degree larger than 0, we iterate over each of its out-neighbors v, and record the in-degree of v by P if $d_H^+(u) \cdot d_H^-(v) = w^*$ (lines 4-6). Afterwards, we use a while-loop to remove the edges corresponding to each in-degree in P (lines 7-19). Specifically, we take the current d^* from P each time, for each vertex $u \in A$, we remove vertex v with in-degree less than $w^*/d_H^+(u)$ from its neighbors (lines 11-13). If $d_H^-(v) = w^*/d_H^+(u)$, we only remove vertices whose in-degrees are equal to d^*, and $(d_H^+(u), d^*)$ is taken as the maximum cn-pair temporarily (lines 14-17). We update the degrees of the affected vertices along with the vertex removal (lines 12-16). After that, if the remaining graph is empty, meaning that we find the maximum cn-pair, PWC terminates the while-loop and returns the $[x^*, y^*]$-core (lines 19-21); otherwise, it repeats the above steps.

Time complexity. For each vertex u in A, updating the out-neighbors and degrees of u costs $O(d_D^+(u))$ time in the worst case, so the total time cost of an iteration in the while-loop is $O(m)$. Assuming that there are t iterations, Algorithm 4 takes $O(t \cdot m)$ time. Notice that when $d_H^+(u) = 0$, u will not be processed, so t is bounded by d_{max}, which is the maximum out-degree/in-degree of all vertices in D. Since the order of degree updates also does not affect the result, the span of each iteration is $O(1)$. Therefore, the span of PWC is $O(d_m)$.

VI. Experiments

We now present the experimental results. Section VI-A shows the setup. We report the results in Sections VI-B and VI-C.

A. Setup

Since we mainly focus on developing efficient parallel approximation algorithms, all the compared algorithms in the experiments are extended to their parallel versions. To make a fair comparison, for the existing serial algorithms, we adapt them such that they can be run in parallel. Specifically, for UDS problem, we compare the performance of the following algorithms:

- **PFW [23], [28]:** a $(1+\epsilon)$-approximation UDS algorithm, where ϵ is set to 1, and we parallelize the Frank-Wolfe process in the algorithm;
- **PBU [5]:** a parallel $2(1+\epsilon)$-approximation UDS algorithm, where ϵ is set to 0.5;
- **Local [25]:** the state-of-the-art parallel nucleus decomposition approach, which can be used for parallel k-core decomposition and getting the k^*-core;
- **PKC [61]:** the state-of-the-art parallel peeling algorithm for k-core computation, which returns the k^*-core;
- **PKMC:** our proposed parallel k^*-core computation algorithm, which is depicted in Algorithm 2.

Note that PBU is a parallel algorithm that can run on MapReduce; Local and PKC are parallel algorithms based on the shared-memory model.

For DDS problem, we compare the performance of the following parallel algorithms:

- **PBS [3]:** it is a greedy-based 2-approximation algorithm which needs n^2 rounds of peeling, and we parallelize it by using a thread to run each peeling round;
- **PFKS [4]:** it is a fixed version of the 2-approximation DDS algorithm which needs n rounds of peeling, and we parallelize it by using a thread to run each peeling round;
- **PFW [28]:** a $(1+\epsilon)$-approximation DDS algorithm, where ϵ is set to 1, and we parallelize the Frank-Wolfe process in the algorithm;
- **PBD [5]:** the parallel $2\delta(1+\epsilon)$-approximation DDS algorithm, where δ=2.0 and ϵ=1.0 by default;
- **PXY [9]:** it is adapted from the serial algorithm of computing the $[x^*, y^*]$-core, called Core-Approx in [7], which sequentially computes $\sqrt{m}$ $[x, y]$-cores to get the $[x^*, y^*]$-core and is parallelized by using a thread to compute each $[x, y]$-core, and it is the state-of-the-art 2-approximation algorithm;

- **PWC:** our proposed 2-approximation DDS algorithm, which is designed based on the w^*-induced subgraph and returns the $[x^*, y^*]$-core, as summarized in Algorithm 4.

Data sets. We consider 12 real-world large graphs, including 6 undirected graphs and 6 directed graphs, to evaluate the proposed UDS and DDS algorithms respectively. Tables 4 and 5 summarize the details of these graphs respectively, which cover a wide range of areas. Specifically, Petster is a graph composed of family links between cats and dogs; Twitter is a social media graph; Amazon is an e-commerce graph; Baidu and Wiki are encyclopedia graphs; EU, IT, and SK are graphs from the world wide web. All these graphs are obtained from KONECT[2] and LAW[3].

TABLE 4: Undirected graphs used in the experiments.

Graphs (Abbr.)	Category	$\|V\|$	$\|E\|$	d_{max}
Petster (PT)	Family link	623,766	15,699,276	80,637
eswiki-2013 (EW)	Knowledge	972,933	23,041,488	145,031
eu-2015 (EU)	Web	11,264,052	379,731,874	68,922
it-2004 (IT)	Web	41,291,594	1,150,725,436	1,326,745
sk-2005 (SK)	Web	50,636,154	1,949,412,601	8,563,808
uk-union (UN)	Web	133,633,040	5,507,679,822	6,366,525

TABLE 5: Directed graphs used in the experiments.

Graphs (Abbr.)	Category	$\|V\|$	$\|E\|$	d^+_{max}	d^-_{max}
Amazon (AM)	E-commerce	403,394	3,387,388	10	2,751
Amazon ratings (AR)	E-commerce	3,376,972	5,838,041	12,217	3,146
Baidu (BA)	Knowledge	2,141,300	17,794,839	2,596	97,950
DBpedialinks (DL)	Knowledge	18,268,992	136,537,566	9,300	631,415
Wikilink_en (WE)	Knowledge	13,593,032	437,217,424	9,534	1,052,128
Twitter (TW)	Social	52,579,682	1,963,263,821	779,958	3,503,656

In the experiments, all the algorithms above are implemented in C++, following the shared memory model on a single server. Note, however, they can also be easily extended for running on distributed parallel computing platforms and we leave it as a future work. All the experiments are run on a Linux server having dual Intel Xeon(R) Gold 5218R 2.10GHz processors (40 cores, 80 threads) and 255 GB memory, with Ubuntu installed. The number of threads p varies from 1 to 64, and its default value is set to 32.

Remark. Notice that the effectiveness of UDS and DDS algorithms (i.e., the actual approximation ratios of the above algorithms) are evaluated extensively in [6] and [7], respectively, so we omit the detailed comparison of density values in this paper.

B. Evaluation of UDS algorithms

In this section, we evaluate the four parallel UDS algorithms on six real large undirected graphs.

Exp-1: Efficiency comparison. We run all the UDS algorithms using 32 threads on six graphs and depict the efficiency results in Fig. 5. We can observe that our algorithm PKMC is at least 5× and up to 20× faster than the state-of-the-art parallel UDS algorithm PBU. It is also up to 13× faster than the nucleus decomposition algorithm based on the h-index. The main reason is that PBU needs to synchronize vertex and edge information to calculate the density of the remaining subgraph after peeling vertices in each iteration which involves much time cost, while Local needs to compute the core numbers of all vertices, causing a lot of redundant computations. In contrast, our PKMC algorithm does not need to compute the density and also avoids computing the core numbers of all vertices by using fewer iterations as shown by the next experiment. It is also noted that our algorithm is up to two orders of magnitude faster than PFW. The main reason is that PFW searches for the maximum density of subgraphs in a binary search manner by solving linear programmings, which is time-consuming.

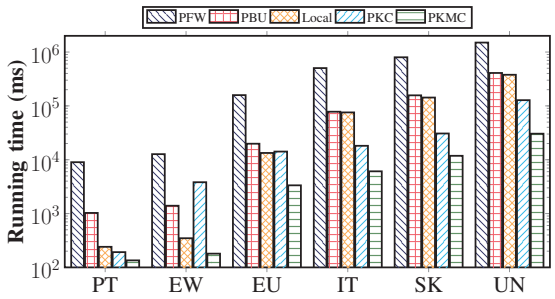


Fig. 5: Efficiency of UDS algorithms on different data sets.

Exp-2: Comparing the number of iterations. We compare the numbers of iterations in three core-based UDS algorithms, including two h-index-based algorithms (i.e., Local and PKMC) and a peeling-based algorithm (i.e., PKC). The numbers of iterations reflect the convergence speed of the core number of the vertices in the k^*-core and the vertices of the entire graph.

TABLE 6: Number of iterations in the core-based algorithms.

	PT	EW	EU	IT	SK	UN
PKC	1,159	17,153	19,332	6,396	9,004	7,133
Local	28	24	860	1,761	3,009	2,396
PKMC	5	4	4	3	3	3

Unlike the h-index-based approaches, PKC computes the core number by peeling vertices in parallel. However, since removing vertices of a specific degree may affect the degrees of other vertices, it can only remove the vertices with a specified degree in each iteration, making the total number of iterations be $k^* + 2$ (the degrees range from 0 to $k^* + 1$). Sariyuce et al. [25] proved that the number of iterations of Local is bounded by the degree levels of the graph which is bounded by the number of vertices, but it is much less than the degree levels and k^* in practice.

Table 6 reports the numbers of iterations of PKC, Local, and PKMC on six graphs. We can see that compared to Local, the number of iterations of PKMC on PT is reduced by 60-70%, while the numbers of iterations on the other five graphs are reduced by more than 99%. This indicates that in the initial stage of Local, the maximum core number and k^*-core have been obtained. The main reason is that the degrees of the four graphs obey the power-law distribution, and the vertices with large degrees are concentrated. In summary, since the number of iterations in PKMC is significantly reduced, it clearly outperforms others in terms of efficiency.

Exp-3: Effect of the number of threads p. In Fig. 6, we show the effect of p on the efficiency by varying it from 1 to 64 on three datasets, and other datasets show the same trend. With the increase of p, the running time of our algorithm PKMC decreases linearly, which shows good parallel scalability. For PKC, its running time is slightly less than PKMC when the threads are less than 8 on PT, this is because the time complexity of PKC is linear to k^*, so

[2]http://konect.cc/networks/
[3]https://law.di.unimi.it/datasets.php

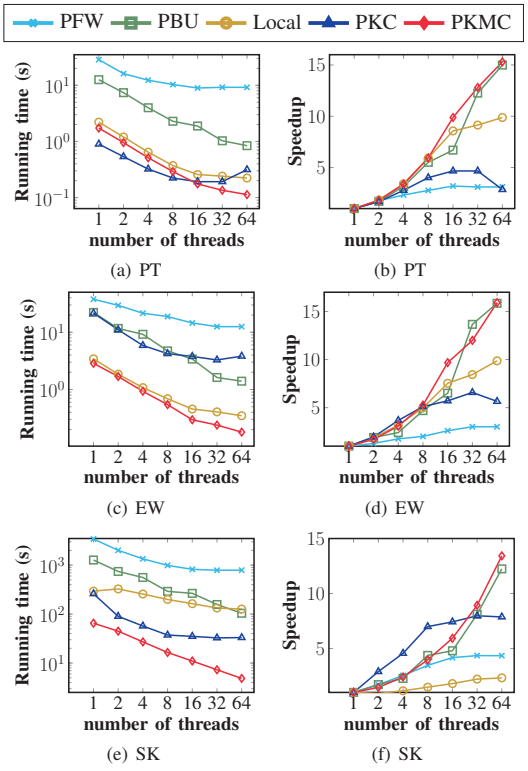


Fig. 6: Effect of the number of threads p.

it is a bit faster when the k^* of the dataset is relatively small. Although it runs faster, the decrease of its running time is not obvious with the increase of p, especially when p increases from 32 to 64 on PT and EW. We conjecture that there are two reasons, (1) there exists a strong dependency on algorithms based on PKC, so the threads that have completed the tasks must synchronize other threads to prevent errors, thereby reducing their parallelism; and (2) the amount of computation in each iteration of the PKC algorithm on these data sets is relatively small, and the creation of threads during OpenMP runtime has an invisible time overhead, so increasing p will cause additional overhead to schedule threads. Local also faces a similar issue since the amount of computation processed decreases as the number of iterations increases.

Exp-4: Scalability test. To evaluate the scalability of all the parallel UDS algorithms, for each graph, we randomly select 20%, 40%, 60%, 80%, and 100% of its edges, and then obtain four subgraphs induced by these edges respectively. Afterwards, we run all the algorithms on these graphs with p=32, and show the efficiency results on SK and UN in Fig. 7. We omit the results on other graphs since their trends are similar. The results show that the running time of all algorithms increases stably as the number of edges increases, and PKMC has the best performance among all algorithms, showing that it achieves good scalability.

C. Evaluation of DDS algorithms

In this section, we evaluate the five algorithms for DDS on six large-scale directed graphs.

Exp-5: Efficiency comparison. We run all the DDS algorithms using 32 threads on five graphs and 4 threads

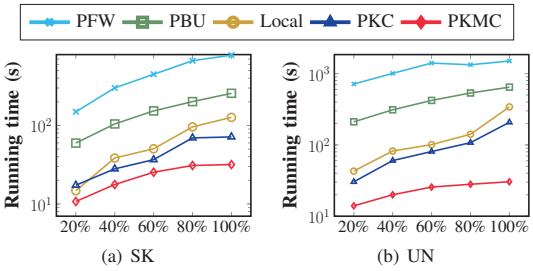


Fig. 7: Scalability of parallel UDS algorithms.

on TW. This is because PXY and PBD cannot run due to memory overflow on TW, since in these algorithms each thread needs to process the entire graph, implying that the memory consumption grows rapidly with the increase of the number of threads. We depict the efficiency results in Fig. 8, where bars touching the upper boundaries mean that the corresponding algorithms cannot finish within 10^5 seconds (around 27.8 hours).

Clearly, both PBS and PFSK cannot obtain results within 10^5 seconds on all datasets, because their time complexities are very high, i.e., $O(n^2(n+m))$ and $O(n(n+m))$. PFW can only obtain results on two datasets, AR and BA, and it is 4 orders of magnitude slower than our algorithm. Besides, PBD is much faster than the greedy-based algorithms PBS and PFKS, since it uses two parameters δ and ϵ to reduce the number of iterations significantly. Nevertheless, its theoretical approximation ratio is $2\delta(1+\epsilon)$=8, leading to a poorer accuracy guarantee. Among all these algorithms, our proposed PWC is up to 30× faster than the state-of-the-art algorithm PXY, since it only needs to decompose the whole graph once, while PBD and PXY need to decompose the graph multiple times.

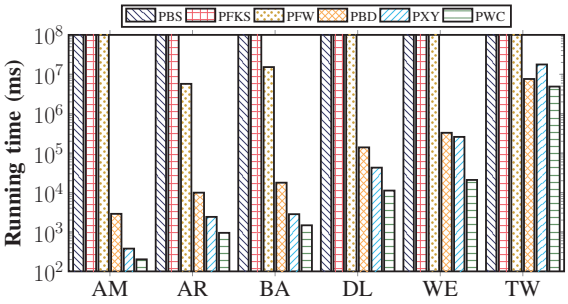


Fig. 8: Efficiency of DDS algorithms on different data sets.

Exp-6: Comparing the sizes of the graphs processed. We analyze the sizes of graphs processed by two core-based algorithms, i.e., PXY and PWC. For PXY, to get $[x^*, y^*]$-core, all possible cn-pairs need to be calculated, and each corresponding $[x, y]$-core needs to be calculated from the entire graph. In PWC, to compute the w^*-induced subgraph, we reduce the size of the graph in each iteration where initially w is set to the maximum degree of the graph.

TABLE 7: The sizes of the graphs processed in PWC and PXY.

	AM	AR	BA	DL	WE	TW
PXY	3,387,388	5,838,041	17,794,839	136,537,566	437,217,424	1,963,263,821
PWC_1	2,751	12,180	193,814	612,308	3,207,622	329,371,005
PWC_{w^*}	2,751	12,180	191,732	612,308	1,865,208	23,032,588
PWC_{D^*}	2,751	12,180	191,732	612,308	1,865,208	22,739,610

Table 7 shows the sizes of the graphs processed by PWC and PXY, where the size means the number of edges in the graph. Here, $\texttt{PWC}_1$ and $\texttt{PWC}_{w^*}$ represent the sizes of the graphs in the first and last iterations respectively, and the size of the densest graph is denoted as $\texttt{PWC}_{D^*}$. Clearly, on AM and AR, since the w-induced subgraphs corresponding to the maximum degree are the $[x^*, y^*]$-core, the results can be obtained immediately. Besides, for other larger graphs (e.g., TW), even the first iteration can reduce the size of the graph by nearly 50%, which greatly improves the efficiency of PWC.

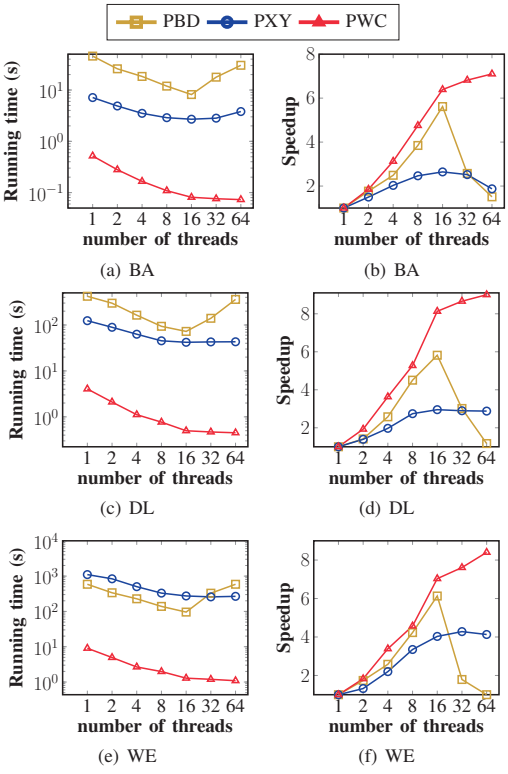


Fig. 9: Effect of the number of threads p.

Exp-7: Effect of the number of threads p. In Fig. 9, we show the effect of p on the efficiency by varying it from 1 to 64 on three datasets, and other datasets show the same trend. Note that since PBS and PFKS cannot obtain results within 10^5 seconds on all graphs and PFW only obtain results on a few graphs, we omit them here. When p=1, our algorithm PWC is 7-10 times faster than PXY. When p increases from 1 to 64, the running time of PWC decreases linearly, while the running time of PBD first decreases and then increases, and achieves the best performance when p=16, since more threads cause thread switching to consume more system resources. In particular, even with 16 threads, the running time of our algorithm is still up to 50× faster than that of PBD. The self-relative speedup of our algorithm is better than that of PXY. The main reason is that in PXY, each cn-pair is calculated independently, where each thread is allocated a specified x (or y) to calculate the corresponding maximum value of y (or x). Since the computational cost of different $x(y)$ values is different, it is easy to cause load imbalance among threads, thereby reducing the parallelism. In addition, on TW, when $p \geq 4$, both PBD and PXY cannot run due to their high memory cost, since both of them run on the entire graph, meaning that larger p will consume more memory.

Exp-8: Scalability test. In this experiment, we evaluate the scalability of PBD, PXY, and PWC on WE and TW. All the subgraphs are sampled in the same way as Exp-4. Since PBD and PXY cannot run due to memory overflow on TW when p>4, we compare the scalability of three parallel algorithms on each graph by setting p=4. As shown in Fig. 10, the time cost of all the algorithms increases when varying the number of edges, which verifies that our algorithm performs well as the graph size grows.

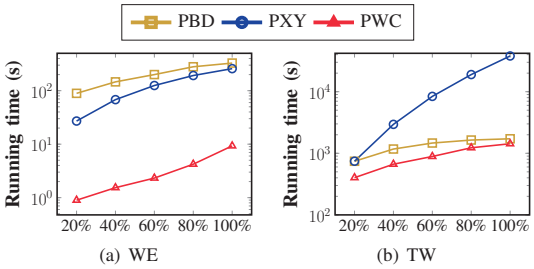


Fig. 10: Scalability of parallel DDS algorithms.

VII. Conclusion

In this paper, we study the densest subgraph discovery problem on both undirected and directed graphs, and develop scalable and efficient parallel algorithms. Specifically, for undirected graphs, we propose an efficient 2-approximation algorithm by computing the k^*-core where k^* is the maximum core number and avoiding computing the core numbers of all the vertices. For directed graphs, we propose a new dense subgraph model, namely w-induced subgraph, and theoretically establish that the $[x^*, y^*]$-core, which offers a 2-approximation solution, can be easily derived from the w^*-induced subgraph, where w^* is the maximum weight. We also propose a fast algorithm to compute the w-induced subgraphs. Experiments on 12 real large graphs show that our proposed algorithms clearly outperform the state-of-the-art algorithms on both undirected and directed graphs, in terms of scalability and efficiency.

In the future, we will implement our algorithms on a distributed computing platform (e.g., GraphX), and test their performance on a cluster. This would be very useful when the graph is too large to be kept by a single machine. Another interesting research direction is to explore the theoretical relationship between other dense subgraphs (e.g., k-truss and k-clique) and densest graph, and then solve DSD problem by exploiting these dense subgraphs.

Acknowledgements

This work was supported by the National Key R&D Program of China under Grant 2020YFB2104000, NSFC under Grants 62202412, 62102341, 62172146, and 62102143, the Natural Science Foundation of Hunan Province under Grant 2022JJ30009, Basic and Applied Basic Research Fund in Guangdong Province under Grant 2022A1515010166, Shenzhen Science and Technology Program ZDSYS20211021111415025, and CUHK-SZ under Grant UDF01002775. Xu Zhou is the corresponding author of this paper.

REFERENCES

[1] A. V. Goldberg, *Finding a maximum density subgraph*. University of California Berkeley, 1984.

[2] R. Kannan and V. Vinay, *Analyzing the structure of large graphs*. Forschungsinst. für Diskrete Mathematik, 1999.

[3] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *International Workshop on Approximation Algorithms for Combinatorial Optimization*. Springer, 2000, pp. 84–95.

[4] S. Khuller and B. Saha, "On finding dense subgraphs," in *International colloquium on automata, languages, and programming*. Springer, 2009, pp. 597–608.

[5] B. Bahmani, R. Kumar, and S. Vassilvitskii, "Densest subgraph in streaming and mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 5, pp. 454–465, 2012.

[6] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin, "Efficient algorithms for densest subgraph discovery," *Proc. VLDB Endow.*, pp. 1719–1732, 2019.

[7] C. Ma, Y. Fang, R. Cheng, L. V. S. Lakshmanan, W. Zhang, and X. Lin, "Efficient algorithms for densest subgraph discovery on large directed graphs," in *SIGMOD 2020*. ACM, pp. 1051–1066.

[8] B. Sun, M. Danisch, T. Chan, and M. Sozio, "Kclist++: A simple algorithm for finding k-clique densest subgraphs in large graphs," *Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[9] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, W. Zhang, and X. Lin, "On directed densest subgraph discovery," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 4, pp. 1–45, 2021.

[10] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," in *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California, USA*, H. J. Karloff, Ed. ACM/SIAM, 1998, pp. 668–677.

[11] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou, "Motifcut: regulatory motifs finding with maximum density subgraphs," in *Proceedings 14th International Conference on Intelligent Systems for Molecular Biology 2006, Fortaleza, Brazil, August 6-10, 2006*, 2006, pp. 156–157.

[12] B. Saha, A. Hoch, S. Khuller, L. Raschid, and X. Zhang, "Dense subgraphs with restrictions and applications to gene annotation graphs," in *Research in Computational Molecular Biology, 14th Annual International Conference, RECOMB 2010, Lisbon, Portugal, April 25-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6044. Springer, 2010, pp. 456–472.

[13] Z. Gyöngyi, P. Berkhin, H. Garcia-Molina, and J. O. Pedersen, "Link spam detection based on mass estimation," in *VLDB, 2006*. ACM, pp. 439–450.

[14] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber, "Piggybacking on social networks," *Proc. VLDB Endow.*, vol. 6, no. 6, pp. 409–420, 2013.

[15] A. Gionis and C. E. Tsourakakis, "Dense subgraph discovery: KDD 2015 tutorial," in *SIGKDD 2015*. ACM, 2015, pp. 2313–2314.

[16] B. A. Prakash, A. Sridharan, M. Seshadri, S. Machiraju, and C. Faloutsos, "Eigenspokes: Surprising patterns and scalable community chipping in large graphs," in *Advances in Knowledge Discovery and Data Mining, 14th Pacific-Asia Conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. Proceedings. Part II*, ser. Lecture Notes in Computer Science, vol. 6119. Springer, 2010, pp. 435–448.

[17] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos, "FRAUDAR: bounding graph fraud in the face of camouflage," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 2016, pp. 895–904.

[18] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, 2003.

[19] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *SIGMOD 2009*. ACM, 2009, pp. 813–826.

[20] F. Zhao and A. K. H. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proc. VLDB Endow.*, vol. 6, no. 2, pp. 85–96, 2012.

[21] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. IEEE Computer Society, 2012, pp. 1049–1060.

[22] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu, "Scalable large near-clique detection in large-scale networks via sampling," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 815–824.

[23] M. Danisch, T.-H. H. Chan, and M. Sozio, "Large scale density-friendly graph decomposition via convex programming," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 233–242.

[24] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature communications*, vol. 7, no. 1, pp. 1–7, 2016.

[25] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *PVLDB*, vol. 12, no. 1, pp. 43–56, 2018.

[26] D. Boob, Y. Gao, R. Peng, S. Sawlani, C. Tsourakakis, D. Wang, and J. Wang, "Flowless: Extracting densest subgraphs without flow computations," in *Proceedings of The Web Conference 2020*, 2020, pp. 573–583.

[27] B. Bahmani, A. Goel, and K. Munagala, "Efficient primal-dual graph algorithms for mapreduce," in *Algorithms and Models for the Web Graph - 11th International Workshop, WAW 2014, Beijing, China, December 17-18, 2014, Proceedings*, vol. 8882. Springer, 2014, pp. 59–78.

[28] H. Su and H. T. Vu, "Distributed dense subgraph detection and low outdegree orientation," in *DISC 2020*, vol. 179. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 15:1–15:18.

[29] C. Chekuri, K. Quanrud, and M. R. Torres, "Densest subgraph: Supermodularity, iterative peeling, and flow," in *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2022, pp. 1531–1555.

[30] C. Tsourakakis, "The k-clique densest subgraph problem," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1122–1132.

[31] R. Samusevich, M. Danisch, and M. Sozio, "Local triangle-densest subgraphs," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2016, pp. 33–40.

[32] S. Sawlani and J. Wang, "Near-optimal fully dynamic densest subgraph," in *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*. ACM, 2020, pp. 181–193.

[33] L. Qin, R.-H. Li, L. Chang, and C. Zhang, "Locally densest subgraph discovery," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 965–974.

[34] N. Tatti and A. Gionis, "Density-friendly graph decomposition," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1089–1099.

[35] E. Galbrun, A. Gionis, and N. Tatti, "Top-k overlapping densest subgraphs," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, pp. 1134–1165, 2016.

[36] R. Dondi, M. M. Hosseinzadeh, G. Mauri, and I. Zoppis, "Top-k overlapping densest subgraphs: approximation algorithms and computational complexity," *Journal of Combinatorial Optimization*, vol. 41, no. 1, pp. 80–104, 2021.

[37] R. Dondi, M. M. Hosseinzadeh, and P. H. Guzzi, "A novel algorithm for finding top-k weighted overlapping densest connected subgraphs in dual networks," *Applied Network Science*, vol. 6, no. 1, pp. 1–17, 2021.

[38] A. Bhaskara, M. Charikar, E. Chlamtac, U. Feige, and A. Vijayaraghavan, "Detecting high log-densities: an o (n $1/4$) approximation for densest k-subgraph," in *Proceedings of the forty-second ACM symposium on Theory of computing*, 2010, pp. 201–210.

[39] A. Bhaskara, M. Charikar, V. Guruswami, A. Vijayaraghavan, and Y. Zhou, "Polynomial integrality gaps for strong sdp relaxations of densest k-subgraph," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2012, pp. 388–405.

[40] N. Bourgeois, A. Giannakos, G. Lucarelli, I. Milis, and V. T. Paschos, "Exact and approximation algorithms for densest k-subgraph," in *International Workshop on Algorithms and Computation*. Springer, 2013, pp. 114–125.

[41] R. Sotirov, "On solving the densest k-subgraph problem on large graphs," *Optimization Methods and Software*, vol. 35, no. 6, pp. 1160–1178, 2020.

[42] F. Bonchi, D. García-Soriano, A. Miyauchi, and C. E. Tsourakakis, "Finding densest k-connected subgraphs," *Discrete Applied Mathematics*, vol. 305, pp. 34–47, 2021.

[43] R. Andersen, "A local algorithm for finding dense subgraphs," *ACM TALG*, vol. 6, no. 4, pp. 1–12, 2010.

[44] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos, "Fraudar: Bounding graph fraud in the face of camouflage," in *SIGKDD*, 2016, pp. 895–904.

[45] Z. Zou, "Polynomial-time algorithm for finding densest subgraphs in uncertain graphs," in *Proceedings of MLG Workshop*, 2013.

[46] A. Miyauchi and A. Takeda, "Robust densest subgraph discovery," in *ICDM*. IEEE, 2018, pp. 1188–1193.

[47] V. Jethava and N. Beerenwinkel, "Finding dense subgraphs in relational graphs," in *ECML PKDD*. Springer, 2015, pp. 641–654.

[48] E. Galimberti, F. Bonchi, and F. Gullo, "Core decomposition and densest subgraph in multilayer networks," in *CIKM*, 2017, pp. 1807–1816.

[49] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano, "Core decomposition in multilayer networks: theory, algorithms, and applications," *TKDD*, vol. 14, no. 1, pp. 1–40, 2020.
[50] L. Chang and L. Qin, *Cohesive subgraph computation over large sparse graphs: algorithms, data structures, and programming techniques*. Springer, 2018.
[51] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.
[52] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, 2012.
[53] A. Conte, T. De Matteis, D. De Sensi, R. Grossi, A. Marino, and L. Versari, "D2k: scalable community detection in massive networks via small-diameter k-plexes," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1272–1281.
[54] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation in bipartite graphs," *The VLDB Journal*, vol. 29, no. 5, pp. 1075–1099, 2020.
[55] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Efficient bitruss decomposition for large-scale bipartite graphs," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 661–672.
[56] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, "Maximum biclique search at billion scale," *Proceedings of the VLDB Endowment*, 2020.
[57] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, "Efficient exact algorithms for maximum balanced biclique search in bipartite graphs," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 248–260.
[58] K. Yu, C. Long, P. Deepak, and T. Chakraborty, "On efficient large maximal biplex discovery," *IEEE Transactions on Knowledge and Data Engineering*, 2021.
[59] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, no. 1, pp. 353–392, 2020.
[60] Y. Fang, K. Wang, X. Lin, and W. Zhang, "Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2829–2838.
[61] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1482–1491.
[62] R. Wang, S. Wang, and X. Zhou, "Parallelizing approximate single-source personalized pagerank queries on shared memory," *VLDB J.*, vol. 28, no. 6, pp. 923–940, 2019.
[63] J. Blanusa, R. Stoica, P. Ienne, and K. Atasu, "Parallelizing maximal clique enumeration on modern manycore processors," in *IEEE IPDPS Workshops 2020*. IEEE, 2020, pp. 211–214.
[64] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 11, pp. 2710–2722, 2020. [Online]. Available: https://doi.org/10.1109/TPDS.2020.3001645
[65] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.
[66] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.
[67] J. E. Hirsch, "An index to quantify an individual's scientific research output," *Proc. Natl. Acad. Sci. USA*, vol. 102, no. 46, pp. 16 569–16 572, 2005.