On Directed Densest Subgraph Discovery

CHENHAO MA, The University of Hong Kong, China
YIXIANG FANG, The Chinese University of Hong Kong, Shenzhen, China
REYNOLD CHENG, The University of Hong Kong, China
LAKS V. S. LAKSHMANAN, The University of British Columbia, Canada
WENJIE ZHANG and XUEMIN LIN, University of New South Wales, Australia

Given a directed graph G, the directed densest subgraph (DDS) problem refers to the finding of a subgraph from G, whose density is the highest among all the subgraphs of G. The DDS problem is fundamental to a wide range of applications, such as fraud detection, community mining, and graph compression. However, existing DDS solutions suffer from efficiency and scalability problems: on a 3,000-edge graph, it takes three days for one of the best exact algorithms to complete. In this article, we develop an efficient and scalable DDS solution. We introduce the notion of $[x, y]$-core, which is a dense subgraph for G, and show that the densest subgraph can be accurately located through the $[x, y]$-core with theoretical guarantees. Based on the $[x, y]$-core, we develop exact and approximation algorithms. We further study the problems of maintaining the DDS over dynamic directed graphs and finding the weighted DDS on weighted directed graphs, and we develop efficient non-trivial algorithms to solve these two problems by extending our DDS algorithms. We have performed an extensive evaluation of our approaches on 15 real large datasets. The results show that our proposed solutions are up to six orders of magnitude faster than the state-of-the-art.

CCS Concepts: • **Theory of computation** → **Graph algorithms analysis**; • **Mathematics of computing** → **Graph algorithms**; *Network flows;*

Additional Key Words and Phrases: Directed graph, densest subgraph discovery

Reynold Cheng and Chenhao Ma were supported by the Research Grants Council of Hong Kong (RGC Projects HKU 17229116 and 17205015), University of Hong Kong (Projects 104005858, 104005994), the Innovation and Technology Commission of Hong Kong (ITF project MRP/029/18), and HKU-TCL Joint Research Center for Artificial Intelligence (Project no. 200009430). Yixiang Fang was supported by NSFC under Grant 62102341 and CUHK-SZ grant UDF01002139. Lakshmanan's research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). Xuemin's research was supported by the ARC DP200101338. Wenjie Zhang's research was supported by the ARC DP200101116.

Authors' addresses: C. Ma, Room 430, Chow Yei Ching Building, The University of Hong Kong, Pokfulam Road, Hong Kong, China; email: chma2@cs.hku.hk; Y. Fang (corresponding author), Room 417d, Dao Yuan Building, The Chinese University of Hong Kong - Shenzhen, 2001 Longxiang Road, Longgang District, Shenzhen, China; email: fangyixiang@cuhk.edu.cn; R. Cheng, Room 303, Chow Yei Ching Building, The University of Hong Kong, Pokfulam Road, Hong Kong, China; email: ckcheng@cs.hku.hk; L. V. S. Lakshmanan, Department of Computer Science, The University of British Columbia, 2366 Main Mall Vancouver, B.C., Canada; email: laks@cs.ubc.ca; W. Zhang, K17, 502, School of Computer Science and Engineering, The University of New South Wales, Sydney NSW 2052, Australia; email: wenjie.zhang@unsw.edu.au; X. Lin, K17, 503, School of Computer Science and Engineering, The University of New South Wales, Sydney NSW 2052, Australia; email: lxue@cse.unsw.edu.

0362-5915/2021/11-ART13 $15.00
https://doi.org/10.1145/3483940

ACM Reference format:
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, Article 13 (November 2021), 45 pages. https://doi.org/10.1145/3483940

1 INTRODUCTION

In emerging systems that manage complex relationship among objects, directed graphs are often used to model their relationships [3, 13, 40, 51]. For example, in online microblogging services (e.g., Twitter and Weibo), the "following" relationships between users can be captured as directed edges [40]. Figure 3(a) depicts a directed graph of the following relationship for five users in a microblogging network. Here, `Alice` has a link to `David` because she is a follower of `David`. As another example, in Wikipedia, each article can be considered as a vertex, and each link between two articles is represented by a directed edge from one vertex to another [13]. As yet another example, the Web can also be viewed as a huge directed graph [3].

In this article, we study the problem of finding the densest subgraph from a directed graph G, which was first proposed by Kannan and Vinay [41]. Conceptually, this ***directed densest subgraph*** **(DDS)** problem aims to find two sets of vertices, S^* and T^*, from G, where (1) vertices in S^* have a large number of outgoing edges to those in T^*, and (2) vertices in T^* receive a large number of edges from those in S^*. To understand DDS, let us explain its usage in fake follower detection [35, 65] and community mining [43]:

• *Fake follower detection* [35, 65] aims to identify fraudulent actions in social networks [35]. Figure 1 illustrates a microblogging network, with edges representing the "following" relationship. By issuing a DDS query, two sets of users, S^* and T^*, are returned. Compared with other users, d (in T^*) has unusually a huge number of followers (a, e, f, g, h) in S^*. It may be worth investigating whether d has bribed the users in S^* for following him/her.

• *Community mining* [43]. In Reference [43], Kleinberg proposed the *hub-authority* concept for finding web communities, based on a hypothesis that a web community is often comprised of a set of *hub pages* and a set of *authority pages*. The hubs are characterized by the presence of a large number of edges to the authorities, while the authorities often receive a large number of links from the hubs. A DDS query can be issued on this network to find hubs and authorities. In Figure 2, for example, websites in S^* can be viewed as hubs providing car rankings and recommendations, while websites in T^* play the roles of authorities as the official websites for well-known automakers.

DDS queries are also useful for *graph compression*, as discussed in Reference [12]. Particularly, Buehrer and Chellapilla [12] proposed to reduce the number of edges by introducing virtual nodes linking to S^* and T^* without sacrificing the connectivity of G. Gionis and Tsourakakis provide more applications of the DDS problem in Reference [31].

Now let us give more details about the DDS query [6, 15, 41, 42]. Given a directed graph $G = (V, E)$ and sets of (not necesssarily disjoint) vertices $S, T \subseteq V$, the density of the directed subgraph induced by (S, T) is the number $|E(S,T)|$ of edges linking vertices in S to the vertices in T over the square root of the product of their sizes, i.e., $\rho(S,T) = \frac{|E(S,T)|}{\sqrt{|S||T|}}$. Based on this definition, the DDS problem aims to find a pair of sets of vertices, S^* and T^*, such that $\rho(S^*, T^*)$ is the maximum among all possible choices of $S, T \subseteq V$. For instance, for the directed graph in Figure 3(a), the DDS is the subgraph induced by $S^* = \{a, b\}$ and $T^* = \{c, d\}$, (see Figure 3(b)). Its density is $\rho^* = \frac{4}{\sqrt{2\times 2}} = 2$. The DDS is exactly what the above applications need. In the DDS definition, we do not impose restrictions on the overlap between S^* and T^*, because it would be interesting to see whether S^* and T^* will overlap with each other by nature. Restricting $S^* \cap T^*$ is hard to set the threshold and may lose some interesting results. We have a case study discussing this issue in Section 9.

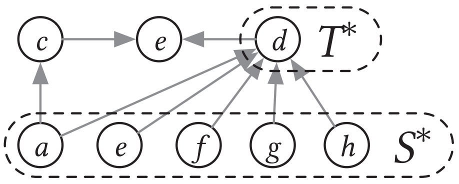


Fig. 1. An example of fake follower detection.

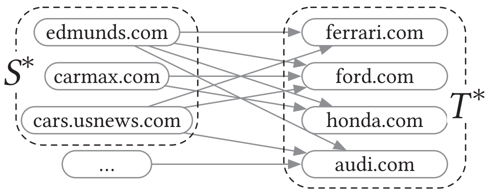
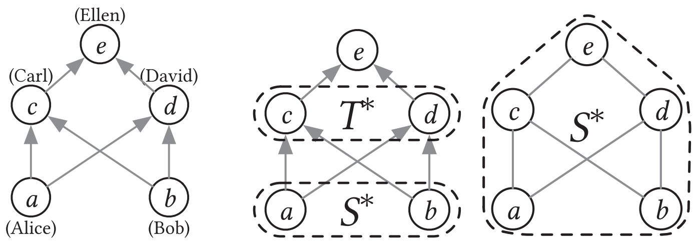


Fig. 2. An example of web community.



(a) A directed graph (b) The DDS (c) Undirected DS

Fig. 3. Illustrating the directed densest subgraph problem (or DDS problem) on the directed graph.

In undirected graphs, the density of a graph $G = (V, E)$ is defined to be $\rho(G) = \frac{|E|}{|V|}$ [32], which is different from that in directed graphs. Hence, finding the **densest subgraph** in undirected graphs (**DS** problem for short) amounts to finding the subgraph with the highest average degree [32]. For example, for the undirected graph G in Figure 3(c), the DS is G itself, and its density is $\frac{6}{5}$, since there is no subgraph with higher density. We can observe that when $S = T$, the density of a directed graph reduces to the classical notion of the density of undirected graphs. Thus, it naturally generalizes the notion of the density of undirected graphs. However, the DDS problem returns two sets, S^* and T^*, which provide the advantage to distinguish different roles of vertices in the above applications.

Impact. The densest subgraph problem lies at the core of large-scale data mining [6]. DDS is an important primitive for real-world applications, such as fake follower detection [35, 65] and community detection [43]. Theoretically, the densest subgraph problem closely connects to fundamental graph problems such as network flow and bipartite matching [69]. Hence, the DDS problem receives much attention from the communities of the database, data mining, theory, and network analysis.

State-of-the-art. For a directed graph $G = (V, E)$, we denote its number of vertices and edges by n and m, respectively. In the literature, both exact [15, 42] and approximation algorithms [6, 15, 41] have been studied. The state-of-the-art exact algorithm is a flow-based algorithm [42], which mainly involves two nested loops: The outer loop enumerates all the n^2 possible values of $\frac{|S|}{|T|}$ ($1 \leq |S|, |T| \leq n$), while the inner loop computes the maximum density by using binary search on a flow network, regarding a specific value of $\frac{|S|}{|T|}$. The inner and outer loops take $O(n^2\sqrt{m})$ and $O(n^2)$ time, respectively, so its overall time complexity is $O(n^4\sqrt{m})$, which is prohibitively expensive for large graphs.

To improve efficiency, approximation algorithms have been developed, the most efficient one being the algorithm in Reference [42], which only costs $O(n + m)$ time, since it iteratively peels the vertex with the smallest indegree or outdegree. However, it was misclaimed to achieve an approximation ratio of 2, as we will show in Section 4.2. Here, the approximation ratio is defined

Table 1. Summary of Exact DDS Algorithms

Algorithm	Time complexity
`LP-Exact` [15]	$\Omega(n^6)$
`Exact` [42]	$O(n^2 \cdot t_{\text{max-flow}})$
`DC-Exact` **(Ours)**	$O(k \cdot t_{\text{max-flow}})$

Note: Theoretically, $k \leq n^2$. But, $k \ll n^2$, in practice. $t_{\text{max-flow}}$ denotes the time complexity of maximum flow computation. In our implementation [2], $t_{\text{max-flow}} = O(n^2\sqrt{m})$.

Table 2. Summary of Approximation DDS Algorithms

Algorithm	Approx. ratio	Time complexity
`KV-Approx` [41]	$O(\log n)$	$O(s^3 n)$
`PM-Approx` [6]	$2\delta(1+\epsilon)$	$O(\frac{\log n}{\log \delta} \log_{1+\epsilon} n(n+m))$
`KS-Approx` [42]	>2	$O(n+m)$
`BS-Approx` [15]	2	$O(n^2 \cdot (n+m))$
`BS-Approx-`δ [15]	2δ	$O(\frac{\log n}{\log \delta}(n+m))$
`Core-Approx` **(Ours)**	2	$O(\sqrt{m}(n+m))$

Note: s is the sample size; $\epsilon > 0$, $\delta > 1$ are the error tolerance parameters. `KS-Approx` [42] made a misclaim that its approximation ratio is 2, which is actually larger than 2.

as the ratio of the density of the DDS over that of the subgraph returned. This makes the algorithm proposed by Charikar in Reference [15] the best available 2-approximation algorithm, and its time complexity is $O(n^2(n+m))$. Clearly, it is still very expensive, warranting more efficient algorithms. Tables 1 and 2 summarize the properties of the exact and approximation algorithms, respectively.

Our technical contributions. To improve the state-of-the-art exact algorithm [42], we optimize its inner and outer loops. Specifically, for the inner loop, we introduce a novel dense subgraph model on directed graphs, namely, $[x, y]$-core, inspired by the k-core [71] on undirected graphs. That is, given two sets of vertices S and T of a graph G, the subgraph induced by S and T in G is the $[x, y]$-core, if each vertex in S has at least x outgoing edges to vertices in T, and each vertex in T has at least y incoming edges from vertices in S. Theoretically, we show that DDS can be accurately located through the $[x, y]$-cores, which are often much smaller than the entire graph. As a result, we can build the flow networks on some $[x, y]$-cores, rather than the entire graph, which greatly improves the efficiency of computing the maximum flow. For the outer loop, we propose a divide-and-conquer strategy, which dramatically reduces the number of values of $\frac{|S|}{|T|}$ examined from n^2 to k. In other words, the number of iterations in the outer loop is k, instead of n^2. Theoretically, $k \leq n^2$, but in practice $k \ll n^2$. Based on the two optimization techniques above, we develop an efficient exact algorithm `DC-Exact`.

In addition, the edges of directed graphs in real applications often carry weights. For example, in the flight network [60] where the airports are represented by vertices and the flights between airports are represented by edges, the weight of an edge denotes the flight frequency between two airports. In the literature, the DS problem on the edge-weighted undirected graphs has been extensively studied [5, 37]. However, the problem of finding the **weighted DDS (or WDDS)** on weighted directed graphs has not been studied yet; thus, this motivates us to study efficient solutions to tackle this problem. To find the WDDS, we extend the $[x, y]$-core to support weighted directed graphs, namely, $[x_w, y_w]$-wcore. Then, we theoretically establish the connection between the $[x_w, y_w]$-wcores and the WDDS. Based on the connection and the divide-and-conquer strategy in DC-Exact, we propose an exact algorithm called WDC-Exact.

We further show that, theoretically, the $[x_w^*, y_w^*]$-wcore, where $x_w^* y_w^*$ is the maximum value among the values of x_w and y_w for all the $[x_w, y_w]$-wcores, provides a 2-approximation solution to the WDDS problem. The cardinality of x_w and y_w in $[x_w, y_w]$-wcores can be extremely large because $x_w, y_w \in \mathbb{R}_+$. Thus, it will be impractical to emumrate all possible $[x_w, y_w]$-wcores. To enable the efficient computation of the $[x_w^*, y_w^*]$-wcore, we propose a fast approximation algorithm, denoted by WCore-Approx, based on a stair-climbing strategy. For the unweighted case, we introduce several novel core number pair concepts. Based on the new concepts, we propose an optimized algorithm for unweighted graphs, called Core-Approx, which completes in $O(\sqrt{m} \cdot (n + m))$ time. Therefore, compared to existing 2-approximation algorithms, it has the lowest time complexity.

Furthermore, in the aforementioned applications, the directed graphs are inherently dynamic. For example, new users join the following/follower relationship network of Twitter, and the following/follower relationships also evolve frequently, which makes the whole network update dynamically. However, the DS problem in dynamic undirected graphs has been extensively studied [5, 9, 18, 37, 68]. A straightforward method to address this problem is to recompute the DDS from scratch each time whenever a new update is made on the network, which obviously is time-consuming, rendering it impractical, especially for large graphs whose updates are highly frequent. To improve the efficiency, we propose novel efficient algorithms for maintaining the 2-approximation DDS under edge updates, since the vertex insertion (respectively, deletion) can be treated as a sequence of edge insertions (respectively, deletions). The algorithms are free of extra memory usage due to their index-free property and unlikely to iterate the whole graph by exploiting the locality of the DDS, so they are able to process large graphs efficiently.

We have experimentally compared our proposed DDS solutions with the state-of-the-art solutions on more than 10 real graphs, where the largest one consists of around 2 billion edges. The results show that for the exact DDS algorithms, our proposed DC-Exact is over six orders of magnitude faster than the baseline algorithm on a graph with around 6,500 vertices and 51,000 edges. Besides, for approximation DDS algorithms, our proposed Core-Approx can scale well on billion-scale graphs and is also up to six orders of magnitude faster than the existing 2-approximation algorithm. Furthermore, the experimental results on dynamic directed graphs show that our proposed DDS maintenance algorithms could be up to five orders of magnitude faster than recomputing from scratch. In addition, we have evaluated our exact and approximation algorithms for the WDDS problem and conduct a case study about fake review detection using the exact algorithm WDC-Exact, which shows that the WDDS on directed graphs can provide valuable insights towards detecting fraudulent behaviors.

Outline. The rest of the article is organized as follows: We review the related work in Section 2. In Section 3, we formally present the DDS problem and the WDDS problem. Section 4 reviews the state-of-the-art DDS algorithms and discusses their limitations. Section 5 introduces the formal definitions of the $[x, y]$-core and the $[x_w, y_w]$-wcore and their density bounds in unweighted and weighted graphs, respectively. We present our exact DDS and WDDS algorithms in Section 6 and approximation DDS and WDDS algorithms in Section 7. Section 8 presents how to maintain the DDS over dynamic directed graphs. Experimental results are presented in Section 9. We conclude the article in Section 10.

2 RELATED WORK

The densest subgraph can be regarded as one type of cohesive subgraphs. Gionis and Tsourakakis provide a comprehensive study over the applications on the densest subgraph discovery [31]. Other related topics contain k-core [71, 78], k-truss [17, 39], cliques and motifs [36, 49, 50, 52], as well as community search [19–26, 38, 77] based on k-core and k-truss. In the following, we focus on two groups of work on densest subgraph discovery on undirected graphs [32] and directed graphs [41], respectively.

Densest subgraph discovery on undirected graphs. In Reference [32], Goldberg introduced the **densest subgraph discovery problem (DS problem)**, which aims to find the subgraph whose edge-density is the highest among all the subgraphs where the edge-density of a graph $G = (V, E)$ is defined as $\frac{|E|}{|V|}$, and proposed a max-flow-based algorithm to compute the exact densest subgraph. Tsourakakis [74] and Mitzenmacher et al. [56] generalized the above edge-density as clique-density and developed efficient exact algorithms for finding the corresponding DS. Sun et al. [72] studied the clique-density based DS problem with larger cliques and provided a simple but efficient near-optimal algorithm. Recently, Fang et al. [27] have proposed efficient DS algorithms by exploiting k-cores that are able to find the densest subgraphs for a wide range of graph density definitions such as edge-density, clique-density, and pattern-density. Generally, the exact DS algorithms [27, 32, 74] work well on small or moderate-size graphs, but they are inefficient for processing large graphs, as shown in Reference [27]. To remedy this issue, several efficient approximation algorithms have been developed. In Reference [15], Charikar developed a 2-approximation algorithm that takes linear time cost. Fang et al. [27] improved the algorithm by exploiting k-cores. In Reference [6], Bahmani et al. designed a parameterized approximation algorithm, which achives an approximation of $2(1+\epsilon)$ where $\epsilon > 0$. Boob et al. [11] present an iterative peeling algorithm to output near-optimal solutions fast by adding a few more passes to Charikar's greedy algorithm. Besides, many variants of the DS problem have been studied. In Reference [8], Bhaskara et al. studied the densest k-subgraph problem, where a k-subgraph means a subgraph consisting of k vertices. Qin et al. [66] developed solutions for finding the top-k locally densest subgraphs. In Reference [56], Mitzenmacher et al. studied the (p, q)-biclique densest subgraph problem on bipartite graphs. Tsourakakis et al. [75] developed algorithms for discovering quasi-cliques with quality guarantees. Recently, Tatti and Gionis [73] and Danisch et al. [16] have studied the topic of density-friendly graph decomposition, which decomposes a graph into a chain of subgraphs, where each subgraph is nested within the next one and the inner one is denser than the outer ones.

For undirected graphs whose edges are associated with weights, Hu et al. [37] and Angel et al. [5] have developed some efficient algorithms to find the densest subgraphs. Hu et al. [37] extended Goldberg's max-flow-based algorithm [32] to support weighted undirected graphs. Angel et al. [5] studied the efficient maintenance of dense subgraphs under streaming edge weight updates. Tsourakakis et al. [76] shows that the DS problem for weighted graphs is polynomial-time solvable when the weights are non-negative and is NP-hard for negative weights. Besides, there are some works [5, 9, 18, 37, 68, 69] about maintaining the densest subgraphs on dynamic graphs, where every update comes online and can either insert an edge to, or remove an existing edge from the graph. Angel et al. [5] proposed a DS maintenance algorithm for processing streaming edge weight updates in a weighted undirected graph. Epasto et al. [18] dynamically maintained the approximate DS where edge insertions are adversarial and deletions are randomly picked. Hu et al. [37] considered the dynamic maintenance of approximate DS in weighted hypergraphs, where an edge can join any number of vertices. Bhattacharya et al. [9] proposed a fully dynamic $(4 + \epsilon)$-approximation algorithm that uses sublinear space and poly-logarithmic amortized time per update, where $\epsilon > 0$. Sawlani and Wang [69] gave the fully dynamic algorithm that maintains a $(1 + \epsilon)$-approximation densest subgraph in worst-case time $\text{poly}(\log n, \epsilon^{-1})$ per update.

Densest subgraph discovery on directed graphs. Kannan and Vinay [41] were the first to introduce the notion of density and the problem of the densest subgraph on directed graphs

(DDS problem). In Reference [15], Charikar developed an exact algorithm for this problem, which completes in polynomial time cost by solving $O(n^2)$ linear programs. Later, in Reference [42], Khuller and Saha proposed a flow-based algorithm, which also takes polynomial time cost. Table 1 summarizes the time complexities of these exact algorithms. Nevertheless, all these exact algorithms are computationally expensive for large graphs, so researchers have developed efficient approximation algorithms. In Reference [41], Kannan and Vinay proposed an $O(\log n)$-approximation algorithm to compute the densest subgraph. In Reference [15], Charikar designed a 2-approximation algorithm with a time complexity of $O(n^2 \cdot (n+m))$. In Reference [42], Khuller and Saha presented a linear approximation algorithm and claimed that it achieves an approximation of 2. Unfortunately, as we shall show in Section 4.2 (Example 4.1), the claim is incorrect. In Reference [6], Bahmani et al. developed a $2(1+\epsilon)$-approximation algorithm based on the streaming model ($\epsilon > 0$). Moreover, in Reference [4], Andersen used the density definition by Kannan and Vinay for finding local dense bipartite graphs given a vertex to be contained. Table 2 summarizes the approximation ratios and time complexities of these approximation algorithms. The maintenance algorithm for Reference [9] can be extended to the directed graphs with an approximation ratio of $8+\varepsilon$, while the approximation ratio of our proposed dynamic maintenance algorithm is 2. Besides, to the best of our knowledge, no previous work has studied the DDS problem on weighted directed graphs, so our work is the first to study the problem.

An earlier version of this article is presented in Reference [53]. Compared to the old version, this article has several newly added contributions, which are summarized as follows:

(1) We develop non-trivial DDS maintenance algorithms, which are able to efficiently maintain the 2-approximation DDS on dynamic directed graphs in Section 8;
(2) We design both efficient exact and approximation algorithms for finding the WDDS on weighted directed graphs in Section 6 and Section 7 based on the new concept, $[x_w, y_w]$-wcore in Section 5.2;
(3) We evaluate the algorithms for maintaining the DDS on dynamic graphs and finding the WDDS on weighted directed graphs in terms of efficiency and effectiveness in Section 9, and we also conduct a case study in Section 9.

3 PROBLEM DEFINITIONS

In this section, we give the formal definitions of the densest subgraph on unweighted directed graphs and weighted directed graphs, respectively. Table 3 lists the notations used in this paper.

3.1 The DDS Problem

Let $G = (V, E)$ be a directed graph, $n = |V|$ and $m = |E|$ be the number of vertices and edges in G, respectively. Given two sets of vertices, $S, T \subseteq V$, which are not necessarily disjoint, we use $E(S, T)$ to denote the set of all the edges linking their vertices, i.e., $E(S, T) = E \cap (S \times T)$. The subgraph induced by S, T, and $E(S, T)$ is called an (S, T)-induced subgraph, denoted by $G[S, T]$. For a vertex $v \in G$, we use $d^+_G(v)$ and $d^-_G(v)$ to denote its outdegree and indegree in G, respectively. Next, we formally present the density of a directed graph [41] and the problem of Directed Densest Subgraph discovery, or DDS problem. Unless mentioned otherwise, all the graphs mentioned later in this article are directed graphs.

Definition 3.1 (Density of a Directed Graph). Given a directed graph $G = (V, E)$ and two sets of vertices $S, T \subseteq V$, the density of the (S, T)-induced subgraph $G[S, T]$ is

$$\rho(S,T) = \frac{|E(S,T)|}{\sqrt{|S| \cdot |T|}}. \tag{1}$$

Table 3. Notations and Meanings

Notation	Meaning
$G = (V, E)$	a directed graph with vertex set V and edge set E
$G_w = (V, E, W)$	a weighted directed graph with vertex set V, edge set E, and edge weight set W
n, m	$n = \|V\|$, $m = \|E\|$
$H = G[S, T]$	the subgraph induced by S and T in G
$E(S, T)$	the edges induced by S and T in G
$d^+_G(v)$, $d^-_G(v)$	the outdegree and indegree of a vertex $v \in G$ respectively
$\rho(S, T)$	the density of the (S, T)-induced subgraph
$D = G[S^*, T^*]$	the densest subgraph D in G
ρ^*	$\rho^* = \max_{S,T \subset V}\{\rho(S, T)\} = \rho(S^*, T^*)$
$\widetilde{D} = G[\widetilde{S^*}, \widetilde{T^*}]$	the approximate densest subgraph in G
$\widetilde{\rho^*} = \rho(\widetilde{S^*}, \widetilde{T^*})$	the density of $\widetilde{D}$
$F = (V_F, E_F)$	a flow network with node set V_F and edge set E_F

Definition 3.2 (DDS). Given a directed graph $G = (V, E)$, a directed densest subgraph (DDS) D is the (S^*, T^*)-induced subgraph, whose density is the highest among all the possible (S, T)-induced subgraphs.

***Problem* 1** (DDS Problem [6, 15, 31, 41, 42, 53, 54])**:** Given a directed graph $G = (V, E)$, return a DDS[1] $D = G[S^*, T^*]$ of G.

Example 3.3. Consider the directed graph in Figure 3(a). Its DDS $D = G[S^*, T^*]$ is the subgraph highlighted in Figure 3(b), where $S^* = \{a, b\}$ and $T^* = \{c, d\}$, since its density $\rho(S^*, T^*) = \frac{4}{\sqrt{2\times 2}} = 2$ is higher than the density of any other (S, T)-induced subgraphs. For instance, if we let $S = V$ and $T = V$, then we get a (V, V)-induced subgraph $H = G[V, V]$, and its density is $\rho(V, V) = \frac{6}{\sqrt{5\times 5}} = \frac{6}{5}$, which is less than 2.

3.2 The WDDS Problem

A weighted directed graph is a triplet $G_w = (V, E, W)$, where V and E denote the sets of vertices and edges, respectively ($E \subseteq V \times V$), and $W = \{w_e | e \in E\}$ contains the edge weights assigned to the edges. Here, we restrict $w_e \geq 0$. For a vertex $v \in G_w$, we use $w^-_{G_w}(v) = \sum_{e=(u,v)\in E} w_e$ and $w^+_{G_w}(v) = \sum_{e=(v,u)\in E} w_e$ to denote its weighted indegree and weighted outdegree in G_w, respectively. Inspired by the formulation of density on weighted undirected graphs [57], we introduce the definition of density for weighted directed graph, also called weighted-density, as follows:

Definition 3.4 (Weighted-density). Given a weighted graph $G_w = (V, E, W)$ and two vertex sets $S_w, T_w \subseteq V$, the weighted-density of the (S_w, T_w)-induced subgraph $H = G_w[S_w, T_w]$ is

$$\rho_w(S_w, T_w) = \frac{\sum_{e\in E(S_w,T_w)} w_e}{\sqrt{|S_w| \cdot |T_w|}}. \tag{2}$$

Definition 3.5 (WDDS). Given a weighted directed graph $G_w = (V, E, W)$, a **weighted directed densest subgraph (WDDS)** D_w is the (S^*_w, T^*_w)-induced subgraph, whose weighted-density is the highest among all the possible (S_w, T_w)-induced subgraphs.

***Problem* 2** (WDDS Problem)**:** Given a weighted directed graph $G_w = (V, E, W)$, return a WDDS[2] $D_w = G_w[S^*_w, T^*_w]$ of G_w.

[1]There might be several directed densest subgraphs of a graph, and our algorithm will find one of them.
[2]There might be several weighted directed densest subgraphs of a graph, and our algorithm will find one of them.

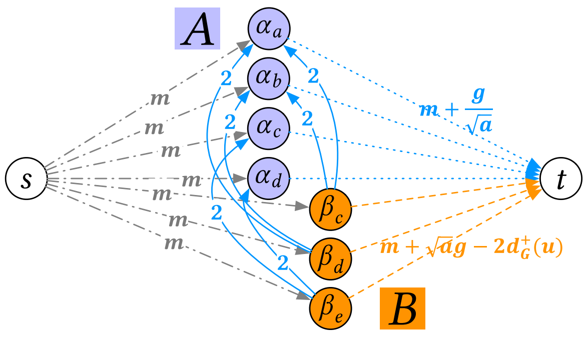


Fig. 4. Illustrating the flow network.

Example 3.6. Figure 7(a) depicts a weighted graph G_w, where the weight of the edge (d, c) is 2, and the weights of others are 1. Figure 7(b) depicts the WDDS of G_w, i.e., $D_w = G_w[S^*_w, T^*_w]$, where $S^*_w = \{a, b, d\}$ and $T^*_w = \{c, d\}$, and its weighted-density is $\rho_w(S^*_w, T^*_w) = \frac{\sum_{e \in E(S^*_w, T^*_w)} w_e}{\sqrt{|S^*_w||T^*_w|}} = \frac{6}{\sqrt{6}} = \sqrt{6}$.

4 STATE-OF-THE-ART DDS ALGORITHMS

In this section, we review the state-of-the-art exact and approximation DDS algorithms [15, 42]. Note that in this article, the approximation ratio of an approximation algorithm is defined as the ratio of the maximum density over the density of the subgraph returned.

4.1 The Exact Algorithm

The state-of-the-art exact algorithm [42] computes the DDS by solving a maximum flow problem, which generally follows the same paradigm of the exact algorithm [32] of finding the densest subgraphs on undirected graphs. We denote this algorithm by Exact. A flow network [34] is a directed graph $F = (V_F, E_F)$, where there is a source node[3] s, a sink node t, and some intermediate nodes; each edge has a capacity, and the amount of flow on an edge cannot exceed the capacity of the edge. The maximum flow of a flow network equals the capacity of its minimum st-cut, $\langle \mathcal{S}, \mathcal{T} \rangle$, which partitions the node set V_F into two disjoint sets, $\mathcal{S}$ and $\mathcal{T}$, such that $s \in \mathcal{S}$ and $t \in \mathcal{T}$.

Algorithm 1 presents Exact. It first enumerates all the possible values of $a = \frac{|S|}{|T|}$ (Algorithm 2). Then, for each a, it guesses the value g of the maximum density via a binary search (lines 2–5). After that, for each pair of a and g, it builds a flow network and runs the maximum flow algorithm to compute the minimum st-cut $\langle \mathcal{S}, \mathcal{T} \rangle$ (lines 6–11). Note that if $\mathcal{S} \backslash \{s\} \neq \emptyset$, then there must be an (S, T)-induced subgraph such that its density is at least g. If such a subgraph exists and g is larger than ρ^*, then we update the DDS D and its corresponding density ρ^* (Algorithm 11). Note A and B are two node sets contained in V_F (cf. Algorithm 15 and Figure 4). To build the flow network, it first creates a set V_F of nodes (lines 14–15), and then adds directed edges with different capacities between these nodes (lines 16–20). For example, for the direct graph depicted in Figure 3(a), we can build a flow network as shown in Figure 4. Note α_e in A (respectively, β_a and β_b in B) is (respectively, are) omitted in Figure 4 for simplicity, because of the lack of outgoing edges for e (respectively, incoming edges for a and b).

Limitations. In Algorithm 1, the number of possible values of a is n^2, and for each a, the while loop of binary search will have $O(\log n)$ iterations. Using a parametric max-flow algorithm [29], the total time cost required for the $O(\log n)$ iterations is the same as the cost of one maximum

[3]We use "node" to mean "flow network node" in this article.

ALGORITHM 1: Exact [42]

```
   Input  : $G = (V, E)$
   Output : The exact DDS $D = G[S^*, T^*]$
1  $\rho^* \leftarrow 0$;
2  foreach $a \in \{\frac{n_1}{n_2} \mid 0 < n_1, n_2 <= n\}$ do
3      $l \leftarrow 0, r \leftarrow \max_{u \in V}\{d^-_G(u), d^+_G(u)\}$;
4      while $r - l \geq \frac{\sqrt{n}-\sqrt{n-1}}{n\sqrt{n-1}}$ do
5          $g \leftarrow \frac{l+r}{2}$;
6          $F = (V_F, E_F) \leftarrow$ BuildFlowNetwok($G, a, g$);
7          $\langle \mathcal{S}, \mathcal{T} \rangle \leftarrow$ Min-ST-Cut($F$);
8          if $\mathcal{S} = \{s\}$ then $r \leftarrow g$ ;
9          else
10             $l \leftarrow g$;
11             if $g > \rho^*$ then $D \leftarrow G[\mathcal{S} \cap A, \mathcal{S} \cap B]$, $\rho^* = g$;
12 return $D$;
13 Function BuildFlowNetwok($G = (V, E), a, g$):
14     $A \leftarrow \{\alpha_u | u \in V\}, B \leftarrow \{\beta_u | u \in V\}, E_F \leftarrow \emptyset$;
15     $V_F \leftarrow \{s\} \cup A \cup B \cup \{t\}$;
16     for $\alpha_u \in A$ do add $(s, \alpha_u)$ to $E_F$ with capacity $m$;
17     for $\beta_u \in B$ do add $(s, \beta_u)$ to $E_F$ with capacity $m$;
18     for $\alpha_u \in A$ do add $(\alpha_u, t)$ to $E_F$ with capacity $m + \frac{g}{\sqrt{a}}$;
19     for $\beta_u \in B$ do add $(\beta_u, t)$ to $E_F$ with capacity $m + \sqrt{a}g - 2d^-_G(u)$;
20     for $(u, v) \in E$ do add $(\beta_v, \alpha_u)$ to $E_F$ with capacity 2;
21     return $F = (V_F, E_F)$
```

flow computation within a constant factor. We use the push-relabel algorithm based on the highest label node selection rule [2], which is generally regarded as the benchmark for maximum flow algorithms with a time complexity of $O(n^2\sqrt{m})$ [33], for the minimum st-cut computation.[4] Consequently, the total time complexity of Exact is $O(n^4\sqrt{m})$, which is very inefficient on even small graphs. For example, our later experiments show that Exact takes more than two days to find the DDS on a graph with ~1,200 vertices and ~2,600 edges. The sources of inefficiency are three-fold: First, it needs to check all the n^2 values of a, which is very costly. Second, the flow network F is always built on the entire graph in each iteration, while the DDS is often a small subgraph of G. Third, the initial lower and upper bounds of ρ^* are not very tight. Therefore, there is room for improving its efficiency.

4.2 Approximation Algorithms

The state-of-the-art approximation algorithm KS-Approx [42] follows the peeling paradigm. Specifically, it works in n rounds. In each round, it removes the vertex whose indegree or outdegree is the smallest and recomputes the density of the residual graph. Finally, the subgraph whose density is the highest is returned. Algorithm 2 outlines these steps.

[4]The push-relabel algorithm can also be replaced by other max-flow algorithms, e.g., Orlin's algorithm, whose time complexity is $O(nm)$ [62].

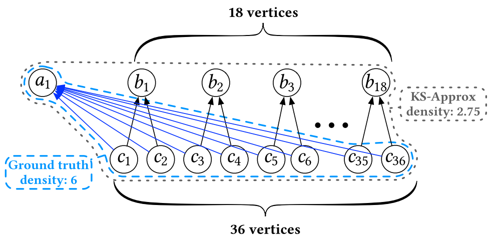


Fig. 5. A counter-example for KS-Approx.

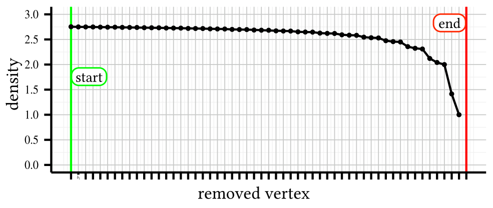


Fig. 6. Running steps by KS-Approx.

ALGORITHM 2: KS-Approx [42]

```
    Input   : $G = (V, E)$
    Output  : An approximate DDS $\widetilde{D}$
1   $\widetilde{\rho^*} \leftarrow 0, \widetilde{D} \leftarrow \emptyset, S \leftarrow V, T \leftarrow V$;
2   while $|E| > 0$ do
3       if $\rho(S,T) > \widetilde{\rho^*}$ then
4           $\widetilde{\rho^*} \leftarrow \rho(S,T), \widetilde{D} \leftarrow (S,T)$;
5       $u_+ \leftarrow \arg\min_u d^-_G(u), u_- \leftarrow \arg\min_u d^+_G(u)$;
6       if $d^-_G(u_+) \le d^+_G(u_-)$ then
7           $E \leftarrow E \setminus \{(v, u_+)|v \in S\}, T \leftarrow T \setminus \{u_+\}$;
8       else
9           $E \leftarrow E \setminus \{(u_-, v)|v \in T\}, S \leftarrow S \setminus \{u_-\}$;
10  return $\widetilde{D}$;
```

It was claimed in Reference [42] that KS-Approx achieves an approximation ratio of 2. Unfortunately, as shown in the following counter-example, their claim is incorrect.[5] Specifically, Example 4.1 shows that KS-Approx may report results whose approximation ratios are larger (i.e., worse) than 2.

Example 4.1. In Figure 5, the graph has three sets of vertices, i.e., $\{a_1\}$, $\{b_i|1 \le i \le 18\}$, and $\{c_i|1 \le i \le 36\}$. Note that a_1 has 36 incoming edges from $c_1, c_2, \ldots, c_{36}$. For each vertex b_i ($i \in [1, 18]$), it has 2 incoming edges from c_{2i-1} and c_{2i}. The exact DDS is the subgraph induced by a_1, $c_1, c_2, \ldots, c_{36}$, and its density is 6.

Figure 6 provides a step-by-step breakdown for KS-Approx on the graph in Figure 5. In the beginning, $S = \{c_i|1 \le i \le 36\}$ and $T = \{a_1\} \cup \{b_i|1 \le i \le 18\}$ (the vertices with no outgoing edge in S and no incoming edge in T are eliminated for simplicity). $\rho(S,T) = 2.7530$. Then, b_1 is removed from T based on the condition in Line 6 of Algorithm 2, and $\rho(S,T) = 2.7499$. Next c_1 is removed from S, and $\rho(S,T)$ becomes 2.7490. c_2 is removed from S afterward, and $\rho(S,T)$ becomes 2.7487. Along with more vertices deleted, the density $\rho(S,T)$ declines gradually. After a_1 is removed from T, the algorithm ends as T becomes empty, and no edge is left in the graph. Thus, KS-Approx will return the whole graph as the approximate DDS, whose density is 2.75. Hence, the actual approximation ratio is $\frac{6}{2.75} > 2$, which contradicts the claim that it is a 2-approximation algorithm.

[5]The authors of Reference [42] have confirmed that the approximation ratio of KS-Approx was misclaimed.

ALGORITHM 3: BS-Approx [15]

```
Input  : $G = (V, E)$
Output : An approximate DDS $\widetilde{D}$
1  $\widetilde{\rho}^* \leftarrow 0, \widetilde{D} \leftarrow \emptyset$;
2  foreach $a \in \{\frac{n_1}{n_2} | 0 < n_1, n_2 <= n\}$ do
3      $S \leftarrow V, T \leftarrow V$;
4      while $S \neq \emptyset \wedge T \neq \emptyset$ do
5          if $\rho(S,T) > \widetilde{\rho}^*$ then $\widetilde{D} \leftarrow G[S,T]$, $\widetilde{\rho}^* \leftarrow \rho(S,T)$;
6          $u \leftarrow \arg\min_{u \in S} d^+_G(u)$;
7          $v \leftarrow \arg\min_{v \in T} d^-_G(v)$;
8          if $\sqrt{a} \cdot d^+_G(u) \leq \frac{1}{\sqrt{a}} \cdot d^-_G(v)$ then $S \leftarrow S \setminus \{u\}$;
9          else $T \leftarrow T \setminus \{v\}$;
10 return $\widetilde{D}$;
```

Why KS-Approx fails? KS-Approx is supported by Theorem 2 in Reference [42]. Theorem 2 requires that there is an iteration that $\forall u \in S, d^+_{G[S,T]}(u) \geq \lambda_o = |E(S^*, T^*)| \cdot (1 - \sqrt{1 - \frac{1}{|S^*|}})$ and $\forall v \in T, d^-_{G[S,T]}(v) \geq \lambda_i = |E(S^*, T^*)| \cdot (1 - \sqrt{1 - \frac{1}{|T^*|}})$. In Example 4.1, $S^* = \{c_i | 1 \leq i \leq 36\}$ and $T^* = \{a_1\}$. Thus, $\lambda_o = 0.5035$ and $\lambda_i = 36$. By reviewing the iterations of KS-Approx over the counter-example, we can find such condition cannot be guaranteed simultaneously.

Furthermore, say we enlarge the graph of Figure 5 to a graph consisting of three sets of vertices, i.e., $\{a_1\}$, $\{b_i | 1 \leq i \leq 2\mu^2\}$, and $\{c_i | 1 \leq i \leq 4\mu^2\}$. The ground-truth DDS will be the subgraph induced by $\{c_i\}$ and $\{a_1\}$, whose density is 2μ. However, the result returned by KS-Approx will still be the whole graph, i.e., the subgraph induced by $\{c_i\}$ and $\{a_1\} \cup \{b_i\}$, whose density is $\frac{4\mu}{\sqrt{2\mu^2+1}}$. As a result, the approximation ratio of KS-Approx is $\frac{\sqrt{2\mu^2+1}}{2}$, which means that the approximation ratio of KS-Approx is proportional to the value of μ and *cannot be bounded by any constant value.*

During our recent communication with the authors of Reference [42], they have proposed a fix (called FKS-Approx), which is a correct 2-approximation algorithm, but costs $O(n \cdot (n + m))$ time. The details about FKS-Approx are provided in Section A.

Since KS-Approx is not a 2-approximation algorithm, the most accurate published approximation algorithm is BS-Approx [15], which is able to correctly find a 2-approximation result. We outline its steps in Algorithm 3. Similar to Exact, BS-Approx enumerates all the possible values of $a = \frac{|S|}{|T|}$ (Algorithm 2), and for each specific a, it iteratively removes the vertex with the minimum degree from S or T based on a predefined condition (Algorithm 8), and then updates S and T, as well as the approximate DDS $\widetilde{D}$ (lines 4–9).

Limitations. Clearly, the time complexity of BS-Approx is $O(n^2 \cdot (n + m))$, where the main overhead comes from the loop of enumerating all the n^2 values of a. Although it is much faster than Exact, it is still inefficient for large graphs. As shown in our experiments later, on a graph with about 3,000 vertices and 30,000 edges, it takes around three days to compute the DDS. Therefore, it is imperative to develop more efficient approximation algorithms.

Besides, to the best of our knowledge, the DDS maintenance problem and the WDDS problem have not been studied in the literature, and also it is not clear how to extend existing solutions above to solve these two problems, calling for the development of novel efficient solutions.

5 A NOVEL CORE MODEL ON DIRECTED GRAPHS

In this section, we introduce a novel core model, namely, $[x, y]$-core, by extending the classic k-core [71] for directed graphs. As we will show, $[x, y]$-cores are useful in locating the DDS in both exact and approximation algorithms. We then extend this core model for weighted directed graphs. We also derive the upper and lower bounds on the density of the cores.

5.1 k-core and $[x, y]$-core

We first review the definition of k-core on undirected graphs.

Definition 5.1 (k-core [7, 71]). Given an undirected graph G and an integer k ($k \geq 0$), the k-core, denoted by $\mathcal{H}_k$, is the largest connected subgraph of G, such that $\forall v \in \mathcal{H}_k$, $deg_{\mathcal{H}_k}(v) \geq k$.

A k-core[6] has some interesting properties [7, 71]: (1) k-cores are "nested": Given two non-negative integers i and j, if $i < j$, then $\mathcal{H}_j \subseteq \mathcal{H}_i$; and (2) computing all the k-cores of a graph, known as k-core decomposition, can be done in linear time [7].

Definition 5.2 ($[x, y]$-core). Given a directed graph $G = (V, E)$, the $[x, y]$**-core** is the largest (S, T)-induced subgraph $H = G[S, T]$, which satisfies:

(1) $\forall u \in S, d^+_H(u) \geq x$ and $\forall v \in T, d^-_H(v) \geq y$;
(2) $\nexists H'$, s.t. $H \subset H'$ and H' satisfies (1).

We call $[x, y]$ the **core number pair** of the $[x, y]$-core, abbreviated as **cn-pair**.

Example 5.3. The subgraph induced by (S^*, T^*), i.e., $D = G[S^*, T^*]$ in Figure 3(b) is a $[2, 2]$-core. $H = G[\{a, b, c, d\}, \{c, d, e\}]$ is a $[1, 2]$-core, and D is contained in H.

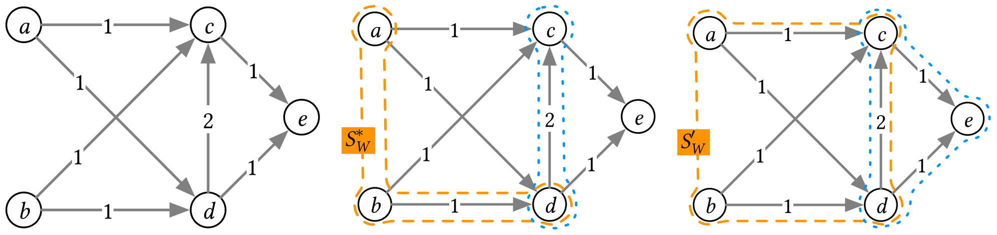
Like the classic k-core, the $[x, y]$-core also has some interesting properties, derived from Definition 5.2.

Lemma 5.4 (Nested Property). *An $[x, y]$-core is contained by an $[x', y']$-core, where $x \geq x' \geq 0$ and $y \geq y' \geq 0$. In other words, if $H = G[S, T]$ is an $[x, y]$-core, then there must exist an $[x', y']$-core H'=$G[S', T']$, such that $S \subseteq S'$ and $T \subseteq T'$.*

Given a pair of x and y, to compute the $[x, y]$-core, we can borrow the idea of k-core decomposition [7]; that is, we can first initialize an (S, T)-induced subgraph such that $S = T = V$, then iteratively remove vertices whose indegrees (respectively, outdegrees) are less than x (respectively, y) from S (respectively, T), and finally return the residual subgraph as the $[x, y]$-core. Clearly, computing a specific $[x, y]$-core takes $O(n + m)$ time by using the bin-sort technique in Reference [7].

Remark. In Reference [30], Giatsidis et al. introduced another core model on a directed graph G, called (k, l)-core, which is the largest subgraph of G such that each vertex's indegree and outdegree are at least k and l, respectively. This is different with our core model, because our $[x, y]$-core is an (S, T)-induced subgraph such that the outdegree of each vertex in S is at least x and the indegree of each vertex in T is at least y. Moreover, S and T are not necessarily disjoint. In addition, when $S = T$, our $[x, y]$-core is reduced to the (k, l)-core by letting $k = x$ and $l = y$. Hence, the $[x, y]$-core naturally generalizes the (k, l)-core.

[6] In Reference [71], it is defined that k-core is a connected subgraph, but it is stated that cores are not necessarily connected subgraphs in Reference [7].



(a) A weighted graph, G_w (b) A [2, 2]-wcore (c) A [1, 2]-wcore

Fig. 7. An example of the weighted graph and the $[x_w, y_w]$-wcores.

5.2 Extending $[x, y]$-core to Weighted Directed Graphs

For weighted directed graphs, we extend the $[x, y]$-core as $[x_w, y_w]$-wcore, based on which we will develop both efficient exact and approximation algorithms.

Definition 5.5 ($[x_w, y_w]$-wcore). Given a weighted directed graph $G_w = (V, E, W)$, the $[x_w, y_w]$-wcore is the largest (S_w, T_w)-induced subgraph $H = G_w[S_w, T_w]$ with $x_w, y_w \in \mathbb{R}_+$, which satisifies:

(1) $\forall u \in S_w, w_H^+(u) \geq x_w$ and $\forall v \in T_w, w_H^-(v) \geq y_w$;
(2) $\nexists H'$, s.t. $H \subset H'$ and H' satisfies (1).

We call $[x_w, y_w]$ the **weighted core number pair** of the $[x_w, y_w]$-wcore, abbreviated as **wcn-pair**.

Similar to the $[x, y]$-core, the $[x_w, y_w]$-wcore also obeys the nested property.

LEMMA 5.6 (NESTED PROPERTY). *An $[x_w, y_w]$-wcore is contained by an $[x'_w, y'_w]$-wcore, where $x_w \geq x'_w \geq 0$ and $y_w \geq y'_w \geq 0$.*

Example 5.7. In Figure 7(b), the WDDS $G_w[S_w^*, T_w^*]$ is the [2, 2]-wcore of the weighted graph G_w. Figure 7(c) depicts a [1, 2]-wcore, $G_w[S'_w, T'_w]$, of G_w. We can see that the [2, 2]-wcore is nested within the [1, 2]-wcore, since $S_w^* = \{a, b, d\} \subseteq S'_w = \{a, b, c, d\}$ and $T_w^* = \{c, d\} \subseteq T'_w = \{c, d, e\}$.

Similar to $[x, y]$-core computation, given a pair of x_w and y_w, we can compute the $[x_w, y_w]$-wcore by borrowing the idea of k-core decomposition [7]; that is, we can first initialize an (S_w, T_w)-induced subgraph such that $S_w = T_w = V$, then iteratively remove vertices whose outdegrees (respectively, indegrees) are less than x_w (respectively, y_w) from S_w (respectively, T_w), and finally return the residual subgraph as the $[x_w, y_w]$-wcore. However, the bin-sort technique used in the k-core and $[x, y]$-core computation is not applicable for the $[x_w, y_w]$-wcore computation. Therefore, we use the priority queue to select the vertex with minimum indegree or minimum outdegree, making the time complexity of computing an $[x_w, y_w]$-wcore be $O((n + m) \log n)$.

5.3 Density Bounds of Cores and (W)DDS

Here, we focus on the weighted graphs, as the unweighted case can be treated as the special case with all weights equal to one. We first introduce an interesting lemma, then establish the relationship between the WDDS and $[x_w, y_w]$-wcore, and derive the lower and upper bounds of the density of the $[x_w, y_w]$-core.

LEMMA 5.8. *Given a weighted directed graph $G_w = (V, E, W)$ and its WDDS $D_w = G_w[S_w^*, T_w^*]$ with weighted-density ρ_w^*, we have following conclusions:*

(1) for any subset U_S of S_w^*, removing U_S from S_w^* will result in losing edges whose weight sum is at least $\frac{\rho_w^*}{2\sqrt{a}} \times |U_S|$,

(2) for any subset U_T of T_w^*, removing U_T from T_w^* will result in losing edges whose weight sum is at least $\frac{\rho_w^* \sqrt{a}}{2} \times |U_T|$,

where $a = \frac{|S_w^*|}{|T_w^*|}$.

Proof. We prove the lemma by contradiction. For (1), we assume that D_w is the WDDS and removing U_S from D_w results in the losing edges whose weight sum is smaller than $\frac{\rho_w^*}{2\sqrt{a}} \times |U_S|$. This implies that, after removing U_S from S_w^*, the weighted-density of the residual graph, $G_w[S_w^* \setminus U_S, T_w^*]$, will be

$$\begin{aligned}\rho_w(S_w^* \setminus U_S, T_w^*) = \frac{\sum_{e \in E(S_w^* \setminus U_S, T_w^*)} w_e}{\sqrt{|E(S_w^* \setminus U_S| \cdot |T_w^*|}} &> \frac{\rho_w^* \sqrt{|S_w^*||T_w^*|} - \frac{\rho_w^*}{2\sqrt{a}} \times |U_S|}{\sqrt{|E(S_w^* \setminus U_S| \cdot |T_w^*|}} \\ &= \rho_w^* \frac{|S_w^*| - \frac{|U_S|}{2}}{\sqrt{|S_w^*|^2 - |S_w^*||U_S|}} \\ &= \rho_w^* \frac{|S_w^*| - \frac{|U_S|}{2}}{\sqrt{(|S_w^*| - \frac{|U_S|}{2})^2 - \frac{|U_S|^2}{4}}} \\ &> \rho_w^*.\end{aligned}$$

The above inequality implies that removing U_S from D will result in a subgraph with higher weighted density than D_w. However, this contradicts the assumption that D_w is the WDDS, so the conclusion of (1) holds. Similarly, we can prove that the conclusion of (2) holds as well by computing $\rho_w(S_w^*, T_w^* \setminus U_T)$ with assuming that removing U_T from D results in the removal of less than $\frac{\sqrt{a}\rho_w^*}{2} \times |U_T|$ edges from D. Hence, the lemma holds. □

Theorem 5.9. *Given a weighted directed graph $G_w = (V, E, W)$, the WDDS $D_w = G_w[S_w^*, T_w^*]$ is contained in the $[\frac{\rho_w^*}{2\sqrt{a}}, \frac{\sqrt{a}\rho_w^*}{2}]$-wcore, where $a = \frac{|S_w^*|}{|T_w^*|}$.*

Proof. By Lemma 5.8, removing any single vertex from S_w^* will result in losing edges whose weight sum is at least $\frac{\rho_w^*}{2\sqrt{a}}$, so we conclude that for each vertex $u \in S_w^*$, $w_{D_w}^+(u) \geq \frac{\rho_w^*}{2\sqrt{a}}$. Similarly, for each vertex $v \in T_w^*$, we have $w_{D_w}^+(v) \geq \frac{\sqrt{a}\rho_w^*}{2}$. Thus, by the definition of $[x^*, y^*]$-wcore, we claim that the theorem holds. □

In the following, we first show the lower bound of the density of an $[x_w, y_w]$-wcore.

Lemma 5.10 (Lower Bound of Weighted-density of $[x_w, y_w]$-wcore). *Given a weighted graph G_w and an $[x_w, y_w]$-wcore, denoted by $H = G_w[S_w, T_w]$ in G_w, the weighted-density of H is*

$$\rho_w(S_w, T_w) \geq \sqrt{x_w y_w}. \tag{3}$$

Proof. For each vertex $u \in S_w$, since its weighted outdegree $w_H^+(u) \geq x_w$, we have $\sum_{e \in E(S_w, T_w)} w_e \geq x_w |S_w|$. Similarly, we can obtain $\sum_{e \in E(S_w, T_w)} w_e \geq y_w |T_w|$. We have

$$\rho_w(S_w, T_w) = \frac{\sum_{e \in E(S_w, T_w)} w_e}{\sqrt{|S_w||T_w|}} = \sqrt{\frac{\left(\sum_{e \in E(S_w, T_w)} w_e\right)^2}{|S_w||T_w|}} \geq \sqrt{\frac{x_w |S_w| \cdot y_w |T_w|}{|S_w||T_w|}} = \sqrt{x_w y_w}.$$

Thus, Lemma 5.10 holds. □

Next, we derive an upper bound of ρ_w^* based on a novel concept of maximum wcn-pair. Note that this bound is also the upper bound of the weighted density of any (S_w, T_w)-induced subgraph.

Definition 5.11 (Maximum wcn-pair). Given a weighted graph $G_w = (V, E, W)$, a wcn-pair $[x_w, y_w]$ is called the **maximum wcn-pair** if $x_w \cdot y_w$ achieves the maximum value among all the possible $[x_w, y_w]$-wcores. For simplicity, we denote the maximum wcn-pair by $[x_w^*, y_w^*]$.

Lemma 5.12 (Upper Bound of ρ_w^*). *Given a weighted graph $G_w = (V, E, W)$ and its maximum wcn-pair $[x_w^*, y_w^*]$, the weighted-density ρ_w^* of the WDDS is*

$$\rho_w^* \leq 2\sqrt{x_w^* y_w^*}. \tag{4}$$

Proof. We prove the lemma by contradiction. Assume that $\rho_w^* > 2\sqrt{x_w^* y_w^*}$. Let $a^* = \frac{S_w^*}{T_w^*}$. By Theorem 5.9, we conclude that the WDDS is in the $[x'_w, y'_w]$-wcore, where $x'_w > \frac{\sqrt{x_w^* y_w^*}}{\sqrt{a^*}}$ and $y'_w > \sqrt{a^*}\sqrt{x_w^* y_w^*}$. Then, we have $x'_w y'_w > x_w^* y_w^*$, which contradicts the fact that $[x_w^*, y_w^*]$ is the maximum wcn-pair of G_w. Hence, the lemma holds. □

Restricting all weighted to be one, we will also get the definition of the *maximum cn-pair* for the unweighted graph. Similarly, the maximum cn-pair will provide the upper bound of ρ^* for the unweighted graph.

6 EXACT ALGORITHMS FOR DDS AND WDDS

In this section, we introduce the exact algorithms for DDS and WDDS. The focus is on the weighted case, as the unweighted case can be treated as a special case with all weights equal to 1. We first adapt `Exact` [42] to weighted directed graphs, then present core-based exact WDDS and DDS algorithms, and finally introduce efficient exact WDDS and DDS algorithms by exploiting the divide-and-conquer strategy.

6.1 Adaptating `Exact` to Weighted Graphs

To adapt `Exact` for finding the WDDS, we carefully assign capacity values to edges of the flow network so the WDDS can be computed with a theoretical guarantee. Specifically, let $\mathcal{W} = \Sigma_{e \in E} w_e$. We assign capacity values by modifying the function `BuildFlowNetwork` in Algorithm 1:

(1) Replace "capacity m" in lines 16 and 17 with "capacity $\mathcal{W}$";
(2) Replace "capacity $m + \frac{g}{\sqrt{a}}$" in Algorithm 18 with "capacity $\mathcal{W} + \frac{g}{\sqrt{a}}$";
(3) Replace "capacity $m + \sqrt{a}g - 2d_G^-(u)$" in Algorithm 19 with "capacity $\mathcal{W} + \sqrt{a}g - 2w_G^-(u)$";
(4) Replace "capacity 2" in Algorithm 21 with "capacity $2 \cdot w_{(u,v)}$".

We denote the above updated function by `WBuildFlowNetwork`, as shown in Algorithm 4. Figure 8 illustrates the flow network, $F = (V_F, E_F)$, constructed from G_w by `WBuildFlowNetwork` given two parameters, $a = \frac{|S_w|}{|T_w|}$ and g. Note α_e in A (respectively, β_a and β_b in B) is (respectively, are) omitted in Figure 8 for simplicity, because of the lack of outgoing edges for e (respectively, incoming edges for a and b).

By replacing `BuildFlowNetwork` with `WBuildFlowNetwork` in `Exact`, we obtain an exact WDDS algorithm, denoted by `W-Exact`. Algorithm 4 presents its detailed steps.

Lemma 6.1. *Given a weighted directed graph $G_w = (V, E, W)$,* `W-Exact` *is able to correctly find a WDDS of G_w.*

Proof. We can prove the lemma by following the proof of the correctness of `Exact` in Reference [42]. For completeness, we formulate it here.

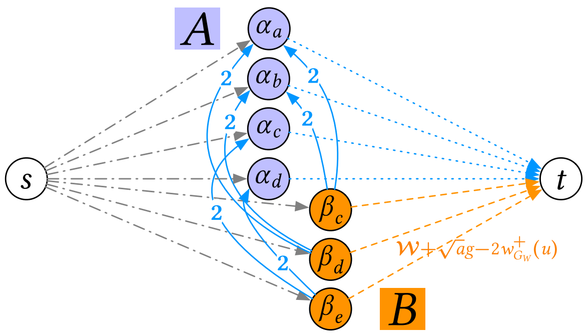


Fig. 8. The flow network constructed from G_w in Figure 7(a).

ALGORITHM 4: `W-Exact`

```
Input  : $G_w = (V, E, W)$
Output : The exact WDDS $D_w = G_w[S_w^*, T_w^*]$
1  Lines 1–12 in Algorithm 1 by replacing BuildFlowNetwork with WBuildFlowNetwork;
2  Function WBuildFlowNetwok($G_w = (V, E, W), a, g$):
3      $A \leftarrow \{\alpha_u | u \in V\}, B \leftarrow \{\beta_u | u \in V\}, E_F \leftarrow \emptyset$;
4      $V_F \leftarrow \{s\} \cup A \cup B \cup \{t\}$;
5      for $\alpha_u \in A$ do add $(s, \alpha_u)$ to $E_F$ with capacity $\mathcal{W}$;
6      for $\beta_u \in B$ do add $(s, \beta_u)$ to $E_F$ with capacity $\mathcal{W}$;
7      for $\alpha_u \in A$ do add $(\alpha_u, t)$ to $E_F$ with capacity $\mathcal{W} + \frac{g}{\sqrt{a}}$;
8      for $\beta_u \in B$ do add $(\beta_u, t)$ to $E_F$ with capacity $\mathcal{W} + \sqrt{a}g - 2w^{-}_{G_w}(u)$;
9      for $(u, v) \in E$ do add $(\beta_v, \alpha_u)$ to $E_F$ with capacity $2 \cdot w_{(u,v)}$;
10     return $F = (V_F, E_F)$
```

Given the flow network, $F = (V_F, E_F)$ constructed by `WBuildFlowNetwork` from the weighted graph $G_w = (V, E, W)$, since the cut $\langle\{s\}, A \cup B \cup \{t\}\rangle$ has weight $\mathcal{W}(|A| + |B|)$, the minimum st-cut value is less or equal than $\mathcal{W}(|A| + |B|)$. Now consider the cut $\langle s \cup S_w \cup T_w, t \cup (A \setminus S_w) \cup (B \setminus T_w)\rangle$, where $S_w \subseteq A$ and $T_w \subseteq B$. The capacity of edges crossing the cut is

$$\begin{aligned}
&\mathcal{W}(|A| - |S_w| + |B| - |T_w|) + \left(\mathcal{W} + \frac{g}{\sqrt{a}}\right)|S_w| + \sum_{i \in T_w} (\mathcal{W} + \sqrt{a}g - 2w^{-}_{G_w}(i)) + \sum_{\substack{i \in T_w, j \in A \setminus S_w,\\ e=(j,i) \in E}} 2w_e \\
&= \mathcal{W}(|A| + |B|) + |S_w|\frac{g}{\sqrt{a}} + |T_w|\sqrt{a}g - 2\sum_{e \in E(S_w, T_w)} w_e \\
&= \mathcal{W}(|A| + |B|) + \frac{|S_w|}{\sqrt{a}}\left(g - \frac{\sum_{e \in E(S_w,T_w)} w_e}{|S_w|/\sqrt{a}}\right) + |T_w|\sqrt{a}\left(g - \frac{\sum_{e \in E(S_w,T_w)} w_e}{|T_w|\sqrt{a}}\right).
\end{aligned} \tag{5}$$

Assume the cut $\langle s \cup S_w \cup T_w, t \cup (A \setminus S_w) \cup (B \setminus T_w)\rangle$ is returned by the `Min-ST-Cut` algorithm. Here, we only consider the case that both S_w and T_w are non-empty sets. Otherwise, the minimum cut will be $\langle\{s\}, A \cup B \cup \{t\}\rangle$. Let $b = \frac{|S_w|}{|T_w|}$. We have

$$
\begin{aligned}
&\frac{|S_w|}{\sqrt{a}}\left(g-\frac{\sum_{e\in E(S_w,T_w)} w_e}{|S_w|/\sqrt{a}}\right)+|T_w|\sqrt{a}\left(g-\frac{\sum_{e\in E(S_w,T_w)} w_e}{|T_w|\sqrt{a}}\right)\\
&=\sqrt{|S_w||T_w|}\frac{\sqrt{b}}{\sqrt{a}}\left(g-\frac{\rho_w(S_w,T_w)}{\sqrt{b}/\sqrt{a}}\right)+\sqrt{|S_w||T_w|}\frac{\sqrt{a}}{\sqrt{b}}\left(g-\frac{\rho_w(S_w,T_w)}{\sqrt{a}/\sqrt{b}}\right)\\
&=\sqrt{|S_w||T_w|}\left(\left(\frac{\sqrt{b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{b}}\right)g-2\rho_w(S_w,T_w)\right).
\end{aligned}
\tag{6}
$$

Notice $(\frac{\sqrt{b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{b}})\geq 2, \forall a,b\in\mathbb{R}_+$. Thus, if the guessed $g>\rho_w^*\geq\rho_w(S_w,T_w)$, then Equation (6) is larger than 0 and the minimum cut will be $\langle\{s\}, A\cup B\cup\{t\}\rangle$. If the guessed $g\leq\rho_w^*$, we can always find a minimum cut, whose source side has more nodes than $\{s\}$, by always ensuring that the minimum st-cut obtained from `Min-ST-Cut` has the largest size on the source side. If the guessed value $g=\rho_w^*$, then we can obtain the optimal solution (S_w^*, T_w^*) when the enumerated $a=\frac{|S_w^*|}{|T_w^*|}$. Hence, the lemma holds. □

6.2 Core-based Exact Algorithms

We first show how to locate the WDDS in some wcores by Theorem 5.9 to downsize the flow network built in `WExact`. Since the value of ρ_w^* may not be known in advance, we can only locate the DDS in some wcores based on $a=\frac{|S_w|}{|T_w|}$ and g guessed by exploiting the nested property of cores. For example, given a specific a and a lower bound l of ρ_w^*, then we can locate the WDDS in the $[\frac{l}{2\sqrt{a}}, \frac{\sqrt{a}l}{2}]$-core, since the $[\frac{\rho_w^*}{2\sqrt{a}}, \frac{\sqrt{a}\rho_w^*}{2}]$-core is nested within the $[\frac{l}{2\sqrt{a}}, \frac{\sqrt{a}l}{2}]$-core. Since the WDDS is in some $[x,y]$-wcores that are often much smaller than G_w, we can build the flow network on these wcores, rather than the entire graph G_w, which will significantly improve the overall efficiency.

Moreover, during the binary search, the lower bound l of ρ_w^* is gradually enlarged, so we can further locate the WDDS in the $[x_w, y_w]$-wcores with larger values of wcn-pairs. As the values of wcn-pairs increase, the sizes of $[x_w, y_w]$-wcores become smaller, so the flow networks built become smaller gradually, and the cost of computing the minimum st-cut is greatly reduced.

Based on the above core-based optimization techniques, we develop a novel exact algorithm, called `WCore-Exact`, which follows the same framework of `WExact`, as shown in Algorithm 5. Specifically, it first runs a 2-approximation algorithm[7] and initializes ρ_w^* as the density of the approximate WDDS (lines 1–2). Then, for each value of a, it sets the lower and upper bounds of ρ_w^* using $\widetilde{\rho_w^*}$ (lines 3–4). After that, it performs a binary search to compute the WDDS, where the flow networks are built based on the $[x_w, y_w]$-wcores (lines 5–13).

By restricting all weights to be one and replacing the wcore-based pruning to the core-based pruning (i.e., rounding up $x\leftarrow\lceil\frac{l}{2\sqrt{a}}\rceil$ and $y\leftarrow\lceil\frac{\sqrt{a}l}{2}\rceil$ in line 6) in `WCore-Exact`, we get the algorithm for the unweighted case, `Core-Exact`.

Analysis. It takes $O(n^2\sqrt{m})$ time to compute the minimum st-cut. Thus, `WCore-Exact` (respectively, `Core-Exact`) takes $O(n^4\sqrt{m})$ time. Nevertheless, since we locate the WDDS (respectively, DDS) in some $[x_w, y_w]$-wcores (respectively, $[x,y]$-cores), the flow networks become smaller, so `WCore-Exact` (respectively, `Core-Exact`) runs much faster than `WExact` (respectively, `Exact`) in practice.

[7]Note that any 2-approximation algorithm can be applied here; in this article, we use our core-based approximation algorithm developed in Section 7.

ALGORITHM 5: WCore-Exact

```
Input   : $G_w = (V, E, W)$
Output : The exact WDDS $D_w = G[S^*_w, T^*_w]$
1  $\widetilde{\rho^*_w}$ ←run a 2-approximation algorithm;
2  $\rho^*_w \leftarrow \widetilde{\rho^*_w}$;
3  foreach $a \in \{\frac{n_1}{n_2} | 0 < n_1, n_2 <= n\}$ do
4      $l \leftarrow \rho^*_w, r \leftarrow 2\widetilde{\rho^*_w}$;
5      while $r - l \geq \frac{\sqrt{n}-\sqrt{n-1}}{n\sqrt{n-1}} \min_{e\in E} w_e$ do
6          $g \leftarrow \frac{l+r}{2}, x_w \leftarrow \frac{l}{2\sqrt{a}}, y_w \leftarrow \frac{\sqrt{a}l}{2}$;
7          $G_r$ ←Get-XY-WCore($G_w, x_w, y_w$);
8          $F = (V_F, E_F)$ ← WBuildFlowNetwok($G_r$,$a$,$g$);
9          $\langle \mathcal{S}, \mathcal{T} \rangle$ ← Min-ST-Cut($F$);
10         if $\mathcal{S} = \{s\}$ then $r \leftarrow g$ ;
11         else
12             $l \leftarrow g$;
13             if $g > \rho^*$ then $D_w \leftarrow G[\mathcal{S} \cap A, \mathcal{S} \cap B]$, $\rho^*_w = g$;
14 return $D_w$;
```

6.3 Divide-and-conquer-based Exact Algorithms

In WCore-Exact, we mainly optimize the inner loop of WExact, i.e., reducing the cost of computing the minimum st-cut. A natural question comes: Can we improve the outer loop of WExact so we can enumerate fewer values of $a = \frac{|S_w|}{|T_w|}$? In the following, we show that this is possible.

Our idea is based on a key observation that given a specific value of a, the results of the binary search (lines 5–13 in Algorithm 5) actually have provided insights for reducing the number of trials of a. Inspired by the results in Reference [42], essentially, the binary search solves the following optimization problem, where a is a pre-given value.

$$\max_{S_w, T_w \in V} g$$
$$\text{s.t.}\ \frac{|S_w|}{\sqrt{a}}\left(g - \frac{\sum_{e\in E(S_w,T_w)} w_e}{|S_w|/\sqrt{a}}\right) + |T_w|\sqrt{a}\left(g - \frac{\sum_{e\in E(S_w,T_w)} w_e}{|T_w|\sqrt{a}}\right) \leq 0. \tag{7}$$

g is the maximum value the binary search can obtain when a is fixed. Then, we can derive the following lemma.

Lemma 6.2. *Given a weighted graph $G_w = (V, E, W)$ and a specific a, assume that S'_w and T'_w are the optimal choices for Equation (7). Let $b = \frac{|S'_w|}{|T'_w|}$ and $c = \frac{a^2}{b}$. Then, for any (S_w, T_w)-induced subgraph $G_w[S_w, T_w]$ of G_w, if $\min\{b, c\} \leq \frac{|S_w|}{|T_w|} \leq \max\{b, c\}$, then we have $\rho_w(S_w, T_w) \leq \rho_w(S'_w, T'_w)$.*

Proof. We first introduce more details regarding Equation (7), and then we prove the lemma by contradiction.

In Equation (7), given a specific a, let $g^*(a)$ be the optimal value of g. Because S'_w and T'_w are the optimal choices for S_w and T_w in Equation (7), S', T', and $g^*(a)$ have the following relationship,

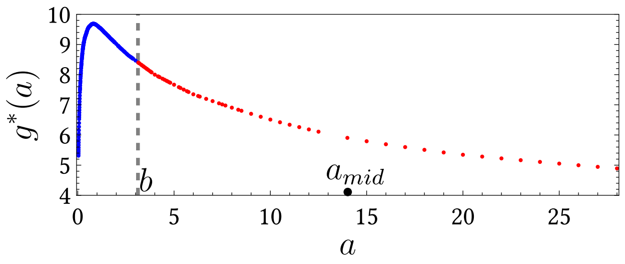


Fig. 9. The pruning effectiveness of Lemma 6.2.

according to the proof of Lemma 6.1:

$$
\begin{aligned}
\frac{|S'_w|}{\sqrt{a}}\left(g^*(a)-\frac{|E(S'_w,T'_w)|}{|S'_w|/\sqrt{a}}\right)+|T'_w|\sqrt{a}\left(g^*(a)-\frac{|E(S'_w,T'_w)|}{|T'_w|\sqrt{a}}\right)&=0\\
\sqrt{|S'_w||T'_w|}\frac{\sqrt{b}}{\sqrt{a}}\left(g^*(a)-\frac{\rho_w(S'_w,T'_w)}{\sqrt{b}/\sqrt{a}}\right)+\sqrt{|S'_w||T'_w|}\frac{\sqrt{a}}{\sqrt{b}}\left(g^*(a)-\frac{\rho_w(S'_w,T'_w)}{\sqrt{a}/\sqrt{b}}\right)&=0\\
\sqrt{|S'_w||T'_w|}\left(\left(\frac{\sqrt{b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{b}}\right)g^*(a)-2\rho_w(S'_w,T'_w)\right)&=0.
\end{aligned}
\tag{8}
$$

Then, $g^*(a)$ can be written as

$$
g^*(a)=\frac{2\rho_w(S'_w,T'_w)}{\frac{\sqrt{b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{b}}}.
\tag{9}
$$

Let $h_a(x)=\frac{\sqrt{x}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{x}}$. As $t+\frac{1}{t}$ is monotonically decreasing for $t\in(0,1]$ and monotonically increasing for $t\in[1,\infty)$, we have that $h_a(x)$ is monotonically decreasing for $x\in(0,a]$ and monotonically increasing for $x\in[a,\infty)$, via replacing t with $\frac{\sqrt{x}}{\sqrt{a}}$. Further, because $h_a(c)=h_a(\frac{a^2}{b})=\frac{\sqrt{a^2/b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{a^2/b}}=\frac{\sqrt{a}}{\sqrt{b}}+\frac{\sqrt{b}}{\sqrt{a}}=h_a(b)$, $h_a(x)\le h_a(b)$ for $\min\{b,c\}\le x\le\max\{b,c\}$.

We now prove the lemma by contradiction. Assume that there exists an $[S_x,T_x]$-induced subgraph, which satisfies $\min\{b,c\}\le x=\frac{|S_x|}{|T_x|}\le\max\{b,c\}$, but it has $\rho_w(S_x,T_y)>\rho_w(S'_w,T'_w)$. Since $h_a(x)\le h_a(b)$ and $\rho_w(S_x,T_y)>\rho_w(S'_w,T'_w)$, we can conclude $\frac{2\rho_w(S_x,T_x)}{h_a(x)}>\frac{2\rho_w(S'_w,T'_w)}{h_a(b)}$. However, this contradicts the assumption that S'_w and T'_w are the optimal choice for Equation (7). Hence, the lemma holds. □

According to Lemma 6.2, after conducting the binary search for a specific value of a, we can skip performing binary search for all the possible values of a in the range $(\min\{b,c\},\max\{b,c\})$, so the overall efficiency can be improved dramatically. Note that, since $a^2=bc$, we always have $a\in(\min\{b,c\},\max\{b,c\})$.

To further illustrate the pruning effectiveness of Lemma 6.2, we experiment on a small real graph and discuss the results in Example 6.3.

Example 6.3. We consider a small dataset MA [70] that consists of 28 vertices and 217 edges; this implies that the values of a are in the range $[\frac{1}{28},28]$. We plot the values of $g^*(a)$ for $a\in[\frac{1}{28},28]$ in Figure 9. Let $a_{mid}=(\frac{1}{28}+28)/2$. After applying the binary search for a_{mid}, we get $a=14.02$, $b=3.125$, and $c=62.88$, by Lemma 6.2. Therefore, we can skip the binary search for all the 78 values of $a\in(3.125,62.88)$, which are marked in red in Figure 9.

ALGORITHM 6: WDC-Exact

```
Input  : $G = (V, E, W)$
Output : The exact WDDS $D_w = G_w[S^*, T^*]$
1 $a_l \leftarrow \frac{1}{n}, a_r \leftarrow n, \rho^*_w \leftarrow 0, D_w \leftarrow \emptyset$;
2 Divide-Conquer($a_l, a_r$);
3 return $D$;
4 Function Divide-Conquer($a_l, a_r$):
5     $a_{mid} \leftarrow \frac{a_l + a_r}{2}$;
6     run Lines 4–13 of Algorithm 5 (replace $x_w \leftarrow \frac{l}{2\sqrt{a}}, y_w \leftarrow \frac{\sqrt{a}l}{2}$ with $x_w \leftarrow \frac{l}{2\sqrt{a_r}}, y_w \leftarrow \frac{\sqrt{a_l}l}{2}$);
7     let $G_w[S'_w, T'_w]$ be the WDDS found by binary search;
8     $b \leftarrow \frac{|S'_w|}{|T'_w|}$;
9     $c \leftarrow \frac{a^2_{mid}}{b}$;
10    if $b > c$ then Swap($b, c$);
11    if $a_l \leq b$ then Divide-Conquer($a_l, b$);
12    if $c \leq a_r$ then Divide-Conquer($c, a_r$);
```

Based on the discussions above, we develop a novel divide-and-conquer algorithm, named WDC-Exact, as shown in Algorithm 6. First, it initializes a_l to the smallest ratio $\frac{1}{n}$, a_r to the largest ratio n, ρ^* to 0, and D to $\emptyset$ (Algorithm 1).

Then, it applies (Algorithm 2) the function Divide-Conquer to check the possible values of a (lines 4–12). Specifically, in Divide-Conquer, it first picks the middle point a_{mid} in range $[a_l, a_r]$ (Algorithm 5). Then, it uses the binary search process (similar to the one in WCore-Exact) to find the (S'_w, T'_w)-induced subgraph $G_w[S'_w, T'_w]$ that maximizes Equation (7) (Algorithm 6). Afterwards, it computes b and c according to Lemma 6.2 (lines 8–10). Finally, it skips the whole range (b, c) of the value of a and searches on the two intervals split by (b, c) recursively to compute the DDS. Note that to exploit the $[x_w, y_w]$-wcores for improving the efficiency, we build the flow networks on the union of $[x_w, y_w]$-wcores for all possible values of a in the range $[a_l, a_r]$, i.e., the $[\frac{l}{2\sqrt{a_r}}, \frac{\sqrt{a_l}l}{2}]$-wcore (Algorithm 6).

By restricting all weights of the graph to be one and replacing the wcore-based pruning to the core-based pruning (i.e., rounding up $x \leftarrow \lceil \frac{l}{2\sqrt{a_r}} \rceil$ and $y \leftarrow \lceil \frac{\sqrt{a_l}l}{2} \rceil$ in line 6) in WDC-Exact, we get the algorithm for the unweighted case, DC-Exact.

Complexity. The time complexity of WDC-Exact (and DC-Exact) is $O(k \cdot n^2 \sqrt{m})$, where k denotes the number of times invoking the binary search, which is at most n^2, since the binary search is invoked at most n^2 times in the worst case. Nevertheless, in practice, we have $k \ll n^2$. As shown by our experiments later, k is often orders of magnitude smaller than n^2.

7 CORE-BASED APPROXIMATION DDS AND WDDS ALGORITHMS

While our exact algorithms, DC-Exact and WDC-Exact, are significantly faster than the state-of-the-art algorithm Exact and its weighted adaptation WExact, we can further speed it up by trading accuracy: We first develop a fast 2-approximation WDDS algorithm, called WCore-Approx. Next, we propose an optimized version for unweighted graphs, named Core-Approx, which achieves an approximation ratio of 2, within $O(\sqrt{m}(m + n))$ time.

7.1 A Core-based Approximation WDDS Algorithm

Our core-based approximation algorithm mainly relies on Theorem 7.1.

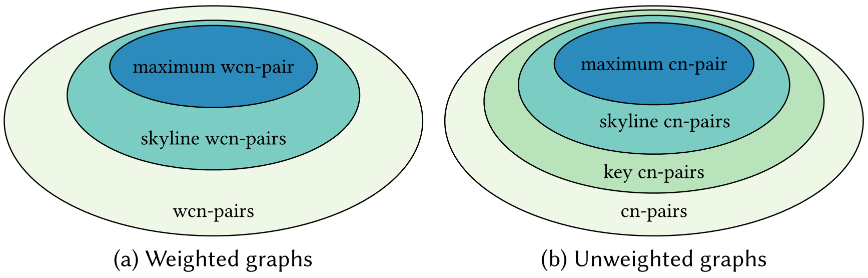


Fig. 10. The Venn diagram of cn-pairs and wcn-pairs.

Theorem 7.1. *Given a weighted directed graph $G_w = (V, E, W)$, the wcore whose wcn-pair is the maximum wcn-pair, i.e., $[x^*_w, y^*_w]$-wcore, is a 2-approximation solution to the WDDS problem.*

Proof. Let the $[x^*_w, y^*_w]$-wcore be an (S_w, T_w)-induced subgraph. By Lemma 5.10, we have $\rho_w(S_w, T_w) \geq \sqrt{x^*_w y^*_w}$. According to Lemma 5.12, we have $\rho^*_w \leq 2\sqrt{x^*_w y^*_w}$, so we conclude

$$\frac{\rho^*_w}{\rho_w(S_w, T_w)} \leq \frac{2\sqrt{x^*_w y^*_w}}{\sqrt{x^*_w y^*_w}} = 2. \tag{10}$$

Hence, the theorem holds. □

According to Theorem 7.1, the wcore with the maximum wcn-pair is a 2-approximation solution. A straightforward method is to compute all the wcores of G_w and then return the one with the maximum wcn-pair. Theoretically, for each vertex u, its weighted outdegree (respectively, indegree) in different (S_w, T_w)-induced subgraphs can have up to $2^{d^+_{G_w}(u)}$ (respectively, $2^{d^-_{G_w}(u)}$) different values. In the worst case, x_w (respectively, y_w) can have up to $\sum_{u\in V} 2^{d^+_{G_w}(u)}$ (respectively, $\sum_{u\in V} 2^{d^-_{G_w}(u)}$) different values. Hence, the strategy to iterate all possible wcn-pairs is not practical for large weighted graphs. To avoid enumerating all the possible combinations of x_w or y_w, we propose a new algorithm based on iterating the *skyline wcn-pairs*, which we will introduce next.

Definition 7.2 (Skyline wcn-pair). Given a weighted directed graph $G_w = (V, E, W)$, the wcn-pair of an $[x_w, y_w]$-wcore is a **skyline wcn-pair**, if there does not exist any other $[x'_w, y'_w]$-wcore in G_w that satisfies $x_w \leq x'_w$ and $y_w \leq y'_w$.

Clearly, the maximum wcn-pair must be a skyline wcn-pair. Figure 10(a) shows the logical inclusion-ship among different kinds of wcn-pairs on weighted graphs. We further illustrate the concept of skyline wcn-pair in Example 7.3.

Example 7.3. Suppose that we have a weighted graph whose wcn-pairs are presented in Figure 11, where each point in the gray area denotes the wcn-pair of the $[x_w, y_w]$-wcore. The skyline wcn-pairs are marked by circles, in which the blue circle denotes the maximum wcn-pair, i.e., $[x_3, y_3]$. Review Figure 11; we can find that all skyline wcn-pairs look like different steps on the stair. Hence, we will adopt a stair-climbing strategy (starting from $[x_5, y_1]$ and ending at $[x_1, y_5]$) to search all skyline wcn-pairs.

Based on the above discussions, we develop `WCore-Approx`, which aims to find the $[x^*_w, y^*_w]$-wcore as a 2-approximation solution. Algorithm 7 presents `WCore-Approx`, in which we sequentially find all the skyline wcn-pairs in a stair-climbing manner. Specifically, for the first skyline wcn-pair $[x'_w, y'_w]$, its first element x'_w is the maximum value of x_w, and we get its second element y'_w by increasing it from its minimum value. After getting $[x'_w, y'_w]$, we find the second skyline

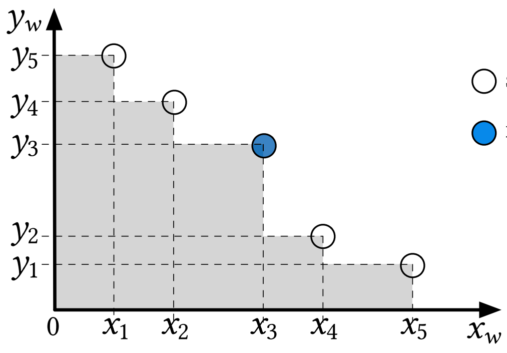


Fig. 11. Illustrating the concepts of skyline wcn-pair and maximum wcn-pair.

ALGORITHM 7: `WCore-Approx`

```
Input  : $G_w = (V, E, W)$
Output : The approximate WDDS $\widetilde{D}_w$, i.e., the $[x_w^*, y_w^*]$-wcore
1  $x_w^* \leftarrow 0, y_w^* \leftarrow 0$;
2  $x_w \leftarrow +\infty, y_w \leftarrow 0$;
3  while $x_w > 0$ do
4      $x_w \leftarrow$ GetMaxXw($G_w, y_w, \leq$);
5      if $x_w = 0$ then break;
6      $y_w \leftarrow$ GetMaxYw($G_w, x_w, <$);
7      if $x_w y_w > x_w^* y_w^*$ then $x_w^* \leftarrow x_w, y_w^* \leftarrow y_w$;
8  return the $[x_w^*, y_w^*]$-wcore;
9  Function GetMaxXw($G_w, y_w, \triangleleft$):
       /* $\triangleleft$ is an inequality operator, either $<$ or $\leq$ */
10     $S_w \leftarrow V, T_w \leftarrow V, x_w \leftarrow 0$ ;
11     while $\exists v \in T_w, w^-_{G_w}(v) \triangleleft y_w$ do
12         $E \leftarrow E \setminus \{(u, v) | u \in S_w\}, T_w \leftarrow T_w \setminus v$;
13     while $|E| > 0$ do
14         $x_w \leftarrow \min_{u \in S_w}\{w^+_{G_w}(u)\}$;
15         while $\exists u \in S_w, w^+_{G_w}(u) \leq x_w$ do
16             $E \leftarrow E \setminus \{(u, v) | v \in T_w\}, S_w \leftarrow S_w \setminus u$;
17             while $\exists v \in T_w, w^-_{G_w}(v) \triangleleft y_w$ do
18                 $E \leftarrow E \setminus \{(u, v) | u \in S_w\}, T_w \leftarrow T_w \setminus v$;
19     return $x_w$;
20 Function GetMaxYw($G_w, x_w, \triangleleft$):
21     reuse lines 10–19 by interchanging $u$ with $v$, $S_w$ with $T_w$, $x_w$ with $y_w$, and $d^-_{G_w}$ with $d^+_{G_w}$;
```

wcn-pair, whose first element is the second largest value of x_w, and we get its second element by increasing its value from y'_w to its maximum value. This process is repeated until we find all the skyline wcn-pairs (lines 2–7).

Given a threshold y_w and an inequality operator $\triangleleft$ (i.e., $<$ or $\leq$), the function `GetMaxXw` computes the wcn-pair whose second element is *larger* (or *larger or equal*, according to $\triangleleft$) than y_w and first element is maximized. In `GetMaxXw`, we first initialize S_w, T_w, and x_w (line 10). Then, we make sure that the indegrees of all the vertices in T_w are larger than y_w (lines 11–12). In the following loop (lines 13–18), we first set x_w as the minimum outdegree of vertices in S_w (line 14), and then we remove the vertex whose outdegree is less or equal than x_w (lines 15–16), but this may make some vertices' indegrees in T_w become less or equal than y_w, so we have to remove these vertices as well (lines 17–18). Finally, we return the last updated value of x_w (i.e., the maximum value of x_w). Analogously, we have a function `GetMaxYw` to get the skyline wcn-pair whose first element is a given x_w.

We further explain `WCore-Approx` by Example 7.4.

Example 7.4. Reconsider the graph and its wcn-pairs in Example 7.3. To find the $[x_w^*, y_w^*]$-wcore, `WCore-Approx` will run steps as follows:

(1) find the maximimum value of x_w, i.e., x_5, while keeping $y_w > 0$, and then find the maximimum value of y_w, i.e., y_1, while keeping $x_w \geq x_5$, so the first skyline wcn-pair is $[x_5, y_1]$;
(2) find the second largest value of x_w, i.e., x_4 while keeping $y_w > y_1$, and then find the maximum value of y_w, i.e., y_2, while keeping $x_w \geq x_4$, so the second skyline wcn-pair is $[x_4, y_2]$;
(3) similarly, find the third to the fifth skyline wcn-pairs, i.e., $[x_3, y_3]$, $[x_2, y_4]$, and $[x_1, y_5]$;
(4) return the $[x_w^*, y_w^*]$-wcore, i.e., $[x_3, y_3]$-wcore, as the 2-approximation WDDS.

Complexity. The time complexity of `WCore-Approx` depends on the number of skyline wcn-pairs, denoted by ξ, in the weighted graph. Thus, its time complexity is bounded by $O(\xi(n + m)\log n)$, where the step of obtaining the minimum outdegree (or indegree) (in line 14) needs $O(\log n)$ operations with the help of a heap.

7.2 The Optimized Approximation Algorithm for the Unweighted Case

When the graph is unweighted, `WCore-Approx` can still provide a 2-approximation DDS. However, we can further improve the time complexity by exploiting the connection among different kinds of cn-pairs. Based on the connection, we propose a 2-approximation DDS algorithm, `Core-Approx`, with time complexity of $O(\sqrt{m}(n + m))$.

The goal of `Core-Approx` is still to find the core with the maximum cn-pair according to Theorem 7.1. First, we introduce three kinds of cn-pairs to facilitate the search of the maximum cn-pair.

Definition 7.5 (Maximum Equal cn-pair). Given a graph $G = (V, E)$, a cn-pair $[x, x]$ is the **maximum equal cn-pair** if x achives the maximum value among all the possible $[x, x]$-cores. We denote the maximum equal cn-pair by $[\gamma, \gamma]$.

Remarks. The approximation algorithm `KS-Approx` [42] actually returns the $[\gamma, \gamma]$-core in the graph. However, the $[\gamma, \gamma]$-core may not be necessarily the $[x^*, y^*]$-core; hence, it is not guaranteed to be the 2-approximation DDS.

Lemma 7.6. *Given a graph $G = (V, E)$ and its maximum equal cn-pair $[\gamma, \gamma]$, for any cn-pair $[x, y]$, we have either $x \leq \gamma$ or $y \leq \gamma$, or both of them.*

Proof. We prove this lemma by contradiction. Assume there is a cn-pair $[x, y]$ where $x > \gamma$ and $y > \gamma$. Then, let $\gamma' = \min\{x, y\} > \gamma$, so there exists a $[\gamma', \gamma']$-core in G, which contradicts $[\gamma, \gamma]$ is the maximum equal cn-pair. □

Definition 7.7 (Skyline cn-pair). Given a graph $G = (V, E)$, the cn-pair of an $[x, y]$-core is a **skyline cn-pair** if there does not exist any other $[x', y']$-core in G satisfying $x \leq x'$ and $y \leq y'$.

Definition 7.8 (Key cn-pair). Given a graph $G = (V, E)$ and its maximum equal cn-pair $[\gamma, \gamma]$, the cn-pair of an $[x, y]$-core is a **key cn-pair**, if one of the following conditions is satisfied:

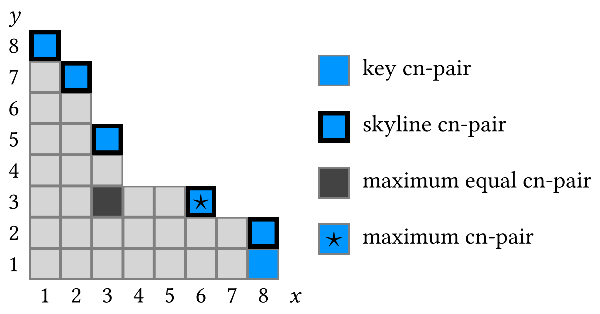


Fig. 12. Illustrating the concepts of cn-pairs.

(1) if $x \leq \gamma$, there does not exist any $[x, y']$-core in G, such that $y' > y$;
(2) if $y \leq \gamma$, there does not exist any $[x', y]$-core in G, such that $x' > x$.

Clearly, the maximum cn-pair is a skyline cn-pair and also a key cn-pair. Any skyline cn-pair is a key cn-pair, but a key cn-pair may not be a skyline cn-pair. Example 7.9 illustrates these concepts.

Example 7.9. Suppose that we have a graph whose cn-pairs are presented in Figure 12, where each colored cell (x, y) denotes the cn-pair of the $[x, y]$-core. Then, the cn-pairs of the blue cells are key cn-pairs, in which the one with a star is the maximum cn-pair. The cn-pair of the black cell, i.e., [3, 3], is the maximum equal cn-pair. The cn-pair [8, 1] is a key cn-pair, but not a skyline cn-pair, while all the other key cn-pairs are skyline cn-pairs.

To further illustrate the relationship among different kinds of cn-pairs, we use Figure 10(b) to show their logical inclusion-ship.

LEMMA 7.10. *Given a graph $G = (V, E)$ and its maximum equal cn-pair $[\gamma, \gamma]$, we have $\gamma \leq \sqrt{m}$.*

PROOF. A $[\gamma, \gamma]$-core contains at least γ vertices whose outdegrees are at least γ. Meanwhile, there are at most m edges in the $[\gamma, \gamma]$-core. Hence, $\gamma \cdot \gamma \leq m$. □

By combining Lemmas 7.10 and 7.6, we get Lemma 7.11.

LEMMA 7.11. *Given a graph $G = (V, E)$, there are at most $2\sqrt{m}$ key cn-pairs in G.*

PROOF. According to Lemma 7.6 and Definition 7.8, there are at most 2γ key cn-pairs in G. Since we have $\gamma \leq \sqrt{m}$ by Lemma 7.10, there are at most $2\sqrt{m}$ key cn-pairs in G. □

Based on the above discussions, we develop `Core-Approx`, which returns the $[x^*, y^*]$-core as an approximate DDS. Precisely, we first compute the maximum equal cn-pair, enumerate all the key cn-pairs, and finally return the core with the maximum cn-pair. Algorithm 8 presents `Core-Approx`. First, it obtains the maximum equal cn-pair (line 2). Then, it enumerates x and y from 1 to γ to search all the key cn-pairs (lines 3–8). Finally, the $[x^*, y^*]$-core is returned.

Given an input x, the function `GetMaxY` computes the key cn-pair whose first element is x. In `GetMaxY`, we first initialize S, T, $y_{\max}$, and y, where y is set to $\lfloor \frac{x^* y^*}{x} \rfloor + 1$. Then, in the loop, if there is a vertex $u \in T$ with indegree less than y, we remove it (lines 14–15); this may make some vertices' outdegrees become less than x, so we have to remove these vertices as well (lines 16–17). After that, we update $y_{\max}$ and increase y by 1 (lines 18–19). Finally, we get the maximum value of y. Similarly, we have a function `GetMaxX` to get the key cn-pair whose second element is a given y. During the searching process of the $[x^*, y^*]$-core, we will also keep track of the densities of the subgraphs by maintaining the remaining number of edges and vertices and capturing the densest

ALGORITHM 8: `Core-Approx`

```
   Input   : $G = (V, E)$
   Output : An approximate DDS $\widetilde{D}$, i.e., the $[x^*, y^*]$-core
1  $x^* \leftarrow 0, y^* \leftarrow 0$;
2  $[\gamma, \gamma] \leftarrow$ compute the $[\gamma, \gamma]$-core by iteratively peeling vertices that have the minimum indegrees or
   outdegrees;
3  for $x \leftarrow 1$ to $\gamma$ do
4      $y \leftarrow$ GetMaxY$(G, x)$;
5      if $xy > x^*y^*$ then $x^* \leftarrow x, y^* \leftarrow y$ ;
6  for $y \leftarrow 1$ to $\gamma$ do
7      $x \leftarrow$ GetMaxX$(G, y)$;
8      if $xy > x^*y^*$ then $x^* \leftarrow x, y^* \leftarrow y$ ;
9  return the $[x^*, y^*]$-core;
10 Function GetMaxY$(G, x)$:
11     $S \leftarrow V, T \leftarrow V, y_{\max} \leftarrow 0, y \leftarrow \lfloor \frac{x^*y^*}{x} \rfloor + 1$;
12     if $y > \max_{u \in T}\{d_G^-(u)\}$ then return $y_{\max}$;
13     while $|E| > 0$ do
14         while $\exists u \in T, d_G^-(u) < y$ do
15             $E \leftarrow E \setminus \{(v, u) | v \in S\}, T \leftarrow T \setminus \{u\}$;
16             while $\exists v \in S, d_G^+(v) < x$ do
17                 $E \leftarrow E \setminus \{(v, u) | u \in T\}, S \leftarrow S \setminus \{v\}$;
18         if $|E| > 0$ then $y_{\max} \leftarrow y$;
19         $y \leftarrow y + 1$;
20     return $y_{\max}$;
21 Function GetMaxX$(G, y)$:
22     reuse lines 11–20 by interchanging $u$ with $v$, $S$ with $T$, $x$ with $y$, $d_G^+$ with $d_G^-$, and changing $y_{\max}$ to
       $x_{\max}$;
```

subgraph among those subgraphs. This heuristic can help get a better approximation to the DDS at limited extra cost.

We further illustrate `Core-Approx` by Example 7.12.

Example 7.12. Given a graph with cn-pairs in Example 7.9, `Core-Approx` will run steps as follows:

(1) finds the maximum equal cn-pair $[3, 3]$;
(2) iterates x from 1 to 3 to compute the key cn-pairs whose first elements are x, i.e., $[1, 8]$, $[2, 7]$, and $[3, 5]$;
(3) iterates y from 1 to 3 to search the key cn-pairs whose second elements are y, i.e., $[8, 1]$, $[8, 2]$, and $[6, 3]$;
(4) returns the $[x^*, y^*]$-core, i.e., $[6, 3]$-core.

Complexity. Computing the $[\gamma, \gamma]$-core takes $O(m+n)$ time, as it iteratively peels vertices with the minimum indegrees or outdegrees. Similarly, functions `GetMaxY` and `GetMaxX` also complete in $O(m + n)$ time. Since there are at most $2\sqrt{m}$ key cn-pairs by Lemma 7.11, the total time cost of `Core-Approx` is bounded by $O(\sqrt{m}(n + m))$.

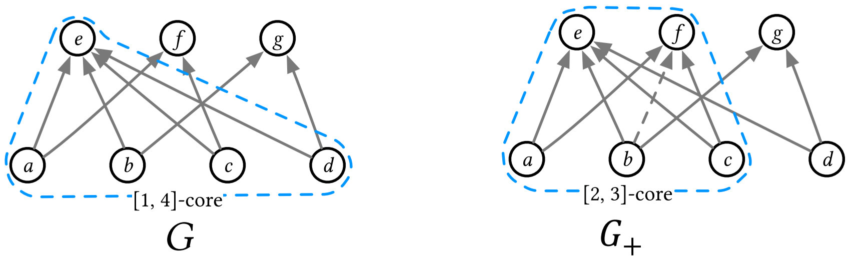


Fig. 13. Illustration of the $[x^*, y^*]$-core before and after the insertion of edge (b, f).

8 DDS MAINTENANCE IN DYNAMIC DIRECTED GRAPHS

In this section, we study the problem of DDS maintenance in dynamic directed graphs, where vertices and edges are inserted and deleted frequently. In line with previous works of DS maintenance in undirected graphs [5, 9, 18, 37, 68], we focus on maintaining the 2-approximation solution in dynamic directed graphs. More precisely, we mainly consider two kinds of graph updates, namely, edge insertion and edge deletion, since vertex insertion (respectively, vertex deletion) can be regarded as a sequence of edge insertions (respectively, edge deletions) preceded (respectively, followed) by the insertion (respectively, deletion) of an isolated vertex. We aim to efficiently output the updated DDS whenever a graph update is made.

To maintain the DDS, a simple solution is to recompute the 2-approximation DDS from scratch when a graph update is made. However, since the graph is typically large and frequently updated, this approach is impractical due to its inefficiency. Another approach is that we dynamically maintain all the $[x, y]$-cores to keep track of the approximation DDS, i.e., the $[x^*, y^*]$-core. Nevertheless, this requires much extra memory to index all $[x, y]$-cores, and also much computation cost, since many $[x, y]$-cores may change even after a single edge insertion or edge deletion. Obviously, this is not an elegant solution either, as we only care about the $[x^*, y^*]$-core. To improve the efficiency, we conduct a careful investigation and observe that inserting or deleting a single edge in a graph G usually results in little or no change on the DDS, since the DDS is often much smaller than G, which motivates us to update the DDS locally by examining the subgraph containing the inserted or deleted edge. Based on this observation, we develop efficient algorithms that focus on maintaining the $[x^*, y^*]$-core for edge insertion and edge deletion in Section 8.1 and Section 8.2, respectively. For ease of presentation, we use G_+ (respectively, G_-) to represent the updated graph after an edge $e = (u, v)$ is inserted to (respectively, removed from) the directed graph G.

8.1 Edge Insertion

When a new edge $e = (u, v)$ is inserted to G, under what condition would the $[x^*, y^*]$-core of G fail to be a 2-approximation DDS in G_+? Only if the new edge is contained in an $[x^*_+, y^*_+]$-core such that $x^*_+ \cdot y^*_+ > x^* \cdot y^*$, which means that $[x^*_+, y^*_+]$ is the new maximum cn-pair in G_+. As a result, we can derive a necessary condition for e to be contained in such an $[x^*_+, y^*_+]$-core:

$$d^+_{G_+}(u) \cdot d^-_{G_+}(v) > x^* \cdot y^*. \tag{11}$$

In other words, when the above condition is satisfied, we need to further check whether the inserted edge can contribute to the $[x^*_+, y^*_+]$-core; otherwise, the $[x^*, y^*]$-core of G is still a 2-approximation DDS of G_+, which means that the DDS does not change. We further illustrate this by Example 8.1.

Example 8.1. Consider the graph G in Figure 13 with the insertion of a new edge (b, f). In G, the maximum cn-pair is $[1, 4]$, so the subgraph induced by $\{a, b, c, d\}$ and $\{e\}$ is a 2-approximation solution. After inserting (b, f), G becomes G_+ and we have $d^+_{G_+}(b) \cdot d^-_{G_+}(f) = 9 > 4$. Therefore, edge (b, f) may contribute to the $[x^*_+, y^*_+]$-core where $x^*_+ \cdot y^*_+ > 1 \times 4$. After further check, we will find the maximum cn-pair of G_+ is $[2, 3]$ and (b, f) is contained in the $[2, 3]$-core. Since $2 \times 3 > 1 \times 4$, we update the 2-approximation DDS as the $[2, 3]$-core of G_+.

Next, we discuss how to examine whether the $[x^*_+, y^*_+]$-core exists in G_+, when the inserted edge (u, v) satisfies the condition in Equation (11). The possible ranges of the values of x^*_+ and y^*_+ can be restricted by Lemma 8.2.

Lemma 8.2. *Given a directed graph G and an edge $e = (u, v)$ to be inserted, let $[x^*_+, y^*_+]$ be the maximum cn-pair of G_+ and $[x^*, y^*]$ be the maximum cn-pair of G. Only if the following conditions are satisfied, the $[x^*, y^*]$-core could fail to be a 2-approximation DDS and the $[x^*_+, y^*_+]$-core gives the 2-approximation DDS in G_+:*

$$\begin{cases} x^*_+ \cdot y^*_+ > x^* \cdot y^*, \\ x^*_+ \le d^+_{G_+}(u), \\ y^*_+ \le d^-_{G_+}(v). \end{cases} \tag{12}$$

Proof. First, the $[x^*, y^*]$-core still exists in G_+, so the $x^*_+ \cdot y^*_+ \ge x^* \cdot y^*$. If $x^*_+ \cdot y^*_+ = x^* \cdot y^*$, then the current $[x^*, y^*]$-core is still a valid 2-approximation solution for G_+. Thus, we need to check whether $x^*_+ \cdot y^*_+ > x^* \cdot y^*$ satisfies.

Second, if $x^*_+ \cdot y^*_+ > x^* \cdot y^*$, then the inserted edge $e = (u, v)$ must be contained in the $[x^*_+, y^*_+]$-core. Otherwise, the $[x^*_+, y^*_+]$-core is a subgraph of G, which contradicts $x^*_+ \cdot y^*_+ > x^* \cdot y^*$. Hence, we have $x^*_+ \le d^+_{G_+}(u)$ and $y^*_+ \le d^-_{G_+}(v)$. □

Let us take the edge insertion in the graph of Figure 13 as an example to illustrate Lemma 8.2. The possible scope of the maximum cn-pair in G_+, i.e., $[x^*_+, y^*_+]$, is depicted in the shaded area of Figure 14, where $x \le 3$, $y \le 3$, and $xy > 4$. This is because after the insertion, the indegree of b and outdegree of f are 3. The (b, f) is supposed to be in the $[x^*_+, y^*_+]$-core, so in this case we have $x^*_+ \le 3$ and $y^*_+ \le 3$.

To find the new maximum cn-pair in G_+, we only need to iterate over the skyline cn-pairs in the possible scope (the shaded area in Figure 14) stated by Lemma 8.2. In Example 8.1, we first fix $x = 2$ and find $y = 3$, where $[2, 3]$ is a valid cn-pair in G_+. Then, we increase x to 3 and find that there is no cn-pair in the scope. Hence, $[2, 3]$ is the maximum cn-pair in G_+, and we can update the 2-approximation solution as the $[2, 3]$-core. The number of skyline cn-pairs of G_+ falling into the area delineated by Lemma 8.2 is much smaller than the total number of skyline cn-pairs of G_+, which explains, to a certain extent, why the dynamic algorithm is much faster than recomputing from scratch.

Based on the above analysis, we propose a maintenance algorithm to handle the case of edge insertion, denoted by `Approx-Ins`. Algorithm 9 presents the details. The algorithm first checks whether the necessary condition in Equation (11) is satisfied (line 2). If yes, then the algorithm will check whether there exists the $[x^*_+, y^*_+]$-core in G_+ by iterating x from x_{lower} to x_{upper} to search the skyline cn-pairs in the possible scope. In detail, to find the first skyline cn-pair, $[2, 3]$, we first find the largest second element $y = 3$ while keeping $x \ge 2$ and $y \ge 3$ to satisfy $x^*_+ \cdot y^*_+ > x^* \cdot y^*$ (line 6) and then find the largest first element $x = 2$ while keeping $y \ge 3$ and $x \ge 2$ (line 8).

Given two thresholds, x and y, and two vertices to be contained, u and v, the function `GetMaxYLocal` (lines 11–31) computes the cn-pair satisfying the following conditions simultaneously:

(1) the first element of the cn-pair is at least x,
(2) the second element of the cn-pair is at least y,

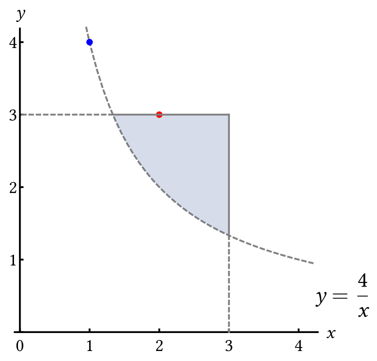


Fig. 14. Finding the maximum cn-pair in G_+.

(3) the second element is maximized,
(4) the corresponding $[x, y]$-core of the cn-pair contains both vertices u and v.

If these conditions can be fulfilled, the algorithm will return the second element y of the cn-pair; otherwise, it will return 0.

Specifically, we first examine whether there is an $[x, y]$-core containing both u and v by generating an $[x, y]$-core starting from u and v. To generate the core, we maintain two queues, Q_{expand} and Q_{shrink} (line 12), two sets, S and T, denoting that the core $G_+[S, T]$ (line 13), and two degree arrays, deg_{in} and deg_{out}, recording indegrees and outdegrees for vertices in T and S, respectively (line 14). To start, we push (u, S) and (v, T) to Q_{expand} (line 15). Here, we use the pair (u, S) (respectively, (v, T)) to denote $u \in S$ (respectively, $v \in T$), because a same vertex could be contained in both S and T. Next, we keep processing the vertices in Q_{expand} and Q_{shrink} until both sets become empty (lines 16–28). If Q_{shrink} is not empty, then there exists a vertex in S being removed whose outdegree is less than x or a vertex in T being removed whose indegree is less than y (line 17). Thus, we need to check whether removing this vertex will result in the removal of its adjacent vertices. If yes, then we remove the affected vertices and push them to Q_{shrink} (lines 19–21). If Q_{expand} is not empty, then there is a vertex newly added to S or T because its outdegree or indegree in G_+ is larger than x or y, respectively (line 22). We need to check whether its outdegree or indegree in $G_+[S, T]$ still fulfills the criterion (lines 24–26). If yes, then we will add the adjacent vertices with indegrees larger or equal than y of the vertex to T and push them into Q_{expand} for further examination assuming the current vertex belongs to S (line 28). Otherwise, the vertex will be removed from S or T and passed into Q_{shrink} to be further processed (line 26). When $G_+[S, T]$ becomes stable, we will check whether u and v are still in the subgraph (line 26). If yes, then we maximize the second element y of the cn-pair by increasing y and removing the vertices that do not fulfill the degree criteria (line 30); otherwise, the algorithm will return 0 (line 31). Similarly, we can have the function `GetMaxXLocal` (lines 32–33).

Complexity. In the worst case, all the skyline cn-pairs are iterated in `Approx-Ins`, and each time the whole graph is visited. Thus, the time complexity of the algorithm is $O(\sqrt{m}(n + m))$, which is the same as that of `Core-Approx`. However, only a small portion of the graph needs to be processed in practice, as demonstrated in the experimental part, so it performs very fast.

8.2 Edge Deletion

We begin with an interesting observation: After removing an edge $e = (u, v)$ from the graph G, if the $[x^*, y^*]$-core of G is not the core with the maximum cn-pair in G_-, which means that the

ALGORITHM 9: `Approx-Ins`

```
   Input  : $G = (V, E)$, the maximum cn-pair of $G$ (i.e., $[x^*, y^*]$), and an inserted edge $e = (u, v)$
   Output : The 2-approximation DDS $\widetilde{D}_+$ of $G_+$
1  $G_+ = (V, E_+) \leftarrow$ insert $(u, v)$ into $G$;
2  if $d^+_{G_+}(u) \cdot d^-_{G_+}(v) \leq x^* \cdot y^*$ then return the $[x^*, y^*]$-core of $G$ ;
3  $x_{lower} \leftarrow \lfloor \frac{x^* \cdot y^*}{d^-_{G_+}(v)} \rfloor + 1$, $x_{upper} \leftarrow d^+_{G_+}(u)$ ;
4  for $x \leftarrow x_{lower}$ to $x_{upper}$ do
5      $y \leftarrow \lfloor \frac{x^* \cdot y^*}{x} \rfloor + 1$;
6      $y \leftarrow$ GetMaxYLocal($G_+, x, y, u, v$);
7      if $y = 0$ then continue;
8      $x \leftarrow$ GetMaxXLocal($G_+, x, y, u, v$);
9      if $x \cdot y > x^* \cdot y^*$ then $x^* \leftarrow x, y^* \leftarrow y$ ;
10 return the $[x^*, y^*]$-core of $G_+$;
11 Function GetMaxYLocal($G_+, x, y, u, v$):
12     initialize two queues $Q_{expand}$ and $Q_{shrink}$;
13     initialize two sets $S \leftarrow \{u\}$ and $T \leftarrow \{v\}$;
14     initialize the degree arrays $deg_{in}$ and $deg_{out}$ for the new $[x, y]$-core consisting of $(u, v)$;
15     push $(u, S)$ and $(v, T)$ to $Q_{expand}$ ;        // $(u, S)$, $(v, T)$ indicate $u \in S$, $v \in T$
16     while $Q_{expand}$ or $Q_{shrink}$ is not empty do
17         if $Q_{shrink}$ is not empty then
18             $Q_{shrink}$ pops out the front pair as $(w, I)$;
               /* Assume $I = S$. If $I = T$, interchange proper variables.  */
19             foreach $p \in \{p | (w, p) \in E_+ \wedge p \in T\}$ do
20                 $deg_{in}[p] \leftarrow deg_{in}[p_T] - 1$;
21                 if $deg_{in}[p] < y$ then push $(p, T)$ to $Q_{shrink}$, remove $p$ from $T$;
22         else
23             $Q_{expand}$ pops out the front pair as $(w, I)$;
               /* Assume $I = S$. If $I = T$, interchange proper variables.  */
24             $V_{tmp} \leftarrow \{p | (w, p) \in E_+ \wedge d^-_{G_+}(p) \geq y \wedge (p, T)$ is never pushed into $Q_{shrink}\}$;
25             $deg_{out}[w] \leftarrow |V_{tmp}|$;
26             if $deg_{out}[w] < x$ then add $(w, I)$ to $Q_{shrink}$, remove $w$ from $S$;
27             else
28                 foreach $p \in \{p | p \in V_{tmp} \wedge p \notin T\}$ do push $(p, T)$ to $Q_{expand}$, add $p$ to $T$ ;
29     if $u \in S \wedge v \in T$ then
30         return $\max_{u, v \in \text{the } [x,y]\text{-core}} y$ ;   // via increasing $y$ and removing vertices from $S$
           and $T$
31     else return 0 ;
32 Function GetMaxXLocal($G_+, x, y, u, v$):
33     reuse lines 12–31 by replacing "$\max_{u, v \in \text{the } [x,y]\text{-core}} y$" with "$\max_{u, v \in \text{the } [x,y]\text{-core}} x$" in line 30.
```

2-approximation DDS solution changes, then e must be contained in the $[x^*, y^*]$-core of G. In other words, if e is not in the $[x^*, y^*]$-core of G, then we do not need to update the 2-approximation DDS. Inspired by this observation, we design a maintenance algorithm for processing the case of edge deletion, denoted by `Approx-Del`.

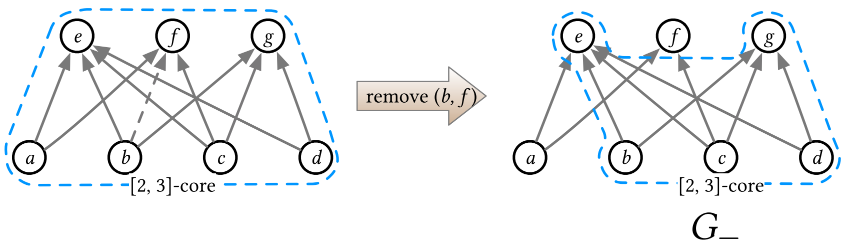


Fig. 15. Illustration of the $[x^*, y^*]$-core before and after the deletion of edge (b, f).

ALGORITHM 10: `Approx-Del`

```
   Input  : $G = (V, E)$, the maximum cn-pair of $G$ (i.e., $[x^*, y^*]$), and a removed edge $e = (u, v)$
   Output : The 2-approximation DDS $\widetilde{D}_-$ of $G_-$
1  $G_- \leftarrow$ delete $(u, v)$ from $G$;
2  if $d^+_G(u) \cdot d^-_G(v) \le x^* \cdot y^*$ then return the $[x^*, y^*]$-core of $G$ ;
3  if $(u, v) \notin$ the $[x^*, y^*]$-core of $G$ then return the $[x^*, y^*]$-core of $G$ ;
4  $\widetilde{D}_- = G[S, T] \leftarrow$ the $[x^*, y^*]$-core of $G$;
5  remove edge $(u, v)$ from $\widetilde{D}_-$;
6  while $\exists w \in S, d^+_{\widetilde{D}_-}(w) < x^* \vee \exists p \in T, d^-_{\widetilde{D}_-}(p) < y^*$ do
7      if $\exists w \in S, d^+_{\widetilde{D}_-}(w) < x^*$ then remove $w$ from $S$ and the outgoing edges of $w$ from $\tilde{D}_-$;
8      if $\exists p \in T, d^-_{\widetilde{D}_-}(p) < y^*$ then remove $p$ from $T$ and the incoming edges of $p$ from $\tilde{D}_-$;
9  if $S \neq \emptyset \wedge T \neq \emptyset$ then return $\widetilde{D}_-$ ;
10 else return Core-Approx($G_-$) ;                    // Call Algorithm 8 to recompute
```

Algorithm 10 presents the key steps of `Approx-Del`. The first condition we check is whether $d^+_G(u) \cdot d^-_G(v) < x^* \cdot y^*$. If this inequation holds, then we can safely remove the edge, and $[x^*, y^*]$ is still the maximum cn-pair for G_- (line 2). Otherwise, we further check whether the $[x^*, y^*]$-core of G consists of (u, v). If no, then the edge can be removed safely, and the $[x^*, y^*]$-core of G is still the 2-approximation solution (line 3); if yes, then the 2-approximation DDS must change, and the algorithm will assign the $[x^*, y^*]$-core of G to $\widetilde{D}_- = G[S, T]$ (line 4). Next, we keep removing the vertices from S whose outdegrees are less than x^* and vertices from T whose indegrees are less than y^* (lines 5–8). After the removal process, $\widetilde{D}_-$ will be returned if $\widetilde{D}_-$ is not empty (line 9). Otherwise, `Core-Approx` will be invoked to re-compute the 2-approximation DDS in G_- (line 10).

Example 8.3. Consider the graph G in Figure 15 and an edge (b, f). Before removing (b, f) from G, the maximum cn-pair of G is $[2, 3]$, so the subgraph induced by $S = \{a, b, c, d\}$ and $T = \{e, f, g\}$ is a 2-approximation solution. Notice that (b, f) is in the $[2, 3]$-core of G. Since $d^+_G(b) \cdot d^-_G(f) = 9 \ge 6$, the approximate DDS must change, and we need to keep removing the vertices whose outdegrees are less than 2 from S and the vertices whose indegrees are less than 3 from T. After the removal process, $S = \{b, c, d\}$ and $T = \{e, g\}$ are still not empty. Hence, $[2, 3]$ is still the maximum cn-pair of G_- and the subgraph induced by S and T is the 2-approximation solution of G_-.

Complexity. In the worst case, the 2-approximation algorithm `Core-Approx` needs to be re-invoked. Consequently, the time complexity of `Approx-Del` is $O(\sqrt{m}(n+m))$, which is the same as that of `Core-Approx`. Nevertheless, in practice, it runs much faster than recomputing from scratch, since the removed edge is not always in the $[x^*, y^*]$-core of G.

Table 4. Unweighted Directed Graphs Used in Our Experiments

Dataset	Full name	Category	$\lvert V \rvert$	$\lvert E \rvert$
MO [28]	moreno-oz	Human Social	217	2,672
TC [1]	maayan-faa	Infrastructure	1,226	2,615
OF [61]	openflights	Infrastructure	2,939	30,501
AD [55]	advogato	Social	6,541	51,127
AM [46]	amazon	E-commerce	403,394	3,387,388
AR [58]	amazon-ratings	E-commerce	3,376,972	5,838,041
BA [59]	baidu-zhishi	Hyperlink	2,141,300	17,643,697
TW [14]	twitter-mpi	Social	52,579,682	1,963,263,821

9 EXPERIMENTS

We now present the experimental results. We first discuss the setup in Section 9.1, then describe the results of exact and approximation algorithms in Sections 9.2–9.3, followed by the results of DDS maintenance over dynamic graphs in Section 9.4 and WDDS computation on weighted graphs in Section 9.5, and finally present case studies in Section 9.6.

9.1 Setup

In the following, we introduce the datasets[8] and the algorithms that are used for our experimental evaluation of DDS algorithms, DDS maintenance algorithms, and WDDS algorithms, respectively.

9.1.1 Unweighted Graphs and DDS Algorithms. For unweighted directed graphs, we use eight real datasets [44], and we report the numbers of vertices and edges of each dataset in Table 4. These graphs cover various domains, including social networks (e.g., Twitter and Advogato), e-commerce (e.g., Amazon), and infrastructures (e.g., flight networks).

We compare following exact DDS algorithms:

- `Core-Exact` is our proposed exact algorithm for DDS computation (Section 6.2).
- `DC-Exact` is our proposed exact algorithm for DDS computation (Section 6.3).
- `Exact` [42] is the state-of-the-art exact algorithm, which is also recapped in Section 4.1.

We also compare following approximation DDS algorithms:

- `Core-Approx` is our proposed 2-approximation algorithm for DDS computation (Section 7).
- `KS-Approx` [42] is an approximation algorithm whose approximation ratio was misclaimed, which is also recapped in Section 4.2.
- `FKS-Approx` (in the Appendix) is the fixed version provided by the authors of Reference [42]. Its time complexity is $O(n \cdot (n + m))$, with an approximation ratio of 2.
- `PM-Approx` [6] is a parameterized approximation algorithm. Note that we use its default parameters in Reference [6] in our experiments ($\delta = 2$, $\epsilon = 1$).
- `BS-Approx` [15] is a 2-approximation algorithm.
- `BS-Approx-`δ [15] is an adaptation of `BS-Approx` providing 2δ-approximation result with time complexity of $O(\frac{\log n}{\log \delta}(n + m))$. In our experiments, we use $\delta = 2$, which is also adopted by `PM-Approx` in Reference [6].

[8] All datasets are available online at http://konect.uni-koblenz.de/networks/, http://snap.stanford.edu/data/index.html and http://networkrepository.com.

Table 5. Dynamic Graphs Used in Our Experiments

Dataset	Full name	Category	$\|V\|$	$\|E\|$	Number of updates
YH [67]	ia-yahoo-messages	Social	100,001	905,199	3,179,718
SU [63]	sx-superuser	Online QA	194,085	924,886	1,443,339
CA [47]	ca-cit-HepTh	Collaboration	22,908	2,444,798	2,673,133
SX [63]	sx-stackoverflow	Online QA	2,601,977	36,233,450	63,497,050

Table 6. Weighted Directed Graphs Used in Our Experiments

Dataset	Full name	Category	$\|V\|$	$\|E\|$	Weight meanings
FD [64]	foodweb-baydry	Trophic	128	2,137	Carbon exchange
AF [60]	opsahl-usairport	Infrastructure	1,574	28,236	Number of flights
BX [79]	bookcrossing-rating	E-commerce	263,757	433,652	Rating
MV [44]	movielens-1m	Movie	9,746	1,000,210	Rating
LI [45]	libimseti	Social	220,970	17,359,346	Rating

9.1.2 Dynamic Graphs and DDS Maintenance Algorithms. To evaluate DDS maintenance algorithms, we use four real datasets [44, 48, 67], and we report the numbers of vertices and edges of each dataset in Table 5. Those graphs have timestamps associated with edges. For example, SX [63] is a temporal network of interactions on the stack exchange web site Stack Overflow.[9] Each timestamp corresponds to an update, and the total number of timestamps on each dataset is also reported in Table 5. We also use two static graphs (i.e., AM and BA) with synthetic updates to evaluate the DDS maintenance algorithms.

We conduct experiments using following 2-approximation DDS maintenance algorithms:

- `Core-Approx` is a baseline method by recomputing the DDS from scratch for each update.
- `Approx-Ins` is the DDS maintenance algorithm for handling edge insertion (Section 8.1).
- `Approx-Del` is the DDS maintenance algorithm for handling edge deletion (Section 8.2)

9.1.3 Weighted Directed Graphs and WDDS Algorithms. For weighted directed graphs, we use three real datasets [44] and report the numbers of vertices and edges, as well as the weight meanings, e.g., the weights denote the numbers of flights in the flight network [60], of each dataset in Table 6. These graphs cover domains of e-commerce (e.g., Amazon), infrastructures (e.g., flight network), and social networks (e.g., Libimseti.cz). We compare the following WDDS algorithms:

- `W-Exact` is the baseline exact WDDS algorithm, extended from `Exact`.
- `WDC-Exact` is our proposed exact WDDS algorithm, which exploits wcore-based optimization and divide-and-conquer strategy.
- `WCore-Approx` is our proposed 2-approximation WDDS algorithm.

All the algorithms above are implemented in C++ with STL used. We run all the experiments on a machine having an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz processor and 256 GB memory, with Ubuntu installed.

9.2 Exact Algorithms of DDS Problem

In Figure 16, we report the efficiency results of exact algorithms on the five smallest datasets (i.e., MO, TC, OF, AD, and AM). As these solutions cannot finish reasonably on larger datasets, we do

[9]https://stackoverflow.com.

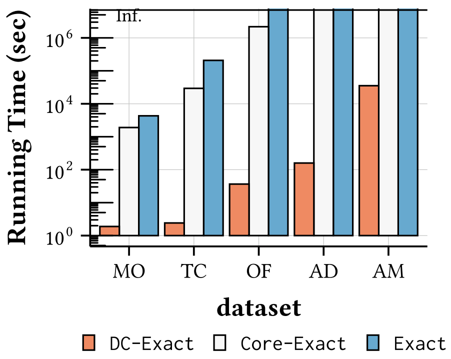


Fig. 16. Efficiency of exact DDS algorithms.

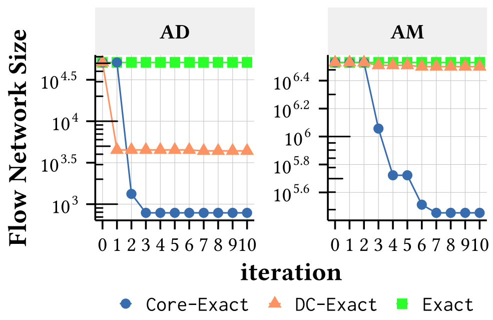


Fig. 17. Flow network sizes in exact algorithms.

not report their results here. Note that Exact and Core-Exact cannot compute the DDS within 600 hours on OF, AD, and AM. Clearly, Core-Exact is at least 2× and up to 100× faster than the state-of-the-art exact algorithm Exact. This is mainly because Core-Exact locates the DDS in some $[x, y]$-cores, which are often much smaller than the entire graph, so the flow networks built on these cores become much smaller, resulting in less time cost on computing the minimum st-cut of the flow networks.

We further investigate how the flow network size (number of edges) changes in the first 10 iterations of the binary search in exact algorithms. Figure 17 reports the flow network size of these three algorithms on two datasets, i.e., AD and AM. Clearly, we can observe that the size of the flow network in Core-Exact is reduced significantly as the iteration goes on, while the flow network size of Exact does not change during these iterations. Thus, we conclude that the $[x, y]$-cores are indeed effective for locating the DDS in some smaller subgraphs, allowing the exact DDS to be computed more efficiently. The sizes of flow networks created in DC-Exact are larger than those in Core-Exact because, in DC-Exact, the flow network is built based on the union of $[x, y]$-cores for all possible values of a in the range of $[a_l, a_r]$.

Meanwhile, from Figure 16, we can see that DC-Exact is up to six and five orders of magnitude faster than Exact and Core-Exact, respectively. The main reason is that DC-Exact exploits a divide-and-conquer strategy, which dramatically reduces the number of $a = \frac{|S|}{|T|}$ examined, as illustrated in Figure 9. To further analyze the speedup of DC-Exact over Exact, we report the numbers of values of a examined in DC-Exact and Exact, which equal to the numbers of times of invoking the loop of binary search. As discussed in Section 6, the total numbers of values of a examined in Exact and DC-Exact are n^2 and k, respectively. The values of n^2, k, and $\frac{n^2}{k}$ on the first five datasets are reported in Table 7. Clearly, n^2 is much larger than k. For example, on the dataset AM, n^2 is over 10 orders of magnitude larger than k. Thus, DC-Exact runs much faster than Exact.

In addition, we have implemented the three exact algorithms (Exact, Core-Exact, and DC-Exact) with/without the help of the parametric max-flow algorithm [29]. We test the running time of both versions (i.e., with the parametric max-flow algorithm and without it) and report the speedup provided by the parametric max-flow algorithm for each algorithm in Table 8, where NA denotes the algorithm cannot finish within 200 hours on the corresponding dataset. We can see that the parametric max-flow algorithm is indeed helpful for further improving the efficiency. Besides, we can observe Core-Exact gains the most from the parametric max-flow algorithm followed by Exact, while DC-Exact gains little, because it is already several orders of magnitude faster than Core-Exact and Exact even without the help of the parametric max-flow algorithm.

Table 7. The Total Numbers of Values of a Examined in DC-Exact and Exact

Dataset	n^2 (Exact)	k (DC-Exact)	$\frac{n^2}{k}$
MO	4.71×10^4	16	2.94×10^3
TC	1.50×10^6	23	6.54×10^4
OF	8.64×10^6	35	2.47×10^5
AD	4.28×10^7	81	5.28×10^5
AM	1.63×10^{11}	13	1.25×10^{10}

Table 8. The Speedup Provided by the Parametric Max-flow Algorithm [29]

Dataset	Exact	Core-Exact	DC-Exact
MO	3.90	5.02	1.31
TC	2.76	2.95	1.17
OF	3.96	6.80	1.41
AD	NA	NA	1.46
AM	NA	NA	1.05

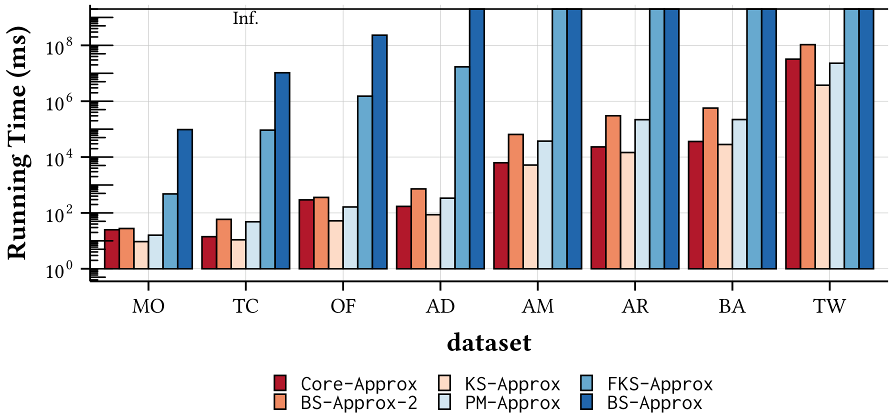


Fig. 18. Efficiency of approximation algorithms.

9.3 Approximation Algorithms of DDS Problem

In Figure 18 (respectively, Figure 19), we show the running time of (respectively, the densities of the approximate DDS's returned by) approximation algorithms on all the eight datasets, where bars touching the upper boundaries (respectively, "NA" labels) mean that the corresponding algorithms cannot finish within 200 hours. We can make the following observations: (1) BS-Approx and FKS-Approx provide high quality results but are the two most inefficient algorithms, because their time complexities, i.e., $O(n^2(n + m))$ and $O(n(n + m))$, are higher than those of other algorithms. (2) KS-Approx is the most efficient one on almost all the datasets, since it takes only linear time cost, i.e., $O(n + m)$. However, its approximation ratio could be larger than 2, as analyzed in Section 4. (3) Core-Approx is the second most efficient one on almost all the datasets, followed by PM-Approx. (4) BS-Approx-δ provides high-quality results, although its theoretical approximation ratio is larger than 2. Its running time is more than 6× longer than Core-Approx,

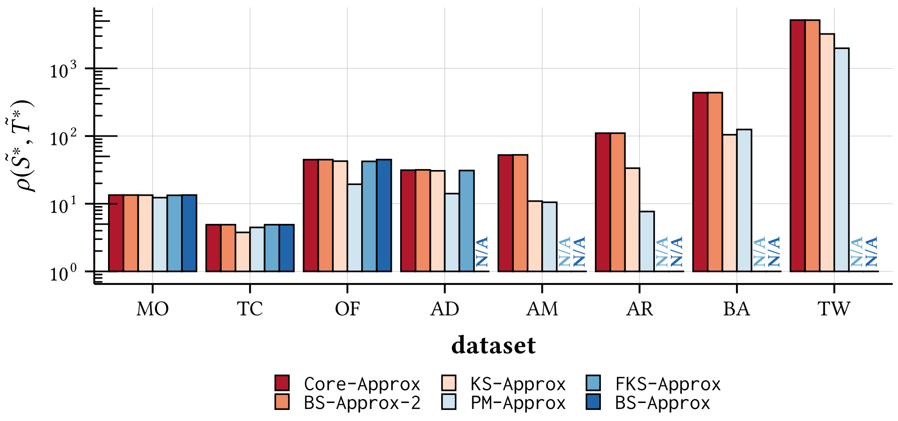


Fig. 19. $\rho_{\tilde{S}^*,\tilde{T}^*}$ returned by all approximation algorithms.

Table 9. Analyzing 2-approximation DDS Algorithms

Dataset	MO	TC	OF	AD	AM	AR	BA	TW
n	217	1,226	2,939	6,541	4.03×10^5	3.38×10^6	2.14×10^6	5.26×10^7
n^2	4.71×10^4	1.50×10^6	8.64×10^6	4.28×10^7	1.63×10^{11}	1.14×10^{13}	4.59×10^{12}	2.76×10^{15}
$\sqrt{m}$	52	51	174	226	1,840	2,416	4,200	44,308
γ	8	3	27	18	10	26	60	2,221
$\frac{n}{\gamma}$	27	408	108	363	4.03×10^4	1.30×10^5	3.57×10^4	2.37×10^4
$\frac{n^2}{\gamma}$	5.89×10^3	5.01×10^5	3.20×10^5	2.38×10^6	1.63×10^{10}	4.39×10^{11}	7.64×10^{10}	1.24×10^{12}

although its theoretical time complexity is lower. (5) Among all the 2-approximation algorithms, Core-Approx is the fastest one, since it is up to six orders of magnitude faster than BS-Approx, and three orders of magnitude faster than FKS-Approx. Moreover, it can process billion-scale graphs. Thus, it obtains not only high-quality results but also achieves high efficiency.

Next, we focus on the 2-approximation algorithms and perform a deeper investigation on why Core-Approx is significantly faster than others. Recall that the time complexities of BS-Approx, FKS-Approx, and Core-Approx are $O(n^2(n+m))$, $O(n\cdot(n+m))$, and $O(\gamma(n+m))$, respectively, so we can roughly use $\frac{n^2}{\gamma}$ and $\frac{n}{\gamma}$ to explain the speedup of Core-Approx over BS-Approx and FKS-Approx, respectively. In Table 9, we report the values of $\frac{n}{\gamma}$ and $\frac{n^2}{\gamma}$ on all the eight datasets. Clearly, on the first four datasets, the values of $\frac{n^2}{\gamma}$ and $\frac{n}{\gamma}$ are roughly the same as the times of speedup in Figure 18. Based on this observation, we conjecture that for other larger datasets (e.g., AR), Core-Approx could be up to 10 and 5 orders of magnitude faster than BS-Approx and FKS-Approx, respectively, although we did not get the actual running time of BS-Approx and FKS-Approx in our experiments.

Besides, we compare the actual approximation ratios of all the six approximation algorithms. Specifically, for each graph, we first obtain the exact DDS using DC-Exact, then compute the approximate DDS's using these approximation algorithms, and get the actual approximation ratios (i.e., the density of the exact DDS over those of approximate DDS's). Note that a smaller actual approximation ratio indicates that the corresponding solution has a better approximation quality w.r.t. the optimal DDS, i.e., higher accuracy w.r.t. the exact DDS. We report the actual approximation ratios of each algorithm on the first five datasets in Figure 20. Clearly, the actual approximation ratios of Core-Approx, BS-Approx-δ, BS-Approx, and FKS-Approx are indeed smaller than their theoretical worst-case approximation ratios. The theoretical approximation ratio of BS-Approx-δ

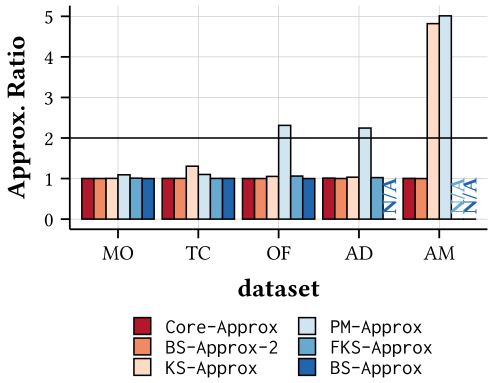


Fig. 20. The actual approximation ratios of all approximation algorithms.

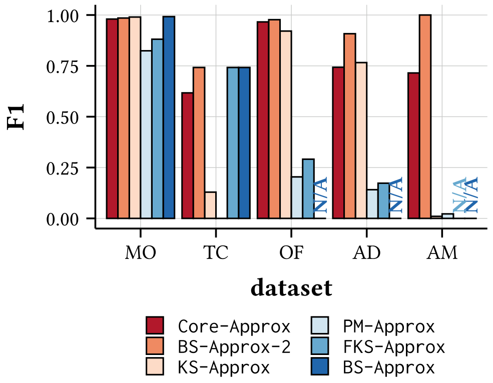


Fig. 21. F1 scores of all approximation algorithms.

Table 10. The Values of $\frac{|S^*|}{|T^*|}$ on the First Five Datasets

Dataset	MO	TC	OF	AD	AM
$\frac{\lvert S^*\rvert}{\lvert T^*\rvert}$	1.04	5.67×10^{-2}	1.01	2.32	2.47×10^{3}

is 4, but its actual approximation ratios over five datasets are quite low and sometimes marginally smaller than our Core-Approx, which indicates that the technique used in BS-Approx-δ to reduce the number of values of $\frac{|S|}{|T|}$ will not introduce much error in practice. Besides, we can see that KS-Approx cannot provide 2-approximation results on some datasets (e.g., AM). This well confirms our finding that KS-Approx may fail to report a 2-approximation result in some cases, as discussed in Section 4.2. In addition, the actual approximation ratios of PM-Approx could be larger than 2 but less than 5: notice that its theoretical approximation ratio is $2\delta(1+\epsilon)$, which under the default parameter setting of $\delta = 2, \epsilon = 1$, recommended by Reference [6], evaluates to 8.

Figure 21 supplements Figure 20 with F1 scores of the subgraphs returned by the approximation algorithms w.r.t. the DDS returned by DC-Exact. Note that BS-Approx and FKS-Approx did not finish in reasonable time on some datasets, so we could not compute the overlap that they could have achieved. From Figure 21, we have following observations: (1) In many cases, DDS's of lower density found by some algorithms tend to overlap less with the optimal DDS. (2) In some cases, surprisingly, lower approximation ratio is associated with higher overlap with the optimal DDS. But this pattern is not consistent: e.g., the actual approximation ratio of KS-Approx is lower (i.e., better) than that of PM-Approx, while the F1 score of KS-Approx is lower than that of PM-Approx. We remark that the (actual) approximation ratio is the primary criterion to evaluate the approximation algorithms, as our goal is to find a subgraph as dense as possible.

To further analyze why KS-Approx leads to very high approximation ratios, we revisit its algorithm steps and find that it actually restricts the search on the subgraph, in which the minimum outdegree and minimum indegree of all vertices are close to each other; in other words, when $\frac{|S^*|}{|T^*|}$ = 1, KS-Approx tends to find an approximate DDS with higher accuracy. In Table 10, we report the ratio of $|S^*|$ over $|T^*|$ on the first five datasets. We can observe that on most of the datasets, when $\frac{|S^*|}{|T^*|}$ is close to 1, KS-Approx tends to find DDS's with higher accuracy than Core-Approx, while when $\frac{|S^*|}{|T^*|}$ largely deviates from 1, it will perform worse.

Another interesting point concerns the comparison between Core-Approx and BS-Approx-δ. On the one hand, BS-Approx-δ has a higher theoretical approximation ratio (i.e., 4) than the

Table 11. Efficiency of 2-approximation DDS Maintenance Algorithms

Type	Dataset	Insert			Delete		
		`Core-Approx`	`Approx-Ins`	Speedup	`Core-Approx`	`Approx-Del`	Speedup
Real	YH	1.54s	0.003s	498.75	1.54s	1.79×10^{-6}	861,960.89
	SU	1.54s	0.03s	48.83	1.57s	5.15×10^{-6}s	305,876.45
	CA	61.01s	8.4×10^{-7}s	72,634,000	61.36s	6.94×10^{-6}s	8,830,429
	SX	288.59s	2.62s	110.24	261.34s	8.03×10^{-5}s	3,256,235.4
Synthetic	AM	4.34s	3.88×10^{-5}s	111,979.78	4.37s	1.17×10^{-6}s	3,728,412
	BA	27.62s	1.96×10^{-2}s	1,409.92	27.58s	1.10×10^{-1}s	250.91

theoretical ratio of `Core-Approx` (i.e., 2). But the actual approximation ratios of `BS-Approx-`δ on the eight datasets are comparable to `Core-Approx`. In detail, `BS-Approx-`δ is marginally smaller on four datasets; `Core-Approx` is smaller on one dataset, and they are equally good on the rest three datasets. This may indicate that the technique used in `BS-Approx-`δ to reduce the number of trials of different $\frac{|S|}{|T|}$ by testing different powers of δ may not introduce large errors in practice. On the other hand, `BS-Approx-`δ has a lower theoretical time complexity (i.e., $O(\log_2 n(n+m))$ via $\delta = 2$) given that the theoretical complexity of `Core-Approx` is $O(\gamma(n+m)) = O(\sqrt{m}(n+m))$. But, in practice, `Core-Approx` is 6.6× faster than `BS-Approx-`δ on average over eight datasets. This is mainly because $\gamma \ll \sqrt{m}$, usually by several orders, as shown in Table 9, in practice.

9.4 DDS Maintenance Algorithms

In this experiment, we test the performance of DDS maintenance algorithms by using four real datasets (YH, SU, CA, and SX) and two synthetic datasets (AM and BA).

For **real datasets**, to evaluate `Approx-Ins`, we insert the latest 1,000 edges to the graph according to the timestamps on the edges and report the average running time for each edge insertion. Similarly, to evaluate `Approx-Del`, we delete the oldest 1,000 existing edges from the graph and report the average running time for each edge deletion.

For **synthetic datasets**, to evaluate `Approx-Del`, we randomly delete 1,000 distinct existing edges from the graph and report the average running time for each edge deletion. Similarly, to evaluate `Approx-Ins`, we insert the 1,000 removed edges to the graph and present the average running time for each edge insertion.

Table 11 reports the efficiency results. From the table, we can observe that `Approx-Ins` is up to seven orders of magnitude faster than recomputing from scratch when handling edge insertion, because it can quickly locate the new maximum cn-pair in the updated graph. `Approx-Del` is up to six orders of magnitude faster than recomputing from scratch when handling edge deletion, since it can avoid the recomputation of many unnecessary cases as discussed in Section 8.2.

Another interesting point is that the speedup of our insertion algorithm differs among different datasets. The reason why the speedup differs across different datasets is that if the degrees of the endpoints of the newly added edges are small compared to x^* and y^* from the current $[x^*, y^*]$-core, the maintenance process will be stopped immediately. If this case happens often, then the speedup will be pretty significant. Otherwise, the speedup will be relatively modest. There are almost 100% newly added edges in CA whose endpoint degree products are smaller than $x^* \cdot y^*$, while there are around 58% newly added edges in YH satisfying such condition. Further, CA is much larger than YH, making CA need more time to recompute from scratch. Hence, under the combined effect of several reasons discussed above, our dynamic insertion algorithm achieves around seven orders of magnitude speedup on CA and 500× speedup on YH.

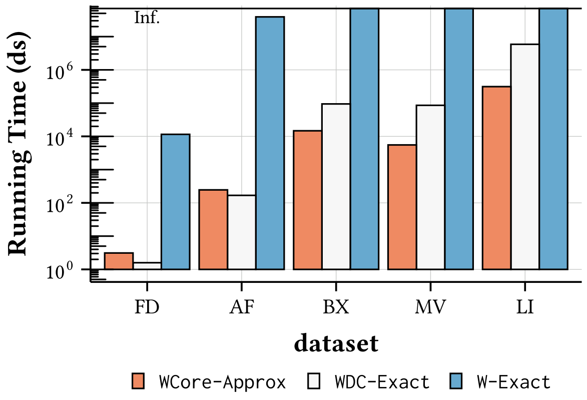


Fig. 22. Efficiency of `WCore-Approx`, `WDC-Exact`, and `W-Exact`.

Table 12. Statistics about `WCore-Approx` on Weighted Directed Graphs

Dataset	FD	AF	BX	MV	LI
Approximation ratio	1.00	1.07	1.02	1.14	1.31
ξ (number of skyline wcn-pairs)	126	745	190	1,346	2,484
k in `WDC-Exact` (number of binary searches)	8	22	207	52	204

9.5 Algorithms of WDDS Problem

We test three WDDS algorithms (i.e., `WCore-Approx`, `WDC-Exact`, and `W-Exact`) and show the efficiency results on five datasets in Figure 22, where bars touching the upper boundary mean that the corresponding algorithms cannot finish within 300 hours. We can see that `WDC-Exact` and `WCore-Approx` are up to five orders of magnitude faster than `W-Exact`. This demonstrates that the core-based optimization and divide-and-conquer strategy are indeed effective for accelerating the computation of the WDDS. On the three larger datasets, `WCore-Approx` is at least 10× faster than `WDC-Exact`. To further analyze the performance, we report some statistics (e.g., the approximation ratio and ξ) of running `WCore-Approx` on weighted graphs in Table 12. Clearly, `WCore-Approx` provides high-quality approximate results (i.e., the approximation ratios are quite close to 1.0).

Besides, from Figure 22, we see that `WCore-Approx` is slightly slower than `WDC-Exact` on two datasets (i.e., FD and AF). The reasons can be explained by the statistics in Table 12: (1) Let $\frac{\xi}{n}$ denote the ratio of the number of the skyline wcn-pairs over the number of vertices in the graph. Then, we can see that the value of $\frac{\xi}{n}$ on FD and AF is much higher than that on BX, MV, and LI, which may make `WCore-Approx` perform worse than `WDC-Exact` on these two datasets. (2) k is much smaller than ξ on these two datasets. As k and ξ are the key parameters affecting the efficiency of `WDC-Exact` and `WCore-Approx`, respectively, except for the graph size, this is one reason why `WCore-Approx` performs worse than `WDC-Exact` on FD and AF. (3) Although BX is much larger than AF, its ξ is smaller than that of AF, because the scope of edge weights is different, i.e., BX only has 10 choices for the ratings ($\{1, 2, \dots, 10\}$), while AF has hundreds of different edge weights due to the variety of flight numbers. This also implies that when the graph has a wide range of edge weights, `WDC-Exact` may be the best option, since it may achieve higher quality results and higher efficiency than `WCore-Approx`. However, for large-scale datasets with limited edge weight cardinality, `WCore-Approx` is still the first choice, as the performance on two more massive datasets (i.e., MV and LI) shows.

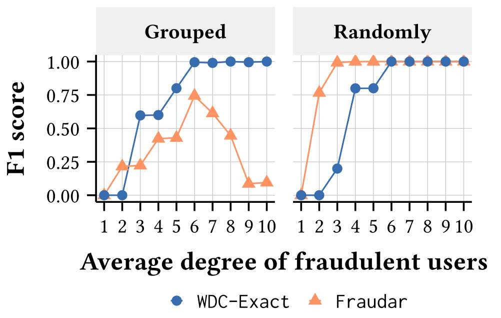


Fig. 23. The performance of fake review detection.

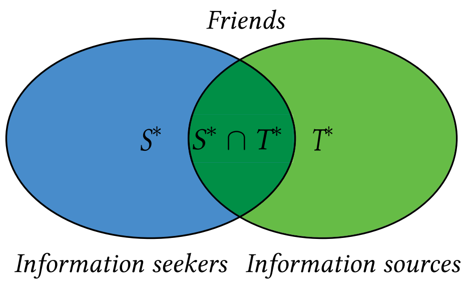


Fig. 24. Information seekers and sources.

9.6 Case Studies

9.6.1 Fake Reviewer Detection. To show the effectiveness of the WDDS solution, we conduct a case study by considering one of its downstream applications, namely, fake review detection. We use the WDDS to detect frauds on a real graph and show that it achieves comparable performance with the state-of-the-art fraud detection algorithm, i.e., `Fraudar` [35]. Given a bipartite graph of users and the products they review, `Fraudar` detects the fake followers by identifying a dense subgraph from the bipartite graph. Note our algorithms naturally extend to the bipartite graph by making one vertex set as the source and the other as the target.

Specifically, we follow the experimental setup in Reference [35] and consider the dataset AR (see Table 4), which is the Amazon review graph consisting of users and products, where all the edges are directed from users to products. We first randomly select 2,000 users and 2,000 products, with around 2,400 edges (similar to Reference [35]). Then, we inject 200 fraudulent users and 200 fraudulent products with different average degrees for fraudulent users. When injecting the fraudulent edges, we follow two strategies:

(1) **Grouped.** The fraudulent users and products are grouped into several small equal-size groups, and the injected edges are all within each group.
(2) **Randomly.** The injected edges are inserted randomly among all fraudulent users and products.

In addition, for each edge (u, v) in the graph G, we set its weight to $W(u, v) = \frac{1}{\sqrt{d^-_G(v)+c}}$, which is a column-weighting strategy proposed in Reference [35] (c=5). Notice that more injected edges imply the higher average degree of fraudulent users.

Next, we use `WDC-Exact` and `Fraudar` to detect the fraudulent users and compute the average $F1$ score of their results (where $F1 = \frac{2\times\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}$) over five trials, which are depicted in Figure 23. We can see that for the first injecting strategy, `WDC-Exact` achieves higher $F1$ scores than `Fraudar` in most cases, while for the second injecting strategy, `WDC-Exact` has the same performance when the average degree of fraudulent users is at least 6. `Fraudar` performs better only when the edges are injected randomly and the density is very low. Meanwhile, as the number of injected edges increases, `WDC-Exact` achieves higher $F1$ scores. Thus, `WDC-Exact` is indeed effective for fake review detection, since it achieves comparable performance with `Fraudar`. In addition, we reckon that the performance of fraud detection can be further improved if the human-in-the-loop process (e.g., asking humans to further check the users and products in the WDDS) is exploited.

9.6.2 Roles of Vertices in S^* and T^*. Here, we use an example to investigate the different roles played by the vertices in S^*, T^*, and $S^* \cap T^*$, as depicted in Figure 24. In the approximate DDS returned by Core-Approx in TW (i.e., twitter-mpi, a follow-link network), S^* contains 162,418 vertices, T^* contains 236 vertices, and $S^* \cap T^*$ contains 7 vertices. Inspired by Reference [40], we can treat the vertices in S as *information seekers* and the vertices in T^* as *information sources*. Further, the vertices in $S^* \cap T^*$ are probably *friends,* as they likely followed each other. Although we cannot trace back the vertices to the exact users in Twitter (as the dataset is anonymized), we envision each scenario based on the current result: Most users in S^* are information seekers; most users in T^* are information sources, which could be news agencies and celebrities; the users in $S^* \cap T^*$ are both information seekers and sources, which need to seek information from the news agencies and other celebrities and share their opinions with their followers. Hence, the flexibility of not restricting the disjointness of S^* and T^* allows us to detect whether some users play both roles, i.e., information seekers and sources. This example illustrates that our problem can result in interesting analysis about the relationship between S^* and T^* (e.g., finding out who interacts with each other in $S^* \cap T^*$). If we enforce S^* and T^* to be disjoint, then we cannot perform this kind of analysis.

10 CONCLUSION

In this article, we study the densest subgraph discovery on directed graphs (DDS problem for short). We first review existing algorithms and discuss their limitations. We show that a previous algorithm [42], which was claimed to achieve an approximation of 2, fails to satisfy the approximation guarantee. To boost the efficiency of finding DDS, we introduce a novel dense subgraph model, namely, $[x, y]$-core, on directed graphs, and establish bounds on the density of the $[x, y]$-core. We then propose a core-based exact algorithm and further optimize it by incorporating a divide-and-conquer strategy. Besides, we find that the $[x^*, y^*]$-core, where x^*y^* is the maximum value of xy for all the $[x, y]$-cores, is a good approximation solution to the DDS problem, with a theoretical guarantee. To compute the $[x^*, y^*]$-core, we develop an efficient algorithm, which is more efficient than all the existing 2-approximation algorithms. Extensive experiments on eight real large datasets show that both our exact and approximation algorithms for DDS are up to six orders of magnitude faster than the state-of-the-art approaches. Besides, we develop efficient non-trivial DDS maintenance algorithms for handling dynamic graphs, and our DDS maintenance algorithms can provide up to five orders of magnitude speedup compared to recomputing the DDS from scratch. We further extend the algorithms to find weighted DDS (or WDDS) on weighted directed graphs. In addition, we present a case study that demonstrates that the WDDS solution is useful for fraud detection.

We will investigate how to efficiently find the DDS with size constraints on directed graphs in the future. Similarly, it is interesting to find the DDS with constraints on the overlap between S^* and T^*, e.g., enforcing S^* and T^* to be disjoint, or requiring at least (or at most) k vertices in $S^* \cap T^*$. Sometimes, users are interested in finding more than one densest subgraph. Hence, it is worth investigating how to define and study the top-k DDS's problem. It would also be interesting to study the DS problem on other types of graphs, e.g., attributed graphs and heterogeneous information networks. Another interesting future research direction is to develop algorithms for solving the bi-clique-based DS problem [10] on bipartite graphs.

APPENDIX

A A FIXED ALGORITHM OF KS-APPROX

Here is the fix for KS-Approx provided by its author Prof. Barna Saha. We name it FKS-Approx in this article. Algorithm 11 presents the details. First, the algorithm lets S contain all vertices in V (line 2). In the outer loop, each time the algorithm assigns V to T (line 4), lets H be the

ALGORITHM 11: FKS-Approx

```
Input  : $G = (V, E)$
Output : An approximate DDS $\widetilde{D}$
1  $\rho^* \leftarrow 0$;
2  $S \leftarrow V$;
3  while $|S| \neq 0$ do
4      $T \leftarrow V, H \leftarrow G[S, T]$;
5      $u \leftarrow \arg\min_{w \in S} d_H^+(w)$;
6      remove $u$ from $S$ and its corresponding edges from $H$;
7      while $|T| \neq 0$ do
8          $v \leftarrow \arg\min_{w \in T} d_H^-(w)$;
9          remove $v$ from $T$ and its corresponding edges from $H$;
10         if $\rho(S,T) > \rho^*$ then $\rho^* \leftarrow \rho(S,T)$, $\widetilde{D} \leftarrow G[S,T]$;
11 reuse lines 2–10 by interchanging $S$ with $T$, $u$ with $v$, $d_H^+(w)$ with $d_H^-(w)$, but $G[S,T]$ keeping
   unchanged;
12 return $\tilde{D}$;
```

subgraph induced by S and T (line 4), and removes the vertex with minimum outdegree from S and its outgoing edges from H (lines 5–6). Then, in the inner loop, each time we remove the vertex with the minimum indegree from T and its incoming edges from H (lines 8–9). Next, the algorithm checks whether the maximum density can be updated by the density of $H = G[S,T]$; if yes, then the maximum density ρ and the approximate DDS $\tilde{D}$ will be updated as $\rho(S,T)$ and $G[S,T]$, respectively (line 10). Then, the algorithm will repeat the above process with T in the outer loop and S in the inner loop (line 11). Finally, the algorithm returns $\tilde{D}$ as the approximate DDS.

Now Theorem 2 in Reference [42] will go through, as there will exist a subgraph where the outdegrees of its vertices in S will be larger than λ_o and the indegree of its vertices in T will be larger than λ_i, simultaneously (refer to Reference [42]). Obviously, the time complexity of FKS-Approx is $O(n(n+m))$.

REFERENCES

[1] Federal Aviation Administration. 2019. Air Traffic Control System Command Center. Retrieved from https://www.faa.gov.

[2] Ravindra K. Ahuja, M. Kodialam, A. K. Mishra, and J. B. Orlin. 1997. Computational investigations of maximum flow algorithms. *European Journal of Operational Research* 97, 3 (1997), 509–542.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Internet: Diameter of the world-wide web. *Nature* 401, 6749 (1999), 130.

[4] Reid Andersen. 2010. A local algorithm for finding dense subgraphs. *Trans. Alg.* 6, 4 (2010), 1–12.

[5] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.* 23, 2 (2014), 175–199.

[6] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *Proc. VLDB Endow.* 5, 5 (2012), 454–465.

[7] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[8] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. 2010. Detecting high log-densities: An O (n 1/4) approximation for densest k-subgraph. In *STOC*. ACM, 201–210.

[9] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *STOC*. 173–182.

[10] John Adrian Bondy, Uppaluri Siva Ramachandra Murty et al. 1976. *Graph Theory with Applications*. Vol. 290. Macmillan London.

[11] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting densest subgraphs without flow computations. In *WWW*. ACM, 573–583.

[12] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *WSDM*. ACM, 95–106.

[13] Andrea Capocci, Vito D. P. Servedio, Francesca Colaiori, Luciana S. Buriol, Debora Donato, Stefano Leonardi, and Guido Caldarelli. 2006. Preferential attachment in the growth of social networks: The internet encyclopedia Wikipedia. *Phys. Rev. E* 74, 3 (2006), 036116.

[14] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. 2010. Measuring user influence in Twitter: The million follower fallacy. In *ICWSM*. 10–17.

[15] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*. Springer, 84–95.

[16] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large scale density-friendly graph decomposition via convex programming. In *WWW*. 233–242.

[17] Soroush Ebadian and Xin Huang. 2019. Fast algorithm for k-truss discovery on public-private graphs. Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019.

[18] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient densest subgraph computation in evolving graphs. In *WWW*. 300–310.

[19] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *VLDB J.* 26, 6 (2017), 803–828.

[20] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *Proc. VLDB Endow.* 10, 6 (2017), 709–720.

[21] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proc. VLDB Endow.* 9, 12 (2016), 1233–1244.

[22] Yixiang Fang, Reynold Cheng, Siqiang Luo, Jiafeng Hu, and Kai Huang. 2017. C-Explorer: Browsing communities in large graphs. *Proc. VLDB Endow.* 10, 12 (2017), 1885–1888.

[23] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2019. A survey of community search over big graphs. *VLDB J.* (2019), 1–40.

[24] Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, and Xiaojun Chen. 2019. On spatial-aware community search. *IEEE Trans. Knowl. Data Eng.* 31, 4 (2019), 783–798.

[25] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2019. Effective and efficient community search over large directed graphs. *IEEE Trans. Knowl. Data Eng.* 31, 11 (2019), 2093–2107.

[26] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *Proc. VLDB Endow.* 13, 6 (Feb. 2020).

[27] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *Proc. VLDB Endow.* 12, 11 (2019), 1719–1732.

[28] Linton Clarke Freeman, Cynthia Marie Webster, and Deirdre M. Kirke. 1998. Exploring social structure using dynamic three-dimensional color images. *Soc. Netw.* 20, 2 (1998), 109–118.

[29] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* 18, 1 (1989), 30–55.

[30] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2013. D-cores: Measuring collaboration of directed graphs based on degeneracy. *Knowl. Inf. Syst.* 35, 2 (2013), 311–343.

[31] Aristides Gionis and Charalampos E. Tsourakakis. 2015. Dense subgraph discovery: KDD 2015 tutorial. In *KDD*. ACM, 2313–2314.

[32] Andrew V. Goldberg. 1984. *Finding a Maximum Density Subgraph*. University of California Berkeley, CA.

[33] Andrew V. Goldberg. 2008. The partial augment–relabel algorithm for the maximum flow problem. In *ESA*. Springer, 466–477.

[34] G. T. Heineman, G. Pollice, and S. Selkow. 2008. Algorithms in a nutshell: A practical guide[M]. O'Reilly Media, Inc.

[35] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. Fraudar: Bounding graph fraud in the face of camouflage. In *KDD*. ACM, 895–904.

[36] Jiafeng Hu, Reynold Cheng, Kevin Chen-Chuan Chang, Aravind Sankar, Yixiang Fang, and Brian Y. H. Lam. 2019. Discovering maximal motif cliques in large heterogeneous information networks. In *ICDE*. IEEE, 746–757.

[37] Shuguang Hu, Xiaowei Wu, and T. H. Hubert Chan. 2017. Maintaining densest subsets efficiently in evolving hypergraphs. In *CIKM*. 929–938.

[38] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[39] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate closest community search in networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287.

[40] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. 2007. Why we Twitter: Understanding microblogging usage and communities. In *9th WebKDD and 1st SNA-KDD*. ACM, 56–65.

[41] Ravi Kannan and V. Vinay. 1999. *Analyzing the Structure of Large Graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn.

[42] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *ICALP*. Springer, 597–608.

[43] Jon M. Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *J. ACM* 46, 5 (1999), 604–632.

[44] Jérôme Kunegis. 2013. — Koblenz network collection. In *WWW*. 1343–1350. Retrieved from http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf.

[45] Jérôme Kunegis, Gerd Gröner, and Thomas Gottron. 2012. Online dating recommender systems: The split-complex number approach. In *4th ACM RecSys Workshop onRSWEB*. 37–44.

[46] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Trans. Web* 1, 1 (2007).

[47] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*. ACM, 177–187.

[48] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.

[49] Xiaodong Li, Tsz Nam Chan, Reynold Cheng, Caihua Shan, Chenhao Ma, and Kevin Chang. 2019. Motif paths: A new approach for analyzing higher-order semantics between graph nodes. *HKU Tech. Rep.* 3 (2019), 4.

[50] Xiaodong Li, Reynold Cheng, Kevin Chen-Chuan Chang, Caihua Shan, Chenhao Ma, and Hongtai Cao. 2021. On analyzing graphs with motif-paths. *Proc. VLDB Endow.* 14, 6 (2021), 1111–1123.

[51] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *SIGMOD*.

[52] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: A motif counting algorithm for uncertain graphs. *Proc. VLDB Endow.* 13, 2 (2019), 155–168.

[53] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient algorithms for densest subgraph discovery on large directed graphs. In *SIGMOD*. 1051–1066.

[54] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. Efficient directed densest subgraph discovery. *ACM SIGMOD Rec.* 50, 1 (2021), 33–40.

[55] Paolo Massa, Martino Salvetti, and Danilo Tomasoni. 2009. Bowling alone and trust decline in social network sites. In *IEEE DASC*. 658–663.

[56] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable large near-clique detection in large-scale networks via sampling. In *KDD*. ACM, 815–824.

[57] Atsushi Miyauchi and Akiko Takeda. 2018. Robust densest subgraph discovery. In *ICDM*. IEEE, 1188–1193.

[58] Arjun Mukherjee, Bing Liu, and Natalie Glance. 2012. Spotting fake reviewer groups in consumer reviews. In *WWW*. 191–200.

[59] Xing Niu, Xinruo Sun, Haofen Wang, Shu Rong, Guilin Qi, and Yong Yu. 2011. Zhishi.me—Weaving Chinese linking open data. In *ISWC*. 205–220.

[60] Tore Opsahl. 2011. Why anchorage is not (that) important: Binary ties and sample selection.

[61] Tore Opsahl, Filip Agneessens, and John Skvoretz. 2010. Node centrality in weighted networks: Generalizing degree and shortest paths. *Soc. Netw.* 3, 32 (2010), 245–251.

[62] James B. Orlin. 2013. Max flows in O (nm) time, or better. In *STOC*. 765–774.

[63] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *WSDM*. 601–610.

[64] J. Patricio. 2000. *Network Analysis of Trophic Dynamics in South Florida Ecosystems, FY 99: The Graminoid Ecosystem.* Master's Thesis. University of Coimbra, Coimbra, Portugal.

[65] B. Aditya Prakash, Ashwin Sridharan, Mukund Seshadri, Sridhar Machiraju, and Christos Faloutsos. 2010. Eigenspokes: Surprising patterns and scalable community chipping in large graphs. In *PAKDD*. Springer, 435–448.

[66] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally densest subgraph discovery. In *KDD*. ACM, 965–974.

[67] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*. Retrieved from http://networkrepository.com.

[68] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Amitabh Trehan. 2012. Dense subgraphs on dynamic networks. In *DISC*. Springer, 151–165.

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

[70] Martin W. Schein and Milton H. Fohrman. 1955. Social dominance relationships in a herd of dairy cattle. *Brit. J. Anim. Behav.* 3, 2 (1955), 45–55.

[71] Stephen B. Seidman. 1983. Network structure and minimum degree. *Soc. Netw.* 5, 3 (1983), 269–287.
[72] Bintao Sun, Maximilien Dansich, Hubert Chan, and Mauro Sozio. 2020. KClist++: A simple algorithm for finding k-clique densest subgraphs in large graphs. *Proc. VLDB Endow.* 13, 10 (2020), 1628–1640.
[73] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly graph decomposition. In *WWW*. 1089–1099.
[74] Charalampos Tsourakakis. 2015. The K-clique densest subgraph problem. In *WWW*. 1122–1132.
[75] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In *KDD*. ACM, 104–112.
[76] Charalampos E. Tsourakakis, Tianyi Chen, Naonori Kakimura, and Jakub Pachocki. 2019. Novel dense subgraph discovery primitives: Risk aversion and exclusion queries. In *ECML PKDD*. Springer, 378–394.
[77] Zhiwei Zhang, Xin Huang, Jianliang Xu, Byron Choi, and Zechao Shang. 2019. Keyword-centric community search. In *ICDE*. 422–433.
[78] Dong Zheng, Jianquan Liu, Rong-Hua Li, Cigdem Aslay, Yi-Cheng Chen, and Xin Huang. 2017. Querying intimate-core groups in weighted graphs. In *ICSC*. 156–163.
[79] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. 2005. Improving recommendation lists through topic diversification. In *WWW*. 22–32.

Received August 2020; revised July 2021; accepted August 2021