In-depth Analysis of Densest Subgraph Discovery in a Unified Framework [Experiment, Analysis & Benchmark]

Yingli Zhou
The Chinese University of Hong Kong, Shenzhen, China
yinglizhou@link.cuhk.edu.cn

Qingshuo Guo
The Chinese University of Hong Kong, Shenzhen, China
qingshuoguo@link.cuhk.edu.cn

Yi Yang
The Chinese University of Hong Kong, Shenzhen, China
yiyang3@link.cuhk.edu.cn

Yixiang Fang
The Chinese University of Hong Kong, Shenzhen, China
fangyixiang@cuhk.edu.cn

Chenhao Ma
The Chinese University of Hong Kong, Shenzhen, China
machenhao@cuhk.edu.cn

Laks V.S. Lakshmanan
The University of British Columbia, Canada
laks@cs.ubc.ca

ABSTRACT

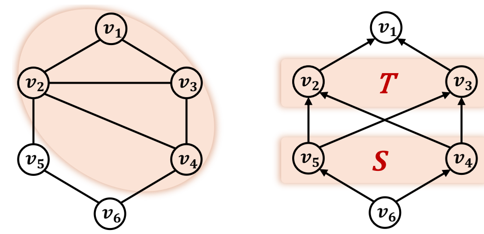
As a fundamental topic in graph mining, *Densest Subgraph Discovery (DSD)* has found a wide spectrum of real applications. Several DSD algorithms, including exact and approximation algorithms, have been proposed in the literature. However, these algorithms have not been systematically and comprehensively compared under the same experimental settings. In this paper, we first summarize a unified framework to incorporate all DSD algorithms from a high-level perspective. We then extensively compare representative DSD algorithms over a range of graphs – from small to billion-scale – and examine the effectiveness of all methods, providing a thorough analysis of DSD algorithms. As a byproduct of our experimental analysis, we are also able to identify new variants of the DSD algorithms over undirected graphs, by combining existing techniques, which are up to 10× faster than the state-of-the-art algorithm with the same accuracy guarantee. Finally, based on the findings, we offer promising research opportunities. We believe that a deeper understanding of the behavior of existing algorithms can provide new valuable insights for future research.

PVLDB Reference Format:
Yingli Zhou, Qingshuo Guo, Yi Yang, Yixiang Fang, Chenhao Ma, and Laks V.S. Lakshmanan. In-depth Analysis of Densest Subgraph Discovery in a Unified Framework [Experiment, Analysis & Benchmark]. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TalionS/DensestSubgraph.

1 INTRODUCTION

Graph data are often used to model relationships between objects in various real-world applications [2, 20, 48, 51, 64]. For example, the Facebook friendship network can be modelled as an undirected graph by mapping users to vertices and friendships to edges [20]; In X (formerly known as Twitter), a directed edge can represent the "following" relationship between two users [48]; The Web network itself can also be modelled as a vast directed graph [2]. Figures 1 (a) and (b) depict an undirected graph and a directed graph respectively.



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

Figure 1: Examples of undirected and directed graphs.

As a fundamental problem in graph mining, the *densest subgraph discovery (DSD)* problem aims to discover a very "dense" subgraph from a given graph [56, 61]. More precisely, given an undirected graph, the DSD problem [40] asks for a subgraph with the highest *density*, defined as the number of edges over the number of vertices in the subgraph, and it is often termed the densest subgraph (DS). The DSD problem lies at the core of graph mining [8, 39], and is widely used in many areas. For instance, in social networks, the DS discovered can be used to detect communities [19, 91], reveal fake followers [11], and identify echo chambers and groups of actors engaged in spreading misinformation [55, 61]. In e-commerce networks [11], the DS is useful for detecting fake accounts. In graph databases, the DSD is a building block for solving many graph problems, such as reachability queries [21] and motif detection [35, 83]. In biological data analysis, DSD solutions have been shown to be useful in identifying regulatory motifs in genomic DNA [35] and gene annotation graphs [83]. Besides, the DSD problem is closely related to other fundamental graph problems, such as network flow and bipartite matching [85]. Due to the theoretical and practical importance, researchers from the database, data mining, computer science theory, and network communities have designed efficient and effective solutions to the DSD problem.

In Table 1, we categorize representative DSD algorithms by their computation models, main methods, and approximation ratio guarantee. The exact algorithms include maximum flow and convex programming-based approaches; the approximation algorithms include peeling-based, convex programming-based, and network flow-based approaches. After a careful literature review, we make the following observations. First, no prior work has proposed a unified framework to abstract the DSD solutions and identify key performance factors. Second, existing works focus on evaluating

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: Classification of existing DSD works.

Graph type	Algorithm	Key technique	Complexity		Approx. ratio	# Iteration	Optimization		
			Time complexity	Space complexity			Early termination	Graph reduction	Parallel friendly
Undirected graphs	`FlowExact [40]`	Network flow	$O(\log n \cdot t_{\text{Flow}})$	$O(m)$	1	N/A	✗	✗	✗
	`CoreExact [33]`	Network flow	$O(\log n \cdot t_{\text{Flow}})$	$O(m)$	1	N/A	✓	✓	✗
	`FWExact [25]`	Convex Programming	$O(T \cdot m + \log T \cdot t_{\text{Flow}})$	$O(m)$	1	$\Omega(T)$	✓	✗	✓
	`MWUExact [42]`	Convex programming	$O(T \cdot m + \log T \cdot t_{\text{Flow}})$	$O(m)$	1	$\Omega(T)$	✓	✗	✓
	`FISTAExact [42]`	Convex programming	$O(T \cdot m + \log T \cdot t_{\text{Flow}})$	$O(m)$	1	$\Omega(T)$	✓	✗	✓
	`Greedy [17]`	Peeling	$O(m+n)$	$O(m)$	2	N/A	✗	✗	✗
	`CoreApp [33]`	Peeling	$O(m+n)$	$O(m)$	2	N/A	✗	✗	✗
	`Greedy++ [15]`	Peeling	$O(T \cdot m \log n)$	$O(m)$	$(1+\epsilon)$	$\Omega(\frac{\Delta(G)}{\rho^*_G \epsilon^2})$	✗	✗	✗
	`FWApp [25]`	Convex programming	$O(T \cdot m)$	$O(m)$	$(1+\epsilon)$	$\Omega(\frac{mn\Delta(G))}{\epsilon^2})$	✓	✗	✗
	`MWUApp [42]`	Convex programming	$O(T \cdot m)$	$O(m)$	$(1+\epsilon)$	$\Omega(x)$	✓	✗	✗
	`FISTAApp [42]`	Convex programming	$O(T \cdot m)$	$O(m)$	$(1+\epsilon)$	$\Omega(\frac{\sqrt{mn\Delta(G)}}{\epsilon^2})$	✓	✗	✗
	`FlowApp*[95]`	Network flow	$O(T \cdot m \log m)$	$O(m)$	$(1+\epsilon)$	$\Omega(\frac{\log m}{\epsilon})$	✓	✓	✗
Directed graphs	`DFlowExact [53]`	Network flow	$O(n^2 \cdot t_{\text{Flow}})$	$O(m)$	1	N/A	✗	✗	✗
	`DCExact [68]`	Network flow	$O(k \cdot t_{\text{Flow}})$	$O(m)$	1	N/A	✓	✓	✗
	`DFWExact [69]`	Convex programming	$O(T \cdot t_{\text{Fw}})$	$O(m)$	1	$\Omega(T)$	✓	✓	✗
	`DGreedy [17]`	Peeling	$O(n^2 \cdot (n+m))$	$O(m)$	2	N/A	✗	✗	✗
	`XYCoreApp [68]`	Peeling	$O(\sqrt{m} \cdot (n+m))$	$O(m)$	2	N/A	✗	✗	✗
	`WCoreApp [62]`	Peeling	$O(\Delta(G) \cdot m)$	$O(m)$	2	N/A	✗	✗	✓
	`DFWApp [69]`	Convex Programming	$O(T \cdot \log_{1+\epsilon} nm)$	$O(m)$	$(1+\epsilon)$	$\Omega(\frac{m \cdot \kappa}{\epsilon^2})$	✓	✓	✓

⋆ Note: n and m denote the numbers of vertices and edges in the graph respectively; $\epsilon > 0$ is a real value.
⋆ Note: $\Delta(G)$ denotes the highest degree of undirected graph G; κ is an integer proportional to the maximum value of highest out-degree and in-degree of directed graph D.

the overall performance, but not individual components. Third, there is no existing comprehensive comparison between all these algorithms.

Our work. To address the above issues, in this paper we conduct an in-depth study on sequential DSD algorithms[1]. We first propose a unified framework with three modules, namely *graph reduction*, *vertex weight update* (`VWU`), and *candidate subgraph extract and verify* (`CSV`), which capture the core ideas of all existing algorithms. Given a graph G and an error threshold ϵ, *graph reduction* aims to locate the DS in a small subgraph; `VWU` aims to update vertex weights over T iterations; and `CSV` extracts a candidate subgraph based on vertex weights and verifies if it satisfies the ϵ error requirement. Under this framework, we systematically compare 12 (resp. 7) representative algorithms for undirected (resp. directed) graphs, respectively. We conduct comprehensive experiments on both real-world and synthetic datasets and provide an in-depth analysis.

In summary, our principal contributions are as follows.

- Summarize a unified framework for DSD solutions from a high-level perspective (Section 3);
- Comprehensively examines DSD algorithms for both undirected and directed graphs respectively (Sections 4 and 5);
- Conduct extensive experiments from different angles using various datasets, providing a thorough analysis of DSD algorithms. Based on our analysis, we identify new variants of DSD algorithms over undirected graphs, by combining existing techniques, which significantly outperform the state-of-the-art (Section 6);
- Summarize lessons learned and propose practical research opportunities that can facilitate future studies (Section 7).

In Section 2, we present the preliminaries and introduce a unified DSD framework in Section 3. Section 8 reviews related work while Section 9 summarizes the paper.

[1]We refer readers to [87] for a comparison of parallel DSD algorithms.

2 PRELIMINARIES

We first provide the definitions of DSD problems over both undirected and directed graphs, i.e., UDS and DDS problems respectively, and then present their convex program (CP) formulations.

2.1 Problem definitions

Table 2: Notations and meanings.

Notation	Meaning
$G = (V, E)$	An undirected graph with vertex set V and edge set E
$D = (V, E)$	A directed graph with vertex set V and edge set E
$N(v, G)$	The set of neighbors of a vertex v in G
$d_G(v)$	The degree of v in G, i.e., $d_G(v)$=$\lvert N(v, G)\rvert$
$d^+_D(v), d^-_D(v)$	The out-degree and in-degree of v in D, respectively
$G[S]$	The subgraph of G induced by vertices in S
$D[S, T]$	The subgraph of D induced by vertices in S and T
$\mathcal{D}(G)$	The densest subgraph of G
$\rho(S, T)$	The density of subgraph $D[S, T]$
k^*	The largest k such that the k-core in G exists.
$\Delta(G)$	The highest degree of G.

We denote an undirected graph by $G = (V, E)$, where $|V| = n$ and $|E| = m$ are the numbers of vertices and edges of G, respectively. The set of neighbors of a vertex u in G is denoted by $N(u, G)$, and the degree of u is $d_G(u) = |N(u, G)|$. Given a vertex set S, we use $G[S] = (S, E(S))$ to denote the subgraph of G induced by S, where $E(S)$= $\{(u, v) \in E \mid u, v \in S\}$ denotes the set of edges in G contained in S. For a given undirected graph H, we denote its sets of vertices and edges by $V(H)$ and $E(H)$, respectively.

*Definition 2.1 (**Density of undirected graph** [40]).* Given an undirected graph $G = (V, E)$, its density $\rho(G)$ is defined as the number of edges over the number of vertices, i.e., $\rho(G) = \frac{|E|}{|V|}$.

Method	**Stage (1):** `ReduceGraph`	**Stage (2):** `VWU`	**Stage (3):** `CSV`
`FlowExact` `CoreExact`	no reduction locate graph into $\lceil \underline{\rho} \rceil$-core	compute the maximum flow	❶ extract the minimum cut; ❷ verify if it is optimal;
`FlowApp`	locate graph into $\lceil \underline{\rho} \rceil$-core	perform the blocking flow	❶ extract the residual graph; ❷ verify the approximation ratio;
CP-based	no reduction	optimize $\mathsf{CP}(G)$;	❶ extract the maximum prefix sum set; ❷ verify if it is exact or satisfies the approximation ratio criteria;
Peeling-based	no reduction	iteratively remove vertices	❶ extract the subgraph with the highest density during the peeling process;

Table 3: Overview of the three stages of the existing UDS algorithms.

Problem 1 (**UDS problem** [33, 40, 90]). *Given an undirected graph G, find the subgraph $\mathcal{D}(G)$ whose density is the highest among all the possible subgraphs, which is also called the undirected densest subgraph (UDS).*

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

*Definition 2.2 (**Directed graph density** [50, 53, 68, 70]).* Given a directed graph D=(V, E) and two vertex sets S and T, the density of an (S, T)-induced subgraph is defined as $\rho(S,T) = \frac{|E(S,T)|}{\sqrt{|S||T|}}$.

Problem 2 (**DDS PROBLEM** [8, 17, 39, 50, 53]). *Given a directed graph D, find the subgraph $\mathcal{D}(D)$=$D[S^*, T^*]$ whose corresponding density is the highest among all the possible (S,T)-induced subgraphs, also called the directed densest subgraph (DDS).*

Denote the density of $\mathcal{D}(G)$ and $\mathcal{D}(D)$ by ρ_G^* and ρ_D^* respectively. E.g., in Figures 1 (a) and (b), the subgraphs in the dashed ellipses are the UDS and DDS respectively, with ρ_G^*=5/4 and ρ_D^*=2.

3 A UNIFIED FRAMEWORK

In this section, we develop a unified framework, consisting of three stages: *Graph reduction*, *Vertex Weight Update* (`VWU`), and *Candidate subgraph Extract and Verify* (`CSV`), which can cover all existing UDS and DDS algorithms, as shown in Algorithm 8.

Algorithm 1: A unified framework of DSD

```
  input  : G = (V, E), ε, T
  output : An exact densest subgraph D(G)
1 f ← False; ρ_ ← k*/2; ρ¯ ← k*; w ← ∅ ;
2 repeat
      // (1) The graph reduction method.
3     G ← ReduceGraph(G, ρ_);
      // (2) The vertex weight update method.
4     w ← VWU(G, w, T, ρ_, ρ¯);
      // (3) The candidate subgraph extract and verify.
5     (f, D(G), ρ_, ρ¯) ← CSV(G, ρ_, ρ¯, w, ε);
6 until f=True;
7 return D(G);
```

Specifically, given a graph G, an approximate ratio ϵ, and the number of iterations T, we initialize the upper and lower bounds of ρ_G^* (line 1), where k^* is the maximum core number which will be introduced later. We then iteratively execute operations in the following three stages (lines 2-6):

(1) Locate the graph into a smaller subgraph (i.e., $\lceil \underline{\rho} \rceil$-core) utilizing the lower bound $\underline{\rho}$ (Section 4.1);
(2) Update the vertex weight vector $\mathbf{w}$ for each vertex over T iterations (Section 4.2);
(3) Extract the candidate subgraph using vertex weight vector $\mathbf{w}$, update upper and lower bounds of ρ_G^*, and verify if the candidate subgraph meets the requirements (Section 4.3). Terminate the process if it does; otherwise, update the parameters and repeat the above steps.

In Table 3, we illustrate the details of these three stages for each category of DSD algorithms for undirected graphs. For example, for `CoreExact`, it discovers UDS via binary search, for each search process: **Stage (1)** locate graph into a smaller k-core; **Stage (2)** compute the maximum flow on this smaller graph, and **Stage (3)** verify the result optimality via minimum cut. The vertex weight vector $\mathbf{w}$ represents the weights assigned to different vertices. It holds different meanings and serves different purposes in different algorithms:

- In the network flow-based algorithms, $\mathbf{w}$ denotes the flows from the vertices to the target node;
- In the CP-based algorithms, $\mathbf{w}$ represents the weight sum received by each vertex;
- In the peeling-based algorithms, $\mathbf{w}$ is used to select which vertex should be removed.

Hence, computing the maximum flow, optimizing the CP formulation, and peeling vertices are exactly the vertex weight update process. Our abstraction unifies the different uses of weights in different algorithms. Due to space limits, we present an overview of the DDS problem in our technical report [81].

Algorithm 2: `FWExact` under our unified framework

```
input  : G = (V, E), ε = 0, T
output : An exact densest subgraph D(G)
f ← False; ρ_ ← k*/2; ρ¯ ← k*;
repeat
    G ← ReduceGraph(G, ρ_);
    w ← FWExactVWU(G, w, T, ρ_, ρ¯) (Algorithm 4);
    (f, D(G), ρ_, ρ¯) ← FWExactCSV(G, ρ_, ρ¯, w, ε) (Algorithm 6);
until f=True;
return D(G);
```

Here, we provide an example to illustrate how to run `FWExact` using the three steps of our unified framework, as shown in Algorithm 2. The other CP-based algorithms are almost the same as

FWExact, due to the space limit, we only present FWExact here, and we also show an example about FlowExact in our technical report [81]. In the VWU stage of FWExact, it repeats T iterations to optimize the CP formulation of the UDS problem (see Algorithm 4). In the CSV stage of FWExact, the vertices with higher weights are more likely to appear in the densest subgraph $\mathcal{D}$, since they are linked by more edges (see Algorithm 6).

4 COMPARISON AND ANALYSIS OF DSD ALGORITHMS FOR UNDIRECTED GRAPHS

In this section, we systematically compare and analyze all the UDS algorithms in terms of the three stages of our unified framework.

4.1 Graph reduction

We first review the notion of *k-core*.

*Definition 4.1 (k-**core** [33, 40, 90]).* Given an undirected graph G and an integer k ($k \geq 0$), its k-core, denoted $\mathcal{H}_k$, is the largest subgraph of G, such that $\forall v \in \mathcal{H}_k$, $deg_{\mathcal{H}_k}(v) \geq k$.

The *core number* of a vertex $v \in V$ is the largest k for which a k-core contains v; the maximum core number among all vertices is denoted k^*. A k-core has an interesting "nested" property [10]: for any two non-negative integers i and j s.t. $i < j$, $\mathcal{H}_j \subseteq \mathcal{H}_i$.

CoreExact [33] locates the UDS into some k-cores. Specifically, the $\mathcal{D}(G)$ is contained in the $\lceil \rho^*_G \rceil$-core, however, since ρ^*_G is unknown in advance, we cannot use it directly.

Instead, we locate the UDS into some k-cores using a lower bound on ρ^*_G, thanks to the nested property of k-cores.

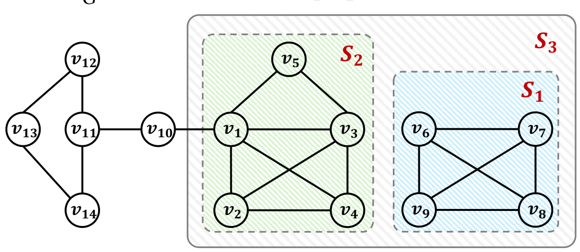


Figure 2: An example of the core-based graph reduction.

Example 1. *For example, for the undirected graph in Figure 2, suppose the lower bound of ρ^*_G is 3. Then, the UDS can be located in the 3-core, i.e., S_1 and S_2, which is smaller than the entire graph.*

Due to the nested property of k-core, vertices with smaller core numbers in the remaining graph can be successively removed in the search process thus gradually reducing the size of the graph, since as we shall see, the lower bound of ρ^*_G is progressively increasing. Besides, as shown in [33], the ρ^*_G cannot be larger than k^* and cannot be smaller than $k^*/2$.

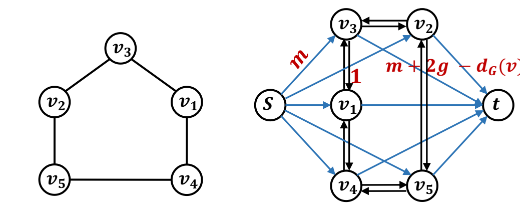
4.2 Vertex weight updating

We show that the key components in various UDS algorithms can be considered as the process of vertex weight updating.

We outline the VWU template in Algorithm 3, which begins by initializing the vertex weight vector $\mathbf{w}$ along with auxiliary variables facilitating the update process of $\mathbf{w}$. We update the variables until the stop condition is met, where in each iteration, we update the auxiliary variables and $\mathbf{w}$ (lines 2-3). Here, the different algorithms have various stop conditions in the while-loop, such as for FlowExact and CoreExact, the stop conditions are the maximum flow is reached. We summarize the stopping conditions of the while-loop for different DSD algorithms in our technical report [81].

Algorithm 3: The template of VWU

```
1  ❶ initialize w and auxiliary variables;
   // Algorithms have varied stop conditions.
2  while stop condition is not met do
3      ❷ update the auxiliary variables;
4      ❸ update w via auxiliary variables;
```



(a) An undirected graph G (b) A flow network $\mathcal{F}$

Figure 3: An undirected Graph G and its flow network $\mathcal{F}$.

• **Network flow-based algorithms.** For the exact algorithms, FlowExact and CoreExact need to ❶ build a flow network based on the guessed maximum density g (i.e., $g = (\underline{\rho} + \overline{\rho})/2$). For lack of space, we omit the detailed steps of building the flow network [40]. For example, Figure 3 shows a flow network of an undirected graph. We use auxiliary variables to denote the capacity and flow of each edge, and $\mathbf{w}(v)$ represents the value of flow from vertex v to the sink node t. ❷ After constructing the flow network, they try to update the flows of some edges in $\mathcal{F}$, and ❸ increase $\mathbf{w}$. The above process is repeated until the maximum flow is reached. CoreExact follows the same steps as FlowExact, but it utilizes k-core for graph reduction (refer Section 4.1). Unlike exact algorithms, FlowApp does not need to calculate the exact maximum flow, it only needs to perform partial maximum flow computations.

• **CP-based algorithms.** We first introduce the well-known CP formulation of the UDS problem [25] as follows:

$$
\begin{aligned}
\mathrm{CP}(G) \quad & \min \sum_{u \in V} \mathbf{w}^2(u) \\
\text{s.t.} \quad & \mathbf{w}(u) = \sum_{(u,v) \in E} \alpha_{u,v}, && \forall u \in V \\
& \alpha_{u,v} + \alpha_{v,u} = 1, && \forall (u,v) \in E \\
& \alpha_{u,v} \geq 0, \alpha_{v,u} \geq 0 && \forall (u,v) \in E
\end{aligned}
\tag{1}
$$

This CP(G) can be visualized as follows. Each edge $(u, v) \in E$ has a weight of 1, which it wants to assign to its endpoints: u and v such that the weight sum received by the vertices is as even as possible. Indeed, after a sufficient number of weight update iterations, we can derive an optimal solution to formulation 1, by inducing the subgraph with vertices of the highest weights based on $\mathbf{w}$, and the weight of any vertex in this subgraph equals the density of UDS. Following this intuition, $\alpha_{u,v}$ in CP(G) indicates the weight assigned to u from edge (u,v), and $\mathbf{w}(u)$ is the weight sum received by u from its adjacent edges.

All CP-based algorithms aim to solve CP(G) in Eq. (1). There are three widely used types of CP solvers for the DSD problem:

Frank-wolfe, MWU, and FISTA-based algorithms. All these algorithms, no matter whether they are designed for approximation or exact solutions, share the same VWU procedure.

Algorithm 4: VWU of Frank-wolfe and MWU

1 ❶ initialize $\mathbf{w}$ and auxiliary variables;
2 **foreach** $(u, v) \in E$ **do** $\alpha^{(0)}_{u,v} \leftarrow 1/2$; $\alpha^{(0)}_{v,u} \leftarrow 1/2$; $t \leftarrow 1$;
3 **foreach** $u \in V$ **do** $\mathbf{w}^{(0)}(u) \leftarrow \sum_{(u,v)\in E} \alpha^{(0)}_{u,v}$;
4 **while** $t \leq T$ **do**
5   ❷ update the auxiliary variables;
6   **foreach** $(u, v) \in E$ **do** update $\widehat{\alpha}_{u,v}$ and $\widehat{\alpha}_{v,u}$;
7     $\alpha^{(t)} \leftarrow (1 - \gamma_t) \cdot \alpha^{(t-1)} + \gamma_t \cdot \widehat{\alpha}$, with $\gamma_t = \frac{2}{t+2}$ or $\frac{1}{t+1}$;
8   ❸ update $\mathbf{w}$ via α;
9   **foreach** $v \in V$ **do** $\mathbf{w}^{(t)}(u) \leftarrow \sum_{(u,v)\in E} \alpha^{(t)}_{u,v}$;
10     $t \leftarrow t + 1$;

The Frank-wolfe and MWU-based algorithms solve $\mathsf{CP}(G)$ in an iterative manner, where $\gamma_t = \frac{2}{t+2}$ for Frank-Wolfe algorithms and $\gamma_t = \frac{1}{t+1}$ for MWU-based algorithms. In each iteration, algorithms linearize the objective function at the current point and move towards minimizing it [18, 25, 47]. Algorithm 4 shows the process, ❶ starting with initializing α and $\mathbf{w}$ (lines 2-4). Next, in the t-th iteration, ❷ each edge $(u, v) \in E$ attempts to distribute its weight, i.e., 1, to the endpoint with a smaller $\mathbf{w}^{(t-1)}$ value, and the $\alpha^{(t)}$ values of all vertices are computed as a convex combination by $\alpha^{(t-1)}$ and $\widehat{\alpha}$ (lines 5–7). Then, ❸ $\mathbf{w}^{(t)}$ is updated by the weight sum received by each vertex in V (line 9).

FISTA-based algorithms [42] also adopt the iterative paradigm, but leverage the *projection* operation [76] to speed up the optimization process. The projection operation in each iteration initially proceeds with a gradient descent step to move towards a local minimum. Subsequently, it adjusts the point to ensure it falls within the feasible region by projecting back if necessary. Harb et al. [42] further proved that they converge faster in theory, but experimentally there is a gap between the theoretical conclusion and practical results, i.e., they are slower than other CP-based algorithms as the *projection* is very time-consuming.

Algorithm 5: VWU of Greedy and Greedy++

1 ❶ initialize $\mathbf{w}$ and auxiliary variables;
2 **foreach** $v \in V$ **do** $\mathbf{w}^{(0)}(v) \leftarrow 0$, $H \leftarrow G$; $t \leftarrow 1$;
3 **while** $t \leq T$ **do**
4     $H \leftarrow G$;
5   **repeat**
6   select vertex v minimizing $\mathbf{w}^{(t-1)}(v) + d_H(v)$;
7   ❷ update the auxiliary variables;
8   **foreach** $u \in N(v, H)$ **do** $d_H(u) \leftarrow d_H(u) - 1$;
9   ❸ update $\mathbf{w}$ via d_H;
10         $\mathbf{w}^{(t)}(v) \leftarrow \mathbf{w}^{(t-1)}(v) + d_H(v)$;
11   remove v and all its adjacent edges (u, v) from H;
12   **until** $V(H) \neq \emptyset$;
13     $t \leftarrow t + 1$;

• **Peeling-based algorithms.** The key idea of peeling-based algorithms is to find a vertex with the minimum weight, remove it from the graph and update the weights of the remaining vertices. Algorithm 5 presents the details of Greedy and Greedy++. Specifically, when $T = 1$, this algorithm is Greedy, removing the vertex with the minimum degree one by one. Greedy++ algorithm is based on Greedy and iteratively removes the vertices via T iterations. In each iteration, it iteratively removes the vertex with the smallest weight, where the weight of vertex v in each iteration is the sum of its induced degree (w.r.t. the remaining vertices) and the weight of v in the previous iteration (line 6). CoreApp reveals that k^*-core can serve as a 2-approximation solution, where the k^*-core can be computed by following a peeling-based process.

4.3 Candidate subgraph extraction and verification

In this section, we explore the CSV stage across various UDS algorithms and examine how to utilize upper and lower bounds of the optimal density ρ^*_G for verifying results. Let g represent the guessed density, and set $g = (\underline{\rho} + \overline{\rho})/2$.

• **FlowExact and CoreExact.** FlowExact and CoreExact need to compute the minimum cut $(\mathcal{S}, \mathcal{T})$ via the maximum flow. If $\mathcal{S}$ contains only the source node $\{s\}$, $\overline{\rho}$ is updated to g; otherwise, $\underline{\rho}$ is set to g and $\mathcal{S} \setminus \{s\}$ is returned as the candidate subgraph.

To verify the quality of the candidate subgraph, FlowExact checks if the difference between $\overline{\rho}$ and $\underline{\rho}$ is less than $\frac{1}{n\cdot(n-1)}$, as the density difference between any two subgraphs must be larger than $\frac{1}{n\cdot(n-1)}$. When the condition is satisfied, the exact solution has been found. In contrast, CoreExact utilizes less stringent stop conditions for verifying results: it checks if the density difference is $< \frac{1}{|V_C|\cdot(|V_C|-1)}$, where V_C is the largest connected component in the graph.

• **FlowApp.** FlowApp searches for an augmenting path in $\mathcal{F}$ after h blocking flows. If such a path exists, $\underline{\rho}$ is updated to g and the residual graph of $\mathcal{F}$ is returned as the candidate subgraph; otherwise, $\overline{\rho}$ is updated to g and no candidate subgraph is obtained.

Algorithm 6: CSV of CP-based algorithms

1 **foreach** $1 \leq i \leq |V(G)|$ **do**
2     $u_i \leftarrow$ the vertex with the i-th highest weight in $V(G)$;
3     $G_i \leftarrow$ the induced subgraph of top-i weight vertices;
4     $y_i \leftarrow d_{G_i}(u_i)$;
5 $s^* \leftarrow \arg\max_{1\leq s\leq n} \frac{1}{s} \sum_{i=1}^{s} y_i$;
6 $\mathcal{S} \leftarrow$ the subgraph induced by the first s^* vertices;
7 $\underline{\rho} = \max(\rho(\mathcal{S}), \rho)$; $\overline{\rho} = \max_{1\leq i\leq n} \min\left\{\frac{1}{i}\binom{i}{2}, \frac{1}{i}\sum_{j=1}^{i} \mathbf{w}(u_j)\right\}$;
8 $f \leftarrow$ verify via maximum flow;
9 **return** $f, \mathcal{S}, \underline{\rho}, \overline{\rho}$;

FlowApp verifies whether a candidate subgraph is a $(1 + \epsilon)$-approximation solution by checking if $\frac{g-\underline{\rho}}{2g} < \frac{\epsilon}{3-2\epsilon}$ [95].

• **CP-based algorithms.** All CP-based algorithms use the vertex weight vector $\mathbf{w}$ and the PAVA algorithm [25] to extract the candidate subgraph.

Algorithm 6 presents the details, where the liens (1-6) is the process of using PAVA to extract the candidate subgraph. Intuitively, the vertices with higher weights are more likely to appear in the densest subgraph $\mathcal{D}(G)$, since they are linked by more edges. Thus, the subgraph induced by the first s^* vertices with the largest weights is returned as the candidate subgraph. Here, $\underline{\rho}$ is updated to $\rho(\mathcal{S})$, and

$\overline{\rho}$ is updated to $\max_{1\le i\le n} \min\left\{\frac{1}{i}\binom{i}{2}, \frac{1}{i}\sum_{j=1}^{i}\mathbf{w}(u_j)\right\}$ [25, 88]. After a sufficient number of iterations, CP-based algorithms converge to the optimal solution, which can be verified using maximum flow [25]. For the approximation algorithms, we can use the ratio of $\overline{\rho}$ over $\rho(\mathcal{S})$ to estimate the empirical approximation ratio, as the optimal density is not known in advance [25, 88].

• **Peeling-based algorithms.** `Greedy` and `Greedy++` extract the highest density subgraph during the process of vertex peeling. In addition, `CoreApp` utilizes the k^*-core as their returned candidate subgraph. These algorithms do not require updating $\underline{\rho}$ and $\overline{\rho}$ as they do not need to verify results.

4.4 Optimizations

There are two optimization strategies used in existing UDS works.

• **Locating the UDS in a connected component.** `CoreExact` locates the UDS in a connected $\lceil\rho\rceil$-core; notice that the $\lceil\rho\rceil$-core may be disconnected. Connected cores tend to be smaller than the disconnected cores they are part of. For example, in Figure 2, the two connected cores S_1 and S_2 can be processed one by one by `CoreExact` algorithm.

• **Simultaneous update strategy.** Danisch [25] employs a simultaneous weight update strategy for `FWExact` and `FWApp`. Specifically, within each iteration, if a vertex v's weight $\mathbf{w}(v)$ changes, then the updated vertex weight is promptly visible to subsequent updates of other vertices in the same iteration. The simultaneous weight update strategy enables a more balanced weight distribution among vertices, making the algorithm converge faster, as all the vertices in the UDS have the same weight upon convergence.

5 COMPARISON AND ANALYSIS OF DSD ALGORITHMS FOR DIRECTED GRAPHS

The DDS problem is more complicated than the UDS problem because it is an induced subgraph of two vertex sets S and T, which leads to a search space of n^2 possible values of the ratio between the size of the two vertex sets (i.e., $c = |S|/|T|$) which must be examined when computing the maximum density. Next, we show how to adapt our framework (Algorithm 8) for the DDS problem.

• **Exact algorithms.** The exact algorithms need to enumerate all possible values of c and compute the DS for each c. For each fixed c, the exact algorithms share the same paradigm with UDS algorithms, indicating that they can be easily incorporated into our framework. Similar to `CoreExact`, `DC-Exact` [68] reduces the size of the graph by locating the DDS into some $[x, y]$-core [68].

Definition 5.1 ($[x, y]$-core [68]). Given a directed graph D=(V, E), the $[x, y]$-**core** is the largest (S, T)-induced subgraph $D[S, T]$, which satisfies:

1. $\forall u \in S, d^+_{D[S,T]}(u) \ge x$ and $\forall v \in T, d^-_{D[S,T]}(v) \ge y$;
2. $\nexists D[S', T'] \ne D[S, T]$, such that $D[S, T]$ is a subgraph of $D[S', T']$, i.e., $S \subseteq S'$, $T \subseteq T'$, and $D[S', T']$ satisfies (1);

The $[x^*, y^*]$-core is the $[x, y]$-core with the largest $x \cdot y$ values. Note that $[x, y]$-core is an extension of the k-core, as each vertex in the $[x, y]$-core has at least x out-neighbors and y in-neighbors.

By Theorem 5.6 in [68], we only need to discover DDS in the $\left[\frac{\underline{\rho}}{2\sqrt{c_r}}, \frac{\sqrt{c_l}\underline{\rho}}{2}\right]$-core, where (c_l, c_r) specifies the interval of c values under consideration during a particular stage of the divide-and-conquer approach and $\underline{\rho}$ is the lower bound of the optimal density.

• **Approximation algorithms**. There are two groups of approximation algorithms, i.e., peeling-based [8, 17, 53, 62, 68] and CP-based [66] algorithms. `DGreedy` needs to enumerate all c values to obtain the approximation solution, and when a specific c is fixed, it is analogous to `Greedy`. `XYCoreApp` focuses on identifying the $[x^*, y^*]$-core. Each time it fixes one dimension and optimizes the other dimension via iteratively peeling vertices in the other dimension to find the $[x^*, y^*]$-core, where the peeling process is similar to `Greedy`. `WCoreApp` sequentially eliminates vertices based on the lowest weight, where a vertex's weight is determined by the product of its out-degree and in-degree. The key difference between `WCoreApp` and `Greedy` is in how they calculate a vertex's weight. The CP-based algorithms need to update $\mathbf{w}_\alpha$ and $\mathbf{w}_\beta$ via two auxiliary variables α and β, whose process is similar to Algorithm 4.

Moreover, `DCExact` [68] first introduced a divide and conquer method to reduce the number of $\frac{|S|}{|T|}$ values examined from n^2 to k, where theoretically $k \le n^2$, but practically $k \ll n^2$. `DFWExact` [66] introduced a new divide-and-conquer method, utilizing the relationship between the DDS and the c-biased DS to skip searches for certain c values in the DSD process.

In addition, Sawlani and Wang [85] showed that the DDS problem can be transformed into $O(\log_{1+\epsilon} n)$ vertex-weighted UDS problems, where the vertex weights are assigned according to $O(\log_{1+\epsilon} n)$ different guesses of $\frac{|S|}{|T|}$. In our study, we test its performance by adapting the SOTA algorithm for the UDS problem.

6 EXPERIMENTS

We now present the experimental results. Section 6.1 discusses the setup. The experimental results of the UDS algorithms and DDS algorithms are reported in Sections 6.2 and 6.3, respectively.

6.1 Setup

Table 4: Undirected graphs used in our experiments.

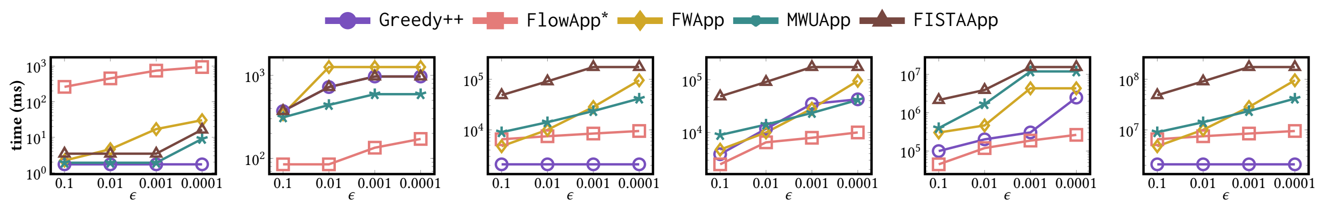
Dataset	Category	$\|V\|$	$\|E\|$	ρ
bio-SC-GT (BG)	Biological	1,716	31,564	18.4
econ-beacxc (EB)	Economic	507	42,176	83.2
DBLP (DP)	Collaboration	317,080	1,049,866	3.3
Youtube (YT)	Multimedia	3,223,589	9,375,374	2.9
LiveJournal (LJ)	Social	4,036,538	34,681,189	8.6
UK-2002 (UK)	Road	18,483,186	261,787,258	14.2
WebBase (WB)	Web	118,142,155	881,868,060	7.5
Friendster (FS)	Social	124,836,180	1,806,067,135	14.5

Table 5: Directed graphs used in our experiments.

Dataset	Category	$\|V\|$	$\|E\|$	ρ
maayan-lake (ML)	Foodweb	183	2,494	13.6
maayan-figeys (MF)	Metabolic	2,239	6,452	2.9
Openflights (OF)	Infrastructure	2,939	30,501	10.4
Advogato (AD)	Social	6,541	51,127	7.8
Amazon (AM)	E-commerce	403,394	3,387,388	8.4
Baidu-zhishi (BA)	Hyperlink	2,141,300	17,794,839	8.3
Wiki-en (WE)	Hyperlink	13,593,032	437,217,424	32.2
SK-2005 (SK)	Web	50,636,154	1,949,412,601	38.5

Table 6: Comparison of 2-approx. UDS algorithms (Red denotes the most efficient algorithm, and Blue denotes the best accuracy).

Algorithm	EB		DP		YT		LJ		WB		FS	
	time	ratio	time	ratio	time	ratio	time	ratio	time	ratio	time	ratio
FlowApp*	43.8 ms	1.556	126.3 ms	1.001	9.1 s	1.761	2.4 s	1.075	41.0 s	1.085	2095.6s	1.799
CoreApp	1.6 ms	1.182	98.0 ms	1.001	0.7 s	1.139	2.0 s	1.033	41.1 s	1.085	156.0 s	1.065
PKMC	4.3 ms	1.182	104.5 ms	1.001	11.2 s	1.139	8.5 s	1.033	37.8 s	1.085	5546.4 s	1.065
FWApp	11.8 ms	1.072	371.6 ms	1.111	9.1 s	1.004	16.4 s	1.078	383.6 s	1.214	1468.2 s	1.200
MWUApp	2.7 ms	1.001	213.6 ms	1.420	24.8 s	1.001	9.8 s	1.115	281.6 s	1.197	723.7 s	1.029
FISTAApp	3.4 ms	1.000	390.3 ms	1.152	49.1 s	1.016	28.7 s	1.480	2377.0 s	1.105	2053.8 s	1.026
Greedy	0.8 ms	1.000	72.8 ms	1.001	0.6 s	1.000	1.8 s	1.013	37.3 s	1.016	130.6 s	1.000



(a) EB (b) DP (c) YT (d) LJ (e) WB (f) FS

Figure 4: Efficiency results of $(1+\epsilon)$-approximation algorithms on undirected graph.

We use sixteen real datasets from different domains including 8 undirected graphs and 8 directed graphs, which are available on the Stanford Network Analysis Platform [79], Laboratory of Web Algorithmics [78], Network Repository [82], and Konect [54]. Tables 4 and 5 report the statistics of these graphs, where ρ denotes the density (i.e., $|E|/|V|$) of each graph. Due to space limitations, we present results on twelve datasets here and include additional datasets and results (e.g., the sizes of UDS and DDS obtained by various DSD algorithms) in our technical report [81]. We implement all the algorithms in C++ and run experiments on a machine having an Intel(R) Xeon(R) Gold 6338R 2.0GHz CPU and 512GB of memory, with Ubuntu installed. If an exact algorithm cannot finish in three days or an approximation algorithm cannot finish in one day, we mark its running time as **INF** in the figures and "—" in the tables.

6.2 Evaluation of UDS algorithms

1. Overall performance. We first report the running time of all 2-approximation algorithms and the actual approximation ratios of approximation solutions returned by each algorithm in Table 6. Here, we set ϵ=1 for $(1+\epsilon)$-approximation algorithms. We observe that Greedy always achieves the best performance when ϵ=1 in terms of accuracy and efficiency.

We then examine the efficiency of $(1+\epsilon)$-approximation algorithms by varying ϵ from 0.1 to 0.0001, and report their running time in Figure 4. Specifically, we report the running time needed by different $(1+\epsilon)$-approximation algorithms to find an approximation solution whose density is at least $(1+\epsilon)\times\rho^*_G$. For a fair comparison, we stop algorithms when they achieve a density of $(1+\epsilon)\times\rho^*_G$, rather than the theoretical number of iterations, since it often takes far fewer iterations in practice than the theoretical value. We observe that FlowApp* and Greedy++ always perform faster than the others, since FlowApp* utilizes the k-core-based graph reduction to reduce the search space; Greedy++ uses fewer iterations to obtain higher accuracy, and for each iteration, Greedy++ requires just a straightforward operation: iteratively remove the vertex with the lowest weight. As for CP-based algorithms, FISTAApp often takes more time to obtain solutions with the same accuracy as FWApp and MWUApp, despite theoretically needing fewer iterations. The is because it requires an extra projection operation in each iteration, which is very time-consuming, especially for large graphs. Besides, FWApp and MWUApp achieve the comparable performance. Finally,

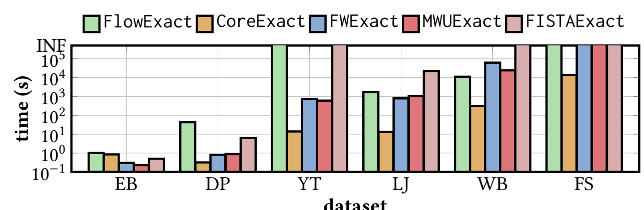


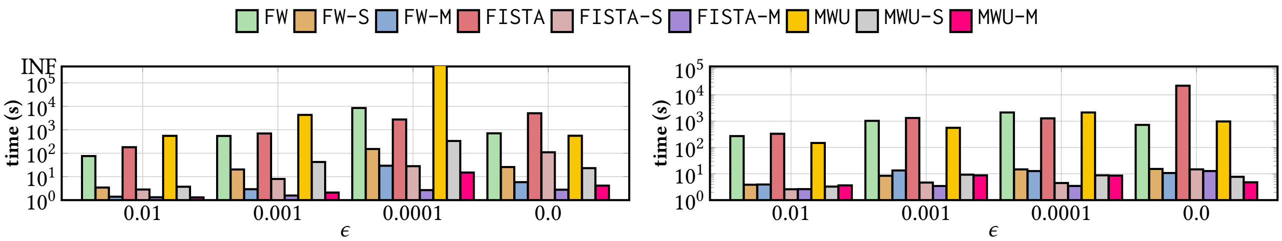
Figure 5: Efficiency results of exact algorithms for UDS.

we present the running time of all exact algorithms in Figure 5. To be specific, CoreExact performs best for the most part, since it can compute the minimum cut on the smaller flow network, and these CP-based algorithms achieve comparable performance.

Table 7: Speedup of graph reduction.

	Type	YT				LJ			
		0.01	0.001	0.0001	0	0.01	0.001	0.0001	0
FW	Single	22	27	56	27	70	122	145	47
	Multi	54	186	289	127	69	77	168	67
MWU	Single	148	102	>779	24	45	60	244	127
	Multi	420	2042	>17257	134	45	64	250	205
FISTA	Single	64	90	98	46	128	284	287	1505
	Multi	136	442	1027	1816	127	384	370	1747

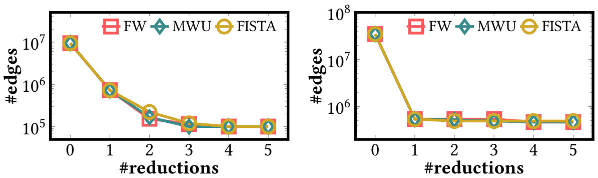
2. Evaluation of graph reduction. We evaluate the effectiveness of two graph reduction strategies on all algorithms, except 2-approximation algorithms: 1) *Single-round reduction*, which involves reducing the entire graph to a $k^*/2$-core once and then executing all algorithms on this smaller graph, instead of the original large graph; and 2) *Multi-round reduction*, which is applied whenever a tighter lower bound of ρ_{G^*} is identified, as it locates the UDS within a smaller subgraph with a higher core number, resulting in a smaller graph. We present the efficiency of all algorithms and their variants in Figure 6 on two datasets, where the original algorithm names appended with "S" and "M" indicate the



(a) YT

(b) LJ

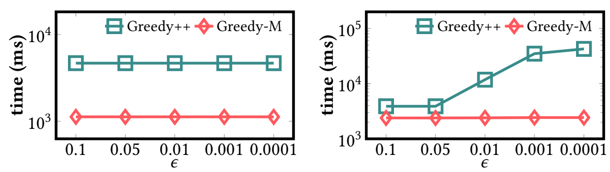
Figure 6: The effect of graph reduction on undirected graphs.



(a) YT

(b) LJ

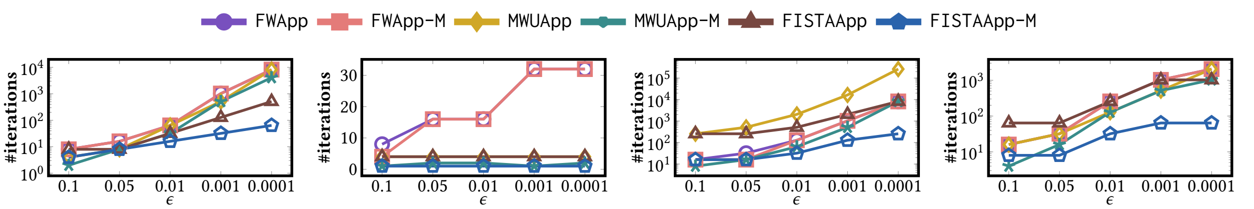
Figure 7: The number of edges in graphs.



(a) YT

(b) LJ

Figure 8: The effect of graph reduction for `Greedy++`.



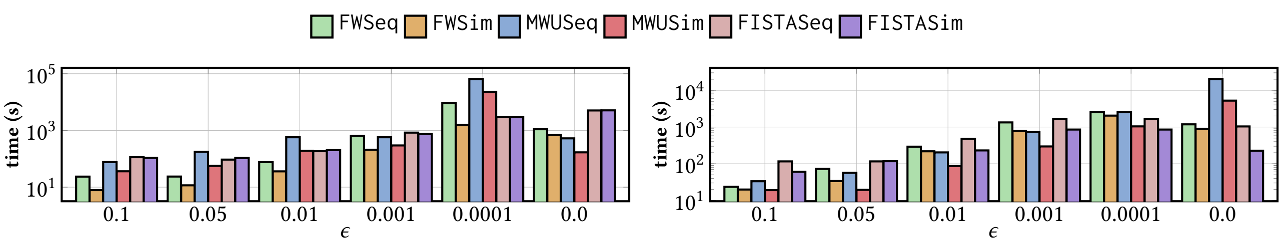
(a) EB

(b) DP

(c) YT

(d) LJ

Figure 9: The number of iterations w.r.t ϵ on undirected graphs.



(a) YT

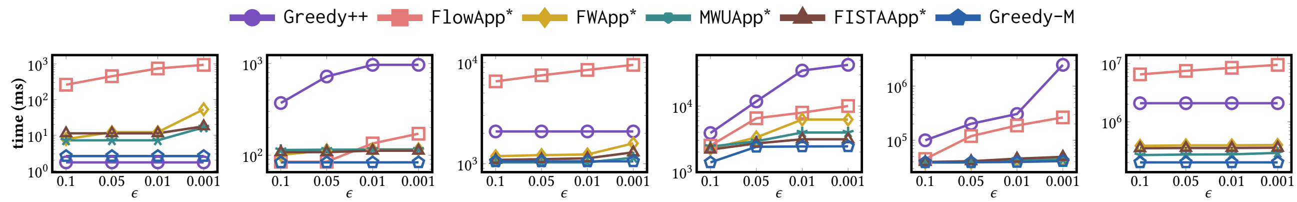
(b) LJ

Figure 10: The effect of update strategies on undirected graphs.

adoption of a single-round and multi-round reduction, respectively. We also present the number of remaining edges after each of the first five rounds of graph reductions for all CP-based exact algorithms, on two datasets (Figure 7), wherein the x-axis, "0" denotes the original graph before any reduction, and we record the speedup ratio of graph reduction over different algorithms in Table 7.

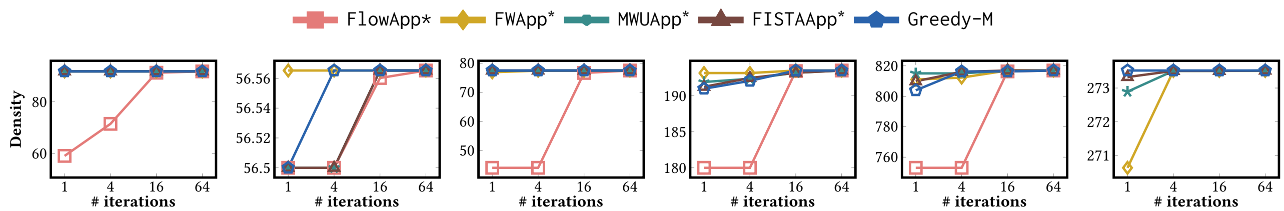
We make the following analysis: 1) Graph reduction significantly enhances the efficiency of all algorithms. 2) Using multi-round reduction can result in a much greater performance increase than just using single-round reduction in most cases, yet in some cases, the multi-round reduction might bring extra time consumption due to the time needed to perform the reduction itself. For example, applying multi-round reduction can speed up `FISTA` 1097 times, more than 98 × speedup achieved with a single-round reduction on the YT dataset with $\epsilon = 0.0001$. 3) `FISTA`, employing multi-round graph reduction outperforms MWU and FW-based algorithms in terms of efficiency while using similar reduction strategies, with the same accuracy. This efficiency boost stems from less projection time on smaller graphs and a reduction in the number of iterations needed. 4) The first graph reduction drastically reduces the graph size, pruning over 98% edges after five rounds of reductions. 5) As shown in Figure 8, graph reduction significantly enhances the performance of `Greedy++`, achieving an improvement of up to one order of magnitude faster than `Greedy++` without graph reduction.

3. Accuracy of CP-based algorithms. In this experiment, we report the number of iterations each algorithm needed to achieve different levels of accuracy in Figure 9. We make the following observations: 1) The actual numbers of iterations needed for all CP-based algorithms are much less than their theoretical numbers. For example, on the YT dataset with $\epsilon = 0.1$, `FWApp`, `MWUApp`, and `FISTAApp` theoretically require at least 10^{16}, 10^{16}, and 10^{8} iterations on the original graph to obtain the approximated solution, while in practice they only need 16, 256, and 256 iterations, respectively; similarly, for the reduced graph, `FWAPP-M`, `MWUApp-M`, and `FISTAApp-M`



(a) EB (b) DP (c) YT (d) LJ (e) WB (f) FS

Figure 11: Efficiency results of new $(1+\epsilon)$-approximation algorithms on undirected graphs.



(a) EB (b) DP (c) YT (d) LJ (e) WB (f) FS

Figure 12: The densities of UDS obtained by the $(1+\epsilon)$-approximation algorithms.

theoretically require 10^{14}, 10^{14}, and 10^{6} iterations, but practically they also only need 16, 8, and 16 iterations. 2) Graph reduction techniques significantly decrease the number of iterations required by CP-based algorithms, and this phenomenon is especially marked in FISTA-based algorithms because it uses Nesterov-like momentum [42] in its projection phase and adopts $\frac{1}{2\cdot\Delta(G)}$ as the learning rate, since graph reduction can reduce $\Delta(G)$.

4. Evaluation of weight update strategies. In this experiment, we test the effect of the vertex weight update strategies, i.e., sequential and simultaneous update strategies, in all CP-based algorithms. As shown in Figure 10, we observe that the algorithms using the simultaneous update strategy almost always take less time to achieve the same approximation ratios than the algorithms using the sequential weight update strategy. This is because the simultaneous update strategy makes a more balanced weight distribution.

Based on the above results and discussions, we obtain two major conclusions: 1) Graph reduction is highly beneficial for all algorithms, and employing multi-round reduction often yields better outcomes than single-round reduction. 2) For CP-based algorithms, the simultaneous update strategy typically outperforms the sequential one. By combining existing techniques, we introduce new algorithms which we present and evaluate in the next section.

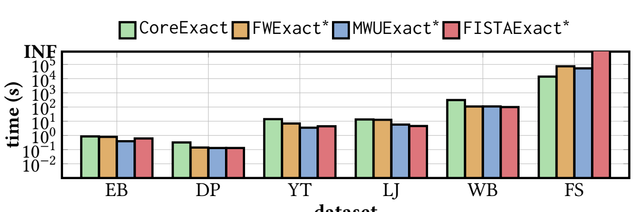


Figure 13: Efficiency of exact algorithms for UDS.

5. New algorithms. Specifically, we apply a simultaneous update strategy and multi-round graph reductions for all CP-based algorithms. We denote modified versions of original algorithms with an asterisk (*). For Greedy++, we enhance efficiency through graph reduction, and denote this variant by Greedy-M. We first evaluate the efficiency of various $(1+\epsilon)$-approximation algorithms with the optimal density used in the early stop check. Figure 11 indicates that for smaller datasets, graph reduction is not very useful. However, for large graphs, Greedy-M is always the best one, and those new CP-based algorithms achieve comparable performance. This is because that Greedy-M is a much simpler algorithm than the CP-based ones. Besides, we report how the density of $\mathcal{D}(G)$ changes as the number of iterations increases for the UDS algorithms, as shown in Figure 12. We can make the following observations: (1) All the iteration-based algorithms converge to solutions with densities very close to the optimal in relatively few iterations, and the number of iterations required is significantly lower than the theoretical estimates. (2) The CP-based algorithms and Greedy-M achieve higher densities with fewer iterations compared to FlowApp*. However, as the number of iterations increases, the solution densities obtained by these algorithms become nearly identical.

We test all CP-based and FlowApp* algorithms by reporting their running time on the four largest datasets. In real situations, the optimal density is not known in advance, so we terminate an algorithm when it meets the early stop conditions that are set according to the specified ϵ, and report the running time in our technical report [81]. Our findings lead to two main insights: firstly, all CP-based algorithms perform similarly well. Secondly, CP-based algorithms are up to one order of magnitude faster than FlowApp*.

In addition, we compare the efficiency of the newly designed CP-based exact algorithms and CoreExact. As shown in Figure 13, the new algorithms generally require less time than CoreExact except on the FS dataset. After we deeply delve into the FS dataset, we discover a subgraph with a density of 273.518, nearly matching the optimal density of 273.519. We conjecture that if some subgraphs have densities very close to the optimal density, the exact CP-based algorithms often require a larger number of iterations to obtain the optimal solution. To verify it, we synthesize four datasets as follows: 1) we generate two cliques, each with 1,000 vertices, and connect these two cliques by a single edge; 2) we then randomly remove 0.001%, 0.01%, 0.1%, and 1% of the edges from the second clique to simulate if there exists a subgraph with a density close to that of the densest subgraph (i.e., the first clique).

Table 8 reports the number of iterations required by these algorithms on these datasets. We observe that as the density of the

Table 8: The number of iterations.

Method	1%	0.1%	0.01 %	0.001%
`FW`	128	2,048	32,768	131,072
`MWU`	128	1,024	16,384	65,536
`FISTA`	1	128	512	1,024

second graph approaches that of a clique, CP-based exact algorithms require a significantly higher number of iterations.

Figure 14: Memory usage on undirected graphs.

6. Memory usage. We report the memory usage of all exact UDS algorithms in Figure 14. We observe that the memory costs of all algorithms are almost at the same scale because all algorithms incur linear memory usage w.r.t. the graph size (i.e., $O(m)$). For approximation algorithms, their theoretical space costs are also the same, so we omit the evaluation.

6.3 Evaluation of DDS algorithms

1. Overall performance. Similar to the UDS part, we first evaluate the overall performance of various DDS algorithms. To be specific, (1) we compare the efficiency and accuracy of all approximation algorithms in Table 9. Here, we also set $\epsilon = 1$ for `DFWApp` and `VWApp`; (2) we test the performance of `DFWApp` and `VWApp` over different ϵ values, from 0.1 to 0.0001 in Figure 15; and (3) we record the running time of `DFlowExact`, `DCExact` and `DFWExact` in Figure 16. Based on the above results, we make the following observations: (1) `WCoreApp` is the most efficient one for larger graphs, but `DFWApp` usually yields a more accurate solution. (2) Although we have incorporated the SOTA UDS algorithm into `VWApp`, it is still slower than `DFWApp`. The main reason is that, although `VWApp` reduces the number of different trials of c from $O(n^2)$ to $O(\log_{1+\epsilon} n)$, this number is still too many compared to that of `DFWApp`. (3) For exact algorithms, `DFWExact` always outperforms the rest, because it significantly reduces the time cost incurred for computing maximum flow, and it can compute the minimum cut on the smaller flow network.

2. Effect of the lower bound of binary search. Recall that for `DFlowExact`, it utilizes binary search to find the maximum density for each c. Intuitively, a tighter binary search space can speed up the algorithm, which suggests we should use $\underline{\rho}$ as the search lower bound. However, as shown in our experiment, we find that a higher lower bound may affect the effectiveness of the divide-and-conquer strategy. `DCExact` sets the lower bound of the binary search as 0 for each c, instead of directly using $\underline{\rho}$. To investigate the effect of the lower bound, we introduce a hyper-parameter γ, which controls the lower bound of binary search by setting $\underline{\rho} = \gamma \cdot \underline{\rho}$. We evaluate `DCExact` by varying γ from `0` to `1` on four datasets, and report their running time and the actual visited c values in Figure 18. We can see that a higher lower bound may increase the number of c values examined and take more time. Although we cannot predict the best γ in advance, smaller values of γ usually perform better. We conjecture that this is because smaller γ leads to more edges remaining, more feasible values of c for the reduced graph, and then a larger interval of c to be pruned. Hence, we follow [68] to set $\underline{\rho} = 0$ for each c to keep the effectiveness of the divide and conquer strategy and improve the overall performance.

3. Evaluation of graph reduction. `DFWApp` [66] already adopts multi-reduction to improve its efficiency. To evaluate its usefulness, we conduct an ablation study by presenting the running time, the average numbers of vertices, and edges before and after the graph reduction for different values of ϵ, across all datasets. The $[x, y]$-core graph reduction method can substantially reduce the size of the original graph and speed up the overall efficiency. For lack of space, we present the detailed results in our technical report [81]

Given a lower bound $\underline{\rho}$ for the optimal density, and the required interval of c values $(c_l, \overline{c_r})$ to be examined, `DFWExact` and `DFWApp` do not use $\left[\frac{\rho}{2\sqrt{c_r}}, \frac{\sqrt{c_l}\rho}{2}\right]$-core to reduce the search space. The main reason is that for a wide interval of c, (i.e., smaller c_l, and larger c_r), only a small fraction of vertices and edges will be removed due to the small values of x and y. To further harness the power of graph reduction, the CP-based algorithms adjust the interval length of c values to enhance the effectiveness of $[x, y]$-core reduction. Specifically, a larger c_l and a smaller c_r lead to higher x and y values, which significantly reduce the size of the graph. To test the effectiveness of this adjustment strategy, we evaluate the CP-based algorithms by employing and not employing this strategy, which are denoted by `DFW` and `DFWReN`, across different ϵ values on the two largest datasets in Table 10. Specifically, we record the number of c values to be checked (marked as "#c"), the average number of edges remaining after applying the $[x, y]$-core reduction (marked as "#edges"), the average number of iterations processed by the `Frank-Wolfe` algorithm (marked as "#iterations"), the product of these three items (marked as "product"), and the running time. We can observe that: (1) the running time is generally proportional to the value of the "product". (2) While this strategy may increase the number of c values that need to be examined, it significantly reduces the scale of the graph and the value of the "product", indicating a better performance. We find that this adjustment strategy is not useful for `DCExact`, since for each c, it sets $\underline{\rho} = 0$ which means that it needs to calculate the maximum flow on a bigger graph.

4. Evaluation of weight update strategies. We also compare the simultaneous and sequential update strategies on directed graphs in Figure 17. Unlike the UDS problem, we discover that the former strategy is not always more effective than the latter one. The main reason is that these two strategies have different effects on the divide-and-conquer strategy, with the performance gap largely due to the different number of c values that need to be examined.

5. Memory usage. In this experiment, we evaluate the memory usage of all exact algorithms. As shown in Figure 19, we can observe that `DCExact` uses less memory than `DFWExact` due to the effectiveness of its graph reduction. For those approximation algorithms, they are around the same scale (i.e., $O(m)$), so we omit the details.

7 LESSONS AND OPPORTUNITIES

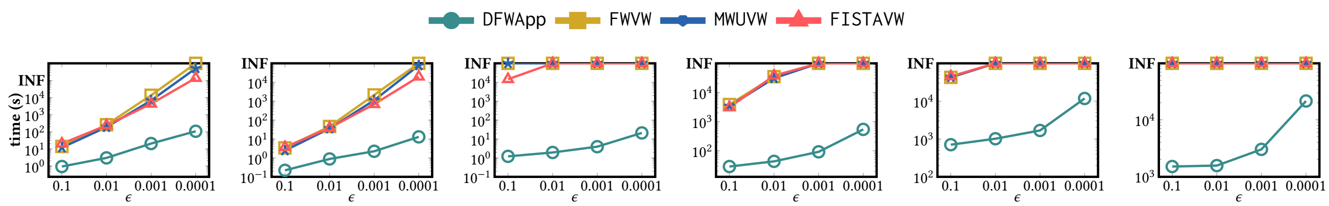
We summarize the lessons (L) for practitioners and propose practical research opportunities (O) based on our observations.

Lessons:

L1. In Figure 20, we depict a roadmap of the recommended DSD algorithms, highlighting which algorithms are best suited for different scenarios.

Table 9: Comparison of 2-approx. DDS algorithms (Red denotes the most efficient algorithm, and Blue denotes the best accuracy).

Algorithm	OF time	OF ratio	AD time	AD ratio	AM time	AM ratio	BA time	BA ratio	WE time	WE ratio	SK time	SK ratio
DGreedy	—	—	—	—	—	—	—	—	—	—	—	—
DFWApp	87.8 ms	1.001	175.3 ms	1.009	822.2 ms	1.004	26.8 s	1	645.9 s	1	1473.6 s	1.000
FWVW	139.0 ms	1.135	391.1 ms	1.122	6623.3 s	1.004	557.1 s	1.226	7855.1 s	1	19585.2 s	1.010
MWUVW	204.7 ms	1.024	600.0 ms	1.122	—	—	480.0 s	1.226	7739.7 s	1	18417.0 s	1.000
FISTAVW	251.7 ms	1.135	480.1 ms	1.122	846.6 s	1.004	408.1 s	1.000	7349.1 s	1	22076.3 s	1.000
XYCoreApp	27.9 ms	1.422	17.2 ms	1.130	751.7 ms	1.004	4.6 s	1.000	131.0 s	1.007	1423.9 s	1.000
WCoreApp	333.7 ms	1.396	6.0 ms	1.130	253.9 ms	1.004	1.4 s	1.000	12.6 s	1.007	100.3 s	1.000



(a) OF (b) AD (c) AM (d) BA (e) WE (f) SK

Figure 15: Efficiency results of $(1+\epsilon)$-approximation algorithms on directed graphs.

Dataset		$\epsilon = 0.01$ DFW	$\epsilon = 0.01$ DFWReN	$\epsilon = 0.001$ DFW	$\epsilon = 0.001$ DFWReN	$\epsilon = 0.0001$ DFW	$\epsilon = 0.0001$ DFWReN	$\epsilon = 0$ DFW	$\epsilon = 0$ DFWReN
WE	#Ratios	25	21	27	32	54	35	20	19
	#Edges	9,458,420	41,014,857	9,597,565	30,606,451	9,221,802	28,859,994	10,767,492	44,912,585
	#Iterations	174	152	486	603	2,156	2,580	100	105
	Product	3.78×10^{10}	1.31×10^{11}	1.35×10^{11}	5.91×10^{11}	6.36×10^{11}	2.61×10^{12}	2.15×10^{10}	8.98×10^{10}
	Time (s)	1,033.3	3,723.7	2,266.8	13,658.4	7,869.2	23,916.9	643.0	2,865.3
SK	#Ratios	17	15	24	18	28	—	21	15
	#Edges	44,552,779	375,979,205	44,837,242	313,662,857	44,362,182	—	45,844,129	365,743,496
	#Iterations	124	120	258	344	1,914	—	124	147
	Product	9.36×10^{10}	6.77×10^{11}	2.78×10^{11}	1.94×10^{12}	2.38×10^{12}	—	1.19×10^{11}	8.05×10^{11}
	Time (s)	1,419.7	15,856.6	3,425.8	48,106.6	24,922.1	—	2,425.7	15,909.8

Table 10: Effectiveness of graph reduction technique on directed graphs.

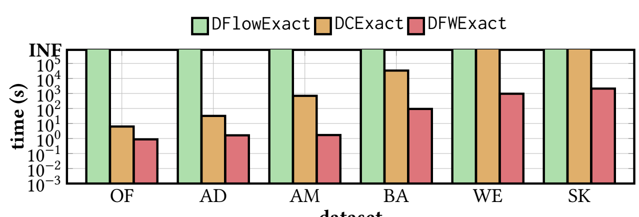
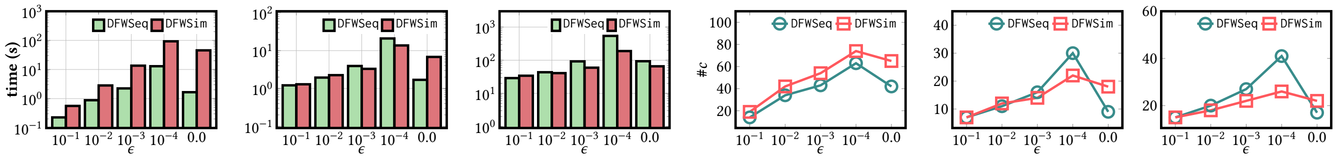


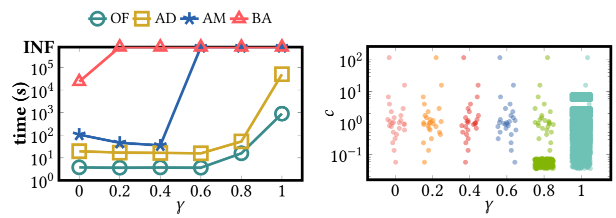
Figure 16: Efficiency of exact algorithms on DDS.

L2. Graph reduction is very useful for all algorithms and using graph reduction multiple times is better than using it only once.
L3. Furthermore, no single CP-based UDS algorithm dominates other algorithms in all tests, as their performance highly depends on the structure of the graph.
L4. For all iteration-based algorithms, the number of iterations required in practice is much less than the theoretical estimate.
L5. It is not a good idea to transfer the DDS problem to the UDS problem, since it cannot utilize the divide and conquer strategy to reduce the number of c values that need to be examined.
L6. For the DDS problem, it is essential to reduce the number of c values examined. Besides, when attempting to speed up the efficiency for a specific c, it is important to evaluate if it affects the number of c values that need to be examined subsequently.
Opportunities:
O1. For all UDS algorithms, simultaneous weight updates tend to result in better efficiency than sequential weight updates. Providing a solid theoretical explanation of this phenomenon is an important open problem.
O2. All existing DSD algorithms (both UDS and DDS) assume the setting of a single machine. What if the graph is too large to fit on a single machine? For example, the Facebook social network contains 1.32 billion nodes and 140 billion edges (http://newsroom.fb.com/companyinfo). Can we design I/O-efficient, distributed, or sublinear time algorithms for the DSD problem? Given the advantages of GPUs in executing multiple tasks concurrently due to their high-bandwidth parallel processing capabilities, designing GPU-friendly DSD algorithms [67] is another interesting opportunity.
O3. In many domains, the network is private, and returning the DSD can reveal information about the network. Some existing UDS algorithms [26, 28] have considered the framework of local differential privacy. Designing a similar extension for DDS is important.
O4. Heterogeneous graphs are prevalent in various domains such as knowledge graphs, bibliographic networks, and biological networks. A promising future research direction is to study DSD for heterogeneous graphs. Besides, studying DSD for attributed graphs is also an interesting future research problem. More lessons and opportunities are reported in our technical report [81].
O5. None of the well-known graph systems, such as Neo4j [75], Nebula [74], and TigerGraph [89], has incorporated DSD algorithms



(a) AD (time) (b) AM (time) (c) BA (time) (d) AD (#c) (e) AM (#c) (f) BA (#c)

Figure 17: The effect of update strategies on directed graphs.



(a) Running time (b) Distribution of c on OF

Figure 18: The effect of setting $\underline{\rho}$ for `DCExact`.

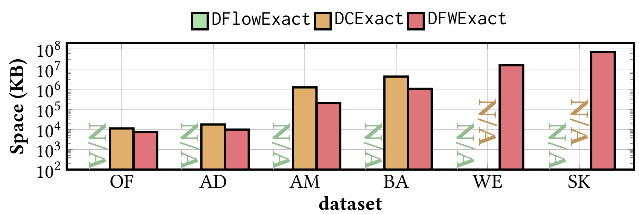


Figure 19: Memory usage on directed graphs.

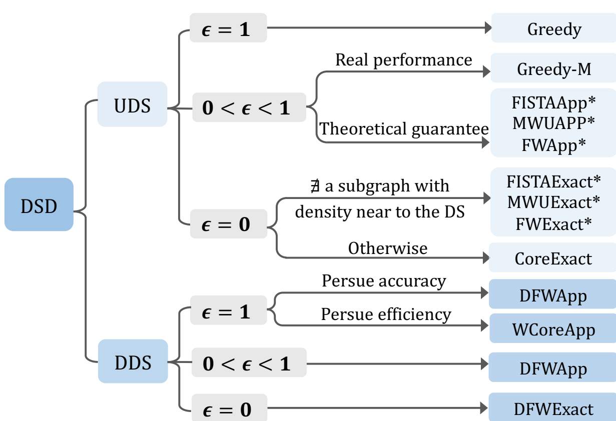


Figure 20: The taxonomy tree of DSD algorithms.

using only their APIs. Therefore, enabling these popular graph systems to support DSD algorithms presents an exciting opportunity.

8 RELATED WORKS

In this section, we review the related works, including the variants of the UDS and DDS solutions and dense subgraph discovery.

• **Other variants of DSD.** Many variants of DSD have been studied [5, 23, 34, 65, 80, 85, 95]. The clique densest subgraph (CDS) problem is proposed to better detect “near-clique” subgraphs [31, 33, 43, 61, 71, 90]. Note that when $k = 2$, it reduces to the UDS problem. The network flow-based [33, 71, 90] and convex programming-based [43, 88] algorithms are developed to solve this problem. The size-constrained DSD problems are studied in undirected and directed graphs [6, 12–14, 16, 41, 52, 77]. Another version of DSD called optimal quasi-clique [91, 92] extracts a subgraph that is more compact, with a smaller diameter than the DSD. To identify locally dense regions, Qin et al.[80] and Ma et al. [65] studied the top-k locally DS problem. Veldt et al. [93] studied the P-mean DSD problem and proposed a generalized peeling algorithm. To personalize results, the anchored DSD problem [23] aims to maximize R-subgraph density of the subgraphs containing an anchored node set. Besides, the DSD problems in bipartite, multilayer, and uncertain graphs were studied [4, 36, 37, 44, 49, 71, 73]. Recently, the fair DS problem and diverse DS problems [3, 72] have been explored to achieve equitable outcomes and overcome algorithmic bias.

• **Dense subgraph discovery.** Another group of works highly related to DSD are about dense subgraph discovery. Many cohesive subgraph models like k-core [10, 86], k-truss [22, 84, 98], k-ECC [45, 97], k-clique [24], quasi-clique [1], and k-plex [9, 100] have been studied. k-core is one of the most widely used dense subgraphs, in which all vertices have a degree of at least k. k-truss is a dense subgraph based on the constraint of the number of triangles, in which each edge is contained by at least k-2 triangles. k-ECC is a subgraph based on edge connectivity, and the edge connectivity of two vertices u and v is the minimum value of the number of edges that need to be removed to make u and v disconnected. k-clique is a complete graph of size k, while k-plex allows for up to k missing connections in a clique, reflecting a quasi-clique model. These models are often used in community search as they are cornerstone of the community [30]. Besides, these models are extended to bipartite graphs, such as (α, β)-core [27, 58], bitruss [94, 101], biclique [63], and biplex [60, 96]. The directed dense subgraph models such as D-core [32, 38, 57] and D-truss [59] have also been studied. Nevertheless, these works are different from DSD since they do not use the density definition as a key metric.

To the best of our knowledge, our work is the first study that provides a unified framework for all existing DSD algorithms and compares existing solutions comprehensively via empirical results.

9 CONCLUSIONS

In this paper, we provide an in-depth experimental evaluation and comparison of existing Densest Subgraph Discovery (DSD) algorithms. We first provide a unified framework, which can cover all the existing DSD algorithms, including exact and approximation algorithms, using an abstraction of a few key operations. We then thoroughly analyze and compare different DSD algorithms under our framework for both undirected and directed graphs, respectively. We further systematically evaluate these algorithms from different angles using various datasets, and also develop variations by combining existing techniques, which often outperform state-of-the-art methods. From extensive empirical results and analysis, we have identified several important findings and analyzed the critical components that affect the performance. In addition, we have summarized the lessons learned and proposed practical research opportunities that can facilitate future studies.

REFERENCES

[1] James Abello, Mauricio GC Resende, and Sandra Sudarsky. 2002. Massive quasi-clique detection. In *LATIN*. Springer, 598–612.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Diameter of the world-wide web. *nature* 401, 6749 (1999), 130–131.

[3] Aris Anagnostopoulos, Luca Becchetti, Adriano Fazzone, Cristina Menghini, and Chris Schwiegelshohn. 2020. Spectral relaxations and fair densest subgraphs. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 35–44.

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

[5] Reid Andersen and Kumar Chellapilla. 2009. Finding dense subgraphs with size bounds. In *WAW*. Springer, 25–37.

[6] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 2000. Greedily finding a dense subgraph. *Journal of Algorithms* 34, 2 (2000), 203–221.

[7] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient primal-dual graph algorithms for mapreduce. In *WAW*. Springer, 59–78.

[8] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest Subgraph in Streaming and MapReduce. *PVLDB* 5, 5 (2012).

[9] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V Hicks. 2011. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research* 59, 1 (2011), 133–142.

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

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

[12] 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*. 201–210.

[13] Aditya Bhaskara, Moses Charikar, Venkatesan Guruswami, Aravindan Vijayaraghavan, and Yuan Zhou. 2012. Polynomial integrality gaps for strong sdp relaxations of densest k-subgraph. In *SODA*. SIAM, 388–405.

[14] Francesco Bonchi, David García-Soriano, Atsushi Miyauchi, and Charalampos E Tsourakakis. 2021. Finding densest k-connected subgraphs. *Discrete Applied Mathematics* 305 (2021), 34–47.

[15] 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*.

[16] Nicolas Bourgeois, Aristotelis Giannakos, Giorgio Lucarelli, Ioannis Milis, and Vangelis Th Paschos. 2013. Exact and approximation algorithms for densest k-subgraph. In *WALCOM*. Springer, 114–125.

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

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

[19] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *TKDE* 24, 7 (2010), 1216–1230.

[20] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *PVLDB* 8, 12 (2015), 1804–1815.

[21] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

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

[23] Yizhou Dai, Miao Qiao, and Lijun Chang. 2022. Anchored Densest Subgraph. In *SIGMOD*. 1200–1213.

[24] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *WWW*. 589–598.

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

[26] Laxman Dhulipala, Quanquan C Liu, Sofya Raskhodnikova, Jessica Shi, Julian Shun, and Shangdi Yu. 2022. Differential privacy from locally adjustable graph algorithms: k-core decomposition, low out-degree ordering, and densest subgraphs. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 754–765.

[27] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient fault-tolerant group recommendation using alpha-beta-core. In *CIKM*. 2047–2050.

[28] Michael Dinitz, Satyen Kale, Silvio Lattanzi, and Sergei Vassilvitskii. 2023. Improved Differentially Private Densest Subgraph: Local and Purely Additive. *arXiv preprint arXiv:2308.10316* (2023).

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

[30] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDBJ* 29, 1 (2020), 353–392.

[31] Yixiang Fang, Wensheng Luo, and Chenhao Ma. 2022. Densest subgraph discovery on large graphs: Applications, challenges, and techniques. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3766–3769.

[32] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *TKDE* 31, 11 (2018), 2093–2107.

[33] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *PVLDB* 12, 11 (2019), 1719–1732.

[34] Uriel Feige, Michael Seltser, et al. 1997. *On the densest k-subgraph problem*. Citeseer.

[35] Eugene Fratkin, Brian T Naughton, Douglas L Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics* 22, 14 (2006), e150–e157.

[36] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core decomposition and densest subgraph in multilayer networks. In *CIKM*. 1807–1816.

[37] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core decomposition in multilayer networks: theory, algorithms, and applications. *TKDD* 14, 1 (2020), 1–40.

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

[39] Aristides Gionis and Charalampos E Tsourakakis. 2015. Dense subgraph discovery: Kdd 2015 tutorial. In *SIGKDD*. 2313–2314.

[40] Andrew V Goldberg. 1984. *Finding a maximum density subgraph*. University of California Berkeley.

[41] Sean Gonzales and Theresa Migler. 2019. The Densest k Subgraph Problem in b-Outerplanar Graphs. In *COMPLEX NETWORKS*. Springer, 116–127.

[42] Elfarouk Harb, Kent Quanrud, and Chandra Chekuri. 2022. Faster and Scalable Algorithms for Densest Subgraph and Decomposition. In *NIPS*.

[43] Yizhang He, Kai Wang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2023. Scaling Up k-Clique Densest Subgraph Detection. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[44] 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 *SIGKDD*. 895–904.

[45] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying minimal steiner maximum-connected subgraphs in large graphs. In *CIKM*. 1241–1250.

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

[47] Martin Jaggi. 2013. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML*. PMLR, 427–435.

[48] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. 2007. Why we twitter: understanding microblogging usage and communities. In *WebKDD/SNA-KDD*. 56–65.

[49] Vinay Jethava and Niko Beerenwinkel. 2015. Finding dense subgraphs in relational graphs. In *ECML PKDD*. Springer, 641–654.

[50] Ravindran Kannan and V Vinay. 1999. *Analyzing the structure of large graphs*. Forschungsinst. für Diskrete Mathematik.

[51] Guy Karlebach and Ron Shamir. 2008. Modelling and analysis of gene regulatory networks. *Nature reviews Molecular cell biology* 9, 10 (2008), 770–780.

[52] Yasushi Kawase and Atsushi Miyauchi. 2018. The densest subgraph problem with a convex/concave size function. *Algorithmica* 80, 12 (2018), 3461–3480.

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

[54] Konect. 2006. Konect. http://konect.cc/networks/.

[55] Laks VS Lakshmanan. 2022. On a Quest for Combating Filter Bubbles and Misinformation. In *SIGMOD*. 2–2.

[56] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzone, and Francesco Bonchi. 2023. A Survey on the Densest Subgraph Problem and its Variants. *arXiv preprint arXiv:2303.14467* (2023).

[57] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core Decomposition over Large Directed Graphs. *PVLDB* 15, 8 (2022), 1546–1558.

[58] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.

[59] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *SIGMOD*. 2183–2197.

[60] Wensheng Luo, Kenli Li, Xu Zhou, Yunjun Gao, and Keqin Li. 2022. Maximum Biplex Search over Bipartite Graphs. In *ICDE*. IEEE, 898–910.

[61] Wensheng Luo, Chenhao Ma, Yixiang Fang, and Laks VS Lakshman. 2023. A Survey of Densest Subgraph Discovery on Large Graphs. *arXiv preprint arXiv:2306.07927* (2023).

[62] Wensheng Luo, Zhuo Tang, Yixiang Fang, Chenhao Ma, and Xu Zhou. 2023. Scalable Algorithms for Densest Subgraph Discovery. In *ICDE*. IEEE.

[63] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *PVLDB* 13, 9 (2020), 1359–1372.

[64] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. Linc: a motif counting algorithm for uncertain graphs. *PVLDB* 13, 2 (2019), 155–168.

[65] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. Finding locally densest subgraphs: a convex programming approach. *PVLDB*

15, 11 (2022), 2719–2732.
[66] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *SIGMOD*. 845–859.
[67] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Xiaolin Han, and Xiaodong Li. 2023. Accelerating directed densest subgraph queries with software and hardware approaches. *The VLDB Journal* (2023), 1–24.
[68] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient algorithms for densest subgraph discovery on large directed graphs. In *SIGMOD*. 1051–1066.
[69] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. Efficient Directed Densest Subgraph Discovery. *ACM SIGMOD Record* 50, 1 (2021), 33–40.
[70] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On Directed Densest Subgraph Discovery. *TODS* 46, 4 (2021), 1–45.
[71] 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 *SIGKDD*. 815–824.
[72] Atsushi Miyauchi, Tianyi Chen, Konstantinos Sotiropoulos, and Charalampos E Tsourakakis. 2023. Densest Diverse Subgraphs: How to Plan a Successful Cocktail Party with Diversity. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1710–1721.
[73] Atsushi Miyauchi and Akiko Takeda. 2018. Robust densest subgraph discovery. In *ICDM*. IEEE, 1188–1193.
[74] nebula. 2010. nebula. https://www.nebula-graph.io/.
[75] neo4j. 2006. neo4j. https://neo4j.com/.
[76] Yu E Nesterov. 1983. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. In *Dokl. Akad. Nauk SSSR,*, Vol. 269. 543–547.
[77] Tim Nonner. 2016. PTAS for Densest k-Subgraph in Interval Graphs. *Algorithmica* 74, 1 (2016), 528–539.
[78] Laboratory of Web Algorithmics. 2013. Laboratory of Web Algorithmics Datasets. http://law.di.unimi.it/datasets.php.
[79] Stanford Network Analysis Project. 2009. SNAP. http://snap.stanford.edu/data/.
[80] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally densest subgraph discovery. In *KDD*. 965–974.
[81] The Technique Report. 2024. In-depth Analysis of Densest Subgraph Discovery in a Unified Framework (technical report). https://github.com/TalionS/DensestSubgraph/blob/master/full_version.pdf.
[82] Network Repository. 2014. Network Repository. https://networkrepository.com/network-data.php.
[83] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense subgraphs with restrictions and applications to gene annotation graphs. In *RECOMB*. Springer, 456–472.
[84] Kazumi Saito, Takeshi Yamada, and Kazuhiro Kazama. 2008. Extracting communities from complex networks by the k-dense method. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 91, 11 (2008), 3304–3311.
[85] Saurabh Sawlani and Junxing Wang. 2020. Near-optimal fully dynamic densest subgraph. In *STOC*. 181–193.
[86] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.
[87] Pattara Sukprasert, Quanquan C Liu, Laxman Dhulipala, and Julian Shun. 2024. Practical Parallel Algorithms for Near-Optimal Densest Subgraphs on Massive Graphs. In *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 59–73.
[88] Bintao Sun, Maximilien Danisch, TH Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. *PVLDB* (2020).
[89] tigergraph. 2010. tigergraph. https://www.tigergraph.com/.
[90] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *WWW*. 1122–1132.
[91] 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 *SIGKDD*. 104–112.
[92] Charalampos E Tsourakakis. 2014. Mathematical and algorithmic analysis of network and biological data. *arXiv preprint arXiv:1407.0375* (2014).
[93] Nate Veldt, Austin R Benson, and Jon Kleinberg. 2021. The generalized mean densest subgraph problem. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1604–1614.
[94] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* 31, 2 (2022), 203–226.
[95] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2023. Efficient and Effective Algorithms for Generalized Densest Subgraph Discovery. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[96] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *SIGMOD*. ACM, 860–873.
[97] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. I/O efficient ECC graph decomposition via graph reduction. *The VLDB Journal* 26, 2 (2017), 275–300.
[98] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting analyzing and visualizing triangle k-core motifs within networks. In *ICDE*. IEEE, 1049–1060.
[99] Yingli Zhou, Qingshuo Guo, Yixiang Fang, and Chenhao Ma. 2024. A Counting-based Approach for Efficient k-Clique Densest Subgraph Discovery. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[100] Yi Zhou, Shan Hu, Mingyu Xiao, and Zhang-Hua Fu. 2021. Improving maximum k-plex solver via second-order reduction and graph color bounding. In *AAAI*, Vol. 35. 12453–12460.
[101] Zhaonian Zou. 2016. Bitruss decomposition of bipartite graphs. In *DASFAA*. Springer, 218–233.

A ADDITIONAL DEFINITIONS

A.1 The LP formulation and Dual of UDS problem

The following is a well-known LP formulation of the UDS problem, introduced in [17]. Associate each vertex v with a variable $x_v \in \{0,1\}$, where $x_v = 1$ signifies v being included in $\mathcal{D}(G)$. Similarly, for each edge, let $y_e \in \{0,1\}$ denote whether or not it is in $E(\mathcal{D}(G))$. Relaxing the variables to be real numbers, we get the following LPs, which we denote by $\mathsf{LP}(G)$, whose optimal is known to be ρ^*_G [17]:

$$\begin{aligned}
\mathsf{LP}(G) \quad & \max \sum_{e\in E} y_e && && (2)\\
\text{s.t.} \quad & y_e \le x_u, x_v, && \forall e=(u,v)\in E && (3)\\
& \sum_{v\in V} x_v \le 1, && && (4)\\
& y_e \ge 0, x_v \ge 0, && \forall e\in E, \forall v \in V && (5)
\end{aligned}$$

The Lagrangian dual $\mathsf{DP}(G)$ of the $\mathsf{LP}(G)$ is as follows [7, 25]:

$$\begin{aligned}
\mathsf{DP}(G) \quad & \min \max_{v\in V} \mathbf{w}(v) && && (6)\\
\text{s.t.} \quad & \mathbf{w}(v) = \sum_{e\in E} \alpha_e(v), && \forall v\in V && (7)\\
& \alpha_e(u) + \alpha_e(v) = 1, && \forall e=(u,v)\in E && (8)\\
& \alpha_e(u) \ge 0, \alpha_e(v) \ge 0 && \forall e=(u,v)\in E && (9)
\end{aligned}$$

Note that $\|\mathbf{w}\|_\infty = \max_{v\in V} \mathbf{w}(v)$, which means that the objective function of $\mathsf{DP}(G)$ is: $\min \|\mathbf{w}\|_\infty$. This DP can be visualized as follows. Each edge $e=(u,v)$ has a weight of 1, which it wants to assign to its endpoints: $\alpha_e(u)$ and $\alpha_e(v)$ such that the total weight on each vertex is at most $\|\mathbf{w}\|_\infty$. The objective is to find the minimum $\|\mathbf{w}\|_\infty$ for which such a load assignment is feasible. Here, we introduce the vertex weight vector $\mathbf{w}$:

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}(v_1) & \mathbf{w}(v_2) & \cdots & \mathbf{w}(v_n) \end{bmatrix}$$

A.2 More Definitions of DDS problem

Definition A.1 (c-biased density[66]). Given a directed graph $D=(V,E)$, a fixed c and two vertex sets S and T, the c-biased density of an (S,T)-induced subgraph is proportional to its edge-density, and is defined as $\rho_c(S,T) = \frac{2\sqrt{c}\sqrt{c'}}{c+c'}\rho(S,T) = \frac{2\sqrt{c}\sqrt{c'}}{c+c'}\frac{|E(S,T)|}{\sqrt{|S||T|}}$, where $c' = \frac{|S|}{|T|}$. Note when $c'=c$, $\rho_c(S,T) = \rho(S,T)$.

Definition A.2 (c-biased directed densest subgraph (DDS) [66]). Given a directed graph D and a fixed c, the subgraph, whose corresponding c-biased density is the highest among all possible ones, is called the c-biased DDS.

Charikar [17] proposed the first exact DDS solution, which is based on linear programming (LP). Because the DDS is related to two vertex subsets S and T, Charikar formulated the DS problem to a series of linear programs w.r.t. the possible values of $c = \frac{|S|}{|T|}$. Because the ratio $c = \frac{|S|}{|T|}$ is not known in advance, there are $O(n^2)$ possible values, which result in $O(n^2)$ different LPs. For each $c = \frac{|S|}{|T|}$, the corresponding LP is formulated by Equation (10).

$$\begin{aligned}
\mathsf{LP}(c) \quad \max \quad & x_{\text{sum}} = \sum_{(u,v)\in E} x_{u,v} && \\
\text{s.t.} \quad & 0 \le x_{u,v} \le s_u, && \forall (u,v)\in E\\
& x_{u,v} \le t_v, && \forall (u,v)\in E\\
& \sum_{u\in V} s_u = \sqrt{c}, &&\\
& \sum_{v\in V} t_v = \frac{1}{\sqrt{c}}. &&
\end{aligned} \tag{10}$$

The variables in Eq. (10) can be used to infer the DDS when $c = \frac{|S^*|}{|T^*|}$. Specifically, s_u, t_v, and $x_{u,v}$ indicate the inclusion of a vertex u/vertex v/edge (u,v) in an optimal densest subgraph according to whether the variable value is larger than 0, when $c = \frac{|S^*|}{|T^*|}$. To find the DDS, Charikar's algorithm needs to solve $O(n^2)$ LPs with LP solvers.

To reduce the number of LPs to be solved, Ma et al. [66] introduced a relaxation, $a+b=2$, to the LP formulation of the DDS problem, as shown in Equation (11).

$$\begin{aligned}
\mathsf{LP}(c) \quad \max \quad & x_{\text{sum}} = \sum_{(u,v)\in E} x_{u,v} && \\
\text{s.t.} \quad & x_{u,v} \ge 0, && \forall (u,v)\in E\\
& x_{u,v} \le s_u, && \forall (u,v)\in E\\
& x_{u,v} \le t_v, && \forall (u,v)\in E\\
& \sum_{u\in V} s_u = a\sqrt{c}, &&\\
& \sum_{v\in V} t_v = \frac{b}{\sqrt{c}}, &&\\
& a+b = 2. &&
\end{aligned} \tag{11}$$

Comparing Equation (11) with Equation (10), we can find that the two formulations are identical if we restrict $a=1$ and $b=1$. By introducing the relaxation, Ma et al. [66] managed to build the connection between each LP and the DDS. Based on the connection, they developed a divide-and-conquer strategy to reduce the number of LPs to be solved.

Ma et al. [66] derived the convex program (CP) of the linear programming formulation of the DDS problem. For each $c = \frac{|S|}{|T|}$, the corresponding CP is formulated by Equation (12).

$$\begin{aligned}
\mathsf{CP}(c) \quad \min \quad & \max_{u\in V}\{|\mathbf{w}_\alpha(u)|, |\mathbf{w}_\beta(u)|\} && \\
\text{s.t.} \quad & \alpha_{u,v} + \beta_{v,u} = 1, && \forall (u,v)\in E\\
& 2\sqrt{c}\sum_{(u,v)\in E} \alpha_{u,v} = \mathbf{w}_\alpha(u), && \forall u\in V\\
& \frac{2}{\sqrt{c}}\sum_{(u,v)\in E} \beta_{v,u} = \mathbf{w}_\beta(u), && \forall u\in V\\
& \alpha_{u,v}, \beta_{v,u} \ge 0, && \forall (u,v)\in E
\end{aligned} \tag{12}$$

Besides, Ma et al [66] presented the convergence of employing Frank-Wolfe algorithm to solve Equation (12), which is given by the following theorem:

Theorem A.3 ([66]). *Given a directed graph D and c, suppose d^+_{max} (resp. d^-_{max}) is the maximum out-degree (resp. in-degree) of D. For $t > 16\frac{\kappa m}{\varepsilon^2}$, we have $\|\max_{u\in V}\{|\mathbf{w}_\alpha(u)|, |\mathbf{w}_\beta(u)|\}\|_\infty - \rho^* \le \varepsilon$,* *where* $\kappa = \sum_{c \in C}(\sqrt{c} + \frac{1}{\sqrt{c}}) \max\{\sqrt{c} d^{+}_{max}, \frac{1}{\sqrt{c}} d^{-}_{max}\}$, *and* $C = \{\frac{a}{b} \mid 1 \le a, b \le n\}$.

B MORE DISCUSSIONS

B.1 The equality of the two divide-and-conquer strategies

To reduce the number of $\frac{|S|}{|T|}$ values that need to be examined, DCExact [68] first proposed a divide and conquer method to reduce the number of $\frac{|S|}{|T|}$ values examined from n^2 to k, where theoretically $k \le n^2$, but practically $k \ll n^2$. This method effectively omits searching for some values of c. DFWExact [66] introduced a new divide and conquer method, utilizing the relationship between the densest subgraph and the c-biased densest subgraph to skip searches for certain c values in the DDS process. In this study, we thoroughly analyze these two strategies and, surprisingly, find them to be theoretically identical. Specifically, for a given c, DCExact [68] solves such optimization problem:

$$\begin{aligned} \max_{S,T \in V} \quad & g \\ \text{s.t.} \quad & \frac{|S|}{\sqrt{c}}\left(g - \frac{|E(S,T)|}{|S|/\sqrt{c}}\right) + |T|\sqrt{c}\left(g - \frac{|E(S,T)|}{|T|\sqrt{c}}\right) \le 0. \end{aligned} \tag{13}$$

Assume that S' and T' are the optimal choices for Equation 13, we can derive that

$$g^*(c) \le \frac{2\rho(S',T')}{\frac{\sqrt{c'}}{\sqrt{c}} + \frac{\sqrt{c}}{\sqrt{c'}}} = \frac{2\sqrt{c}\sqrt{c'}}{c + c'}\rho(S',T') = \rho_c(S',T') \tag{14}$$

where $c' = \frac{|S'|}{|T'|}$. Equation (14) reveals that the divide and conquer strategy in DCExact [68] is also based on c-biased DDS. To be more specific, DCExact [68] utilizes network flow computation to find the c-biased DDS, while DFWExact [66] obtains it via Frank-Wolfe.

B.2 The verification of CP-based UDS Exact algorithms

For the CP-based UDS Exact algorithms, we need to use the following algorithm for optimal result verification.

Algorithm 7: Verification of CP-based Exact algorithms

input : S=($V(S)$, $E(S)$), $\mathbf{w}$
output: Whether S is a densest subgraph of G
1 **if** *$V(S)$ is a stable set* **then**
2   **if** *the Theorem B.2 is satisfied* **then return** True;
3     $n \leftarrow |V(S)|$; $m \leftarrow |E(S)|$;
4   build flow network $\mathcal{F}$ on S;
5     $f \leftarrow$ maximum flow from s to t;
6   **return** $f = nm$;
7 **return** False;

After a sufficient number of iterations, CP-based algorithms tend to converge to the optimal solution. We begin by explaining a stable set to verify this.

Definition B.1 (Stable Set [25, 88]). A non-empty vertex set $S \subseteq V$ is a stable set if the following conditions hold:

(1) $\forall u \in S$ and $v \in V \setminus S$, $\mathbf{w}(u) > \mathbf{w}(v)$.
(2) $\forall v \in V \setminus S$, we have $\forall (u, v) \in E \cap (S \times \{v\})$, $\alpha_{u,v} = 0$.

Next, we present a theorem for optimal testing.

Theorem B.2 (Improved Goldberg's Condition [88]). *Given a vertex weight vector* $\mathbf{w}$ *on subgraph* S *of* G *and supposed* $\mathbf{w}(u_1) \ge \mathbf{w}(u_2) \ge \cdots \ge \mathbf{w}(u_{|n|})$. *Let* $n = |V(S)|$, *and* $m = |E(S)|$, *if* $\forall i \in [0, n-1]$, $\min\left\{\frac{1}{i}\binom{i}{2}, \frac{1}{i}\sum_{j=1}^{i}\mathbf{w}(u_j)\right\} - \rho(S) < \max\left\{\frac{1}{ni}, \frac{1}{i}\left(\lceil\frac{im}{n}\rceil - \frac{im}{n}\right)\right\}$, *then* S *is densest subgraph of* G.

Algorithm 7 presents the details of using maximum flow for optimal result testing. Specifically, if S is a stable set and either the condition of Theorem B.2 is satisfied or there is a feasible flow with value nm exists in $\mathcal{F}$, then the Algorithm 7 will return true; otherwise it will return false (lines 1-9).

B.3 An example of FlowExact under our unified framework

Here, we first offer a concrete example illustrating how to run FlowExact using the three steps of our unified framework, as shown in Algorithms from 8 to 10. Note that CoreExact follows the same steps as the FlowExact, the only difference is that CoreExact includes an extra graph reduction step (e.g., the first stage in our framework).

Algorithm 8: FlowExact under our unified framework

input : $G = (V, E)$, $\epsilon = 0$, $T = 1$
output: An exact densest subgraph $\mathcal{D}(G)$
$f \leftarrow$ False; $\underline{\rho} \leftarrow 0$; $\overline{\rho} \leftarrow \Delta(G)$;
repeat
    $\mathbf{w} \leftarrow$ FlowExactVWU(G, $\mathbf{w}$, T, $\underline{\rho}$, $\overline{\rho}$);
    $(f, \mathcal{D}(G), \underline{\rho}, \overline{\rho}) \leftarrow$ FlowExactCSV(G, $\underline{\rho}$, $\overline{\rho}$, $\mathbf{w}$, ϵ);
until f=True;
return $\mathcal{D}(G)$;

Algorithm 9: FlowExactVWU (G, $\mathbf{w}$, T, $\underline{\rho}$, $\overline{\rho}$)

❶ initialize $\mathbf{w}$ and auxiliary variables;
$g \leftarrow (\underline{\rho} + \overline{\rho})/2$; $f \leftarrow \mathbf{0}$;$\mathbf{w} \leftarrow \mathbf{0}$;
build flow network $\mathcal{F}$ based on G and g;
while *the maximum flow has not been reached* **do**
    ❷ update the auxiliary variables;
    foreach $e = (u, v) \in E(\mathcal{F})$ **and** $f(e)$ *can be increased* **do**
        increase $f(e)$;
    ❸ update $\mathbf{w}$ via $f(e)$;
    foreach $v \in V(G)$ ***and*** *the flow from v to t changed* **do**
        $\mathbf{w}(v) \leftarrow$ the flow from v to t;
return $\mathbf{w}$;

Algorithm 10: FlowExactCSV (G, $\underline{\rho}$, $\overline{\rho}$,$\mathbf{w}$, ϵ)

find the minimum st-cut $(\mathcal{S}, \mathcal{T})$ from $\mathcal{F}$ based on $\mathbf{w}$;
$g \leftarrow (\underline{\rho} + \overline{\rho})/2$;
if $\mathcal{S} = \{s\}$ **then** $\overline{\rho} \leftarrow g$; ;
else
    $\underline{\rho} \leftarrow g$; $\mathcal{S} \leftarrow \mathcal{S} \backslash \{s\}$;
if $\overline{\rho} - \underline{\rho} < \frac{1}{n \cdot (n-1)}$ **then** $f \leftarrow$ True ;
return f, $\mathcal{S}$, $\underline{\rho}$, $\overline{\rho}$;

Specifically, in the VWU stage of FlowExact, the vertex weights denote the flow from those vertices to the target node t in the

Method	Stage (1): ReduceGraph	Stage (2): VWU	Stage (3): CSV
FlowExact	no reduction	compute the maximum flow	❶ extract the minimum cut;
DCExact	locate graph into certain $[x, y]$-core		❷ verify if it is optimal;
CP-based	locate graph into certain $[x, y]$-core	optimize $\text{CP}(c)$;	❶ extract the maximum prefix sum set; ❷ verify if it is exact or satisfies the approximation ratio criteria;
Peeling-based	no reduction	iteratively remove vertices	❶ extract the subgraph with the highest density during the peeling;

Table 11: Overview of the three stages of the existing DDS algorithms.

flow network of the graph. Here, we use an auxiliary variable $f(e)$ to denote the flow of each edge, and FlowExact computes the maximum flow. Note that the key idea of all maximum flow algorithms is saturating edge flows to reach the maximum flow from s to t. We represent the flow from vertex v to the target node t as $\mathbf{w}(v)$. The algorithm stops when it reaches the maximum flow. If not, it means that the flows of some edges in $\mathcal{F}$ can be increased. We increase the flows and update $\mathbf{w}$ (lines 6-10); If yes, $\mathbf{w}(s)$ is the maximum flow of the network $\mathcal{F}$. In the CSV stage of FlowExact, we first utilize the vertex weights to obtain the minimum st-cut $(\mathcal{S}, \mathcal{T})$ of G, and then if $\mathcal{S}$ contains only the source node $\{s\}$, $\overline{\rho}$ is updated to g; otherwise, $\underline{\rho}$ is set to g and $\mathcal{S} \setminus \{s\}$ is returned as the candidate subgraph. Besides, in Algorithm 10, to verify the quality of the candidate subgraph, FlowExact checks if the difference between $\overline{\rho}$ and $\underline{\rho}$ is less than $\frac{1}{n\cdot(n-1)}$, as the density difference between any two subgraphs must be larger than $\frac{1}{n\cdot(n-1)}$. When the condition is satisfied, the exact solution has been found.

B.4 Overview of the existing DDS algorithms

We illustrate the details of these three stages for each category of DSD algorithms for directed graphs in Table 11, which is very similar to the algorithms for undirected graphs.

Algorithm 11: DFWExact under our unified framework

input : $D = (V, E)$, $\epsilon = 0$, T
output: An exact densest subgraph $\mathcal{D}(D)$
1 $f \leftarrow$ False; $\underline{\rho} \leftarrow \sqrt{\Delta(D)}$; $\overline{\rho} \leftarrow \Delta(D)$;
2 **repeat**
3 $\quad D' \leftarrow$ReduceGraph$(D, \underline{\rho}, c_l, c_r)$;
4 $\quad \mathbf{w} \leftarrow$ DFWExactVWU$(D', \mathbf{w}, T, \underline{\rho}, \overline{\rho}, c_l, c_r)$;
5 $\quad (f, \mathcal{D}(D), \underline{\rho}, c_l, c_r) \leftarrow$ DFWExactCSV$(D', \underline{\rho}, \overline{\rho}, \mathbf{w}, \epsilon, c_l, c_r)$;
6 **until** f=True;
7 **return** $\mathcal{D}(D)$;

Here, we provide an example to illustrate how to run DFWExact using the three steps of our unified framework, as shown in Algorithms from 11 to 13. Specifically, in the VWU stage of DFWExact, it first initializes $\alpha_e^{(0)}(v) = 1/2$ for each edge. For each vertex $u \in V'$, its in (resp. out) weight is initialized as $\mathbf{w}^{(0)}(u^+) = 2\sqrt{c}\sum_{(u,v)\in E}\alpha_{u,v}^{(0)}$ (resp. $\mathbf{w}^{(0)}(u^-) \leftarrow \frac{2}{\sqrt{c}}\sum_{(u,v)\in E'}\alpha_{v,u}^{(0)}$) (lines 3-5). Next, it repeats T iterations to update α, and $\mathbf{w}$. In each iteration, it computes $\widehat{\alpha}$ and updates $\alpha^{(t)}$ based on α^{t-1} and $\widehat{\alpha}$ (lines 8-9). The algorithm aggregates $\alpha^{(t)}$ to obtain $\mathbf{w}^{(t)}$ (lines 11-12).

In the CSV stage of DFWExact, the vertices with higher weights are more likely to appear in the densest subgraph $\mathcal{D}$, since they are linked by more edges. Thus, the subgraph induced by the first s^* vertices with the largest weights is returned as the candidate subgraph (lines 4-5). Next, the algorithm uses the maximum flow to test the result optimally. DFWExactCSV first checks whether $D(S, T)$ is a stable set; if so, it verifies whether a feasible flow with value $|E(S, T)|$ exists in $\mathcal{F}$ and confirms that all c values have been checked. Otherwise, it returns false (lines 6-17).

Algorithm 12: DFWExactVWU $(D', \mathbf{w}, T, \underline{\rho})$

1 ❶ initialize $\mathbf{w}$ and auxiliary variables;
2 $c \leftarrow \frac{c_l + c_r}{2}$;
3 **foreach** $(u, v) \in E'$ **do** $\alpha_{u,v}^{(0)} \leftarrow 1/2$; $\alpha_{v,u}^{(0)} \leftarrow 1/2$; $t \leftarrow 1$;
4 **foreach** $u \in V'$ **do** $\mathbf{w}^{(0)}(u^+) \leftarrow 2\sqrt{c}\sum_{(u,v)\in E'}\alpha_{u,v}^{(0)}$;
5 **foreach** $u \in V'$ **do** $\mathbf{w}^{(0)}(u^-) \leftarrow \frac{2}{\sqrt{c}}\sum_{(u,v)\in E'}\alpha_{v,u}^{(0)}$;
6 **while** $t \leq T$ **do**
7 $\quad$ ❷ update the auxiliary variables;
8 $\quad$ **foreach** $(u, v) \in E'$ **do** update $\widehat{\alpha}_{u,v}$ and $\widehat{\alpha}_{v,u}$;
9 $\quad \alpha^{(t)} \leftarrow (1-\gamma_t)\cdot\alpha^{(t-1)} + \gamma_t \cdot \widehat{\alpha}$, with $\gamma_t = \frac{2}{t+2}$;
10 $\quad$ ❸ update $\mathbf{w}$ via α;
11 $\quad$ **foreach** $u \in V'$ **do** $\mathbf{w}^{(t)}(u^+) \leftarrow 2\sqrt{c}\sum_{(u,v)\in E'}\alpha_{u,v}^{(t)}$;
12 $\quad$ **foreach** $u \in V'$ **do** $\mathbf{w}^{(t)}(u^-) \leftarrow \frac{2}{\sqrt{c}}\sum_{(u,v)\in E'}\alpha_{v,u}^{(t)}$;
13 $\quad t \leftarrow t + 1$;
14 **return** $\mathbf{w}$;

Algorithm 13: DFWExactCSV $(D', \underline{\rho}, \overline{\rho}, \mathbf{w}, \epsilon, c_l, c_r)$

1 **foreach** $1 \leq i \leq 2|V'|$ **do**
2 $\quad u_i \leftarrow$ the vertex with the i-th highest out/in weight in V';
3 $\quad D_i = (S_i, T_i) \leftarrow$ the induced subgraph of top-i weight vertices;
4 $s^* \leftarrow \arg\max_{1\leq s\leq 2n}\rho_c(S_s, T_s)$;
5 $D(S, T) \leftarrow$ the subgraph induced by the first s^* vertices;
6 **if** $D(S, T)$ *is a stable set* **then**
7 $\quad n \leftarrow |S \cap T|$; $m \leftarrow |E(S, T)|$;
8 $\quad$ build flow network $\mathcal{F}$ on $D(S, T)$;
9 $\quad f \leftarrow$ maximum flow from s to t;
10 $\quad$ **if** $f = |E(S, T)|$ **then**
11 $\quad\quad$ **if** $\rho(D(S, T)) > \underline{\rho})$ **then**
12 $\quad\quad\quad \mathcal{D}(D) \leftarrow D(S, T)$;
13 $\quad\quad\quad \underline{\rho} = \max(\rho(D(S, T)), \underline{\rho})$;
14 $\quad\quad$ **if** *all c values has been checked* **then**
15 $\quad\quad\quad f \leftarrow$ True;
16 $\quad\quad$ **else** update c_l and c_r ;
17 **return** f; $\mathcal{D}(D)$, $\underline{\rho}$, $\overline{\rho}$, c_l, c_r;

Table 12: The stop conditions and the meanings of vertex weights for the VWU stages in the UDS algorithms.

Method	Stop condition of while-loop	The meanings of w
FlowExact	the maximum flow has been reached	flows from the vertices to the target node
CoreExact	the maximum flow has been reached	flows from the vertices to the target node
FlowApp	perform h blocking flows	flows from the vertices to the target node
CP-based	optimize CP formulation for T iterations	the weight sum received by each vertex
Peeling	iteratively remove vertices for T iterations	the vertex weights used to select which vertex should be removed

B.5 Extending our framework into the variants DSD algorithms

It is worth noting that our framework can also incorporate the variants of DSD problems that utilize the generalized supermodular density as a density metric. For example, Xu et al. [18, 95] proposed the generalized densest subgraph (GDS) problem, which includes weighted UDS problem [40, 44], clique densest subgraph problem [33, 43, 99], and DSD problem on hyper-graphs [46], and developed exact solutions for the GDS problem [95] based on graph reduction and maximum flow.

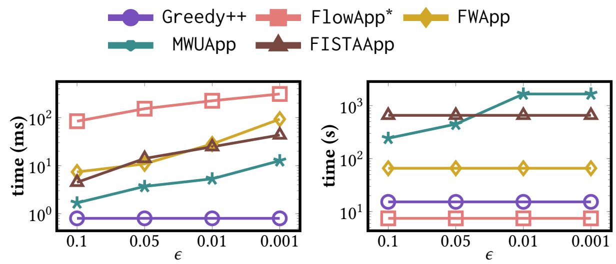
B.6 More Lessons and Opportunities

L7. For the exact UDS algorithms, CoreExact and those new CP-based exact algorithms achieve comparable performance. For the DDS problem, DFWExact is always the best one.
L8. For all the DSD algorithms, including exact and approximation, the memory usage is almost the same, scaling linearly with the number of edges in the graph.
L9. For all DSD algorithms, both exact and approximate, the density of the original graph does not directly affect their running time.
O5. An interesting future research direction is to study the application-driven variants of the DSD problem, by carefully considering the requirements of real-life scenarios. For example, the DSD solutions can be used for detecting network communities [19]. However, in a geo-social network, a community often contains a group of users that are not only linked densely, but also have close physical distance [29]. Thus, it would be interesting to study how to incorporate distance into the DSD problem.
O6. While Greedy++ algorithm achieves performs well in practice, theoretically it needs $\Omega(\frac{\Delta(G)}{\rho^*_G \epsilon^2})$ iterations to obtain a $(1+\epsilon)$-approximation ratio solution. Hence, designing an early-stop strategy for Greedy++, similar to those used in CP-based algorithms, is very useful for real situations. Besides, how to devise the Greedy++ like algorithm for the DDS problem is also an exciting research problem.

C ADDITIONAL EXPERIMENTS

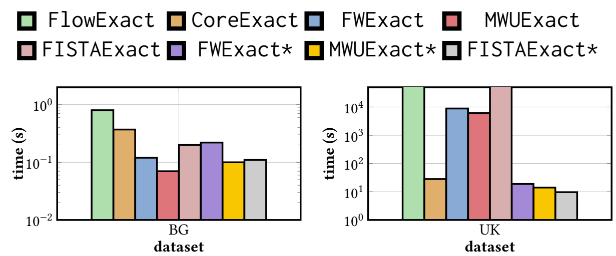
C.1 Additional Datasets.

We present the statistics of the additional four graphs on Table 13, where two undirect graphs, two direct graphs, and from the different domains. They are available on the Stanford Network Analysis Platform, Laboratory of Web Algorithmics, Network Repository, and Konect.



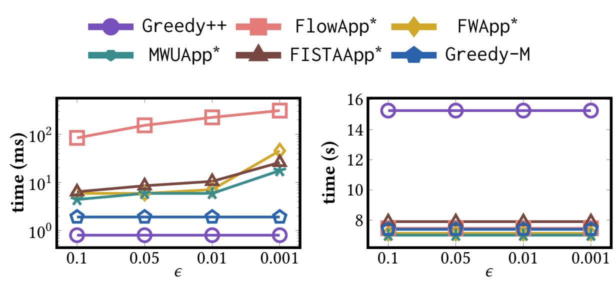
(a) BG (b) UK

Figure 21: Efficiency results of $(1+\epsilon)$-approximation algorithms on undirected graph.



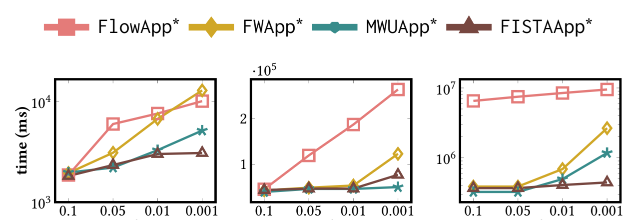
(a) BG (b) UK

Figure 22: Efficiency results of exact algorithms for UDS.



(a) BG (b) UK

Figure 23: Efficiency results of new $(1+\epsilon)$-approximation algorithms on undirected graphs.



(a) LJ (b) WB (c) FS

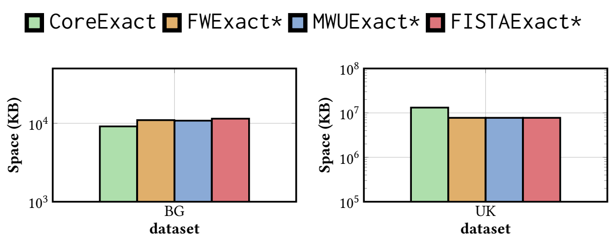
Figure 24: Efficiency results of $(1+\epsilon)$-approximation algorithms for a given ϵ on undirected graphs.

Table 13: Additional four graphs.

Dataset	Category	$\|V\|$	$\|E\|$
bio-SC-GT (BG)	Biological	1,716	31,564
UK-2002 (UK)	Road	18,483,186	261,787,258
maayan-lake (ML)	Foodweb	183	2,494
maayan-figeys (MF)	Metabolic	2,239	6,452

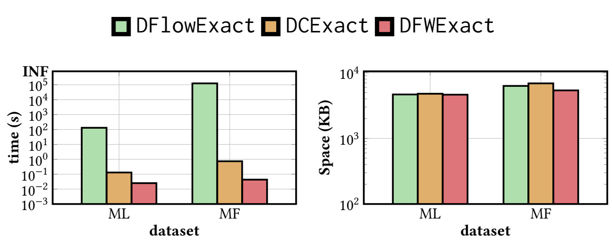
Table 14: Comparison of 2-approximation algorithms on undirected graphs (Red denotes most efficient algorithm, and Blue denotes the best accuracy).

Dataset	BG		UK	
	time	ratio	time	ratio
FlowApp*	21.7 ms	1.550	7.3 s	1
CoreApp	1.3 ms	1.202	6.6 s	1
PKMC	6.6 ms	1.202	7.2 s	1
FWApp	6.0 ms	1.016	80.3 s	1.535
MWUApp	2.8 ms	1.038	101.3 s	1.362
FISTAApp	3.1 ms	1.120	583.2 s	1
Greedy	0.8 ms	1.000	5.9 s	1



(a) BG (b) UK

Figure 25: Memory usage on undirected graphs.



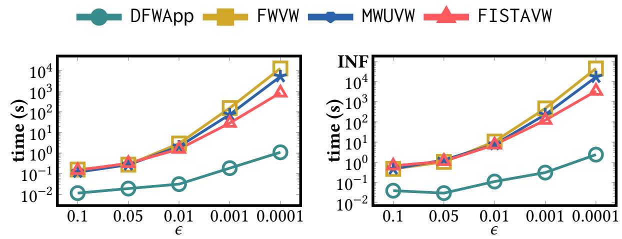
(a) Time (b) Memory

Figure 26: Running time and memory usage of Exact DDS algorithm.

C.2 Additional Experiments on UDS problem

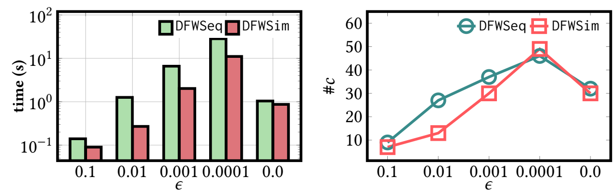
In this section, we first present the additional experimental results on the BG and UK datasets in table 14, and Figure 21 - 25. The results for these two datasets align with experimental results from other datasets presented in Section 6.2.

We report the size of the UDS obtained by various algorithms. The results are reported in Tables 15, 16, and **??** where the edges of the UDS and the ratio of the number of vertices in the UDS (obtained by exact algorithms) to that of the whole graph are also



(a) ML (b) MF

Figure 27: Efficiency results of $(1+\epsilon)$-approximation algorithms on directed graphs.



(a) OF (time) (b) OF (#c)

Figure 28: The effect of update strategies on directed graphs.

presented. Notably, in almost all datasets, except BG and EB, the vertices in the densest subgraph comprise no more than 0.04% of the total vertices in the original graph. This is mainly because the DSD problem aims to maximize the density where vertices with small degrees are excluded.

C.3 Additional Experiments on DDS problem

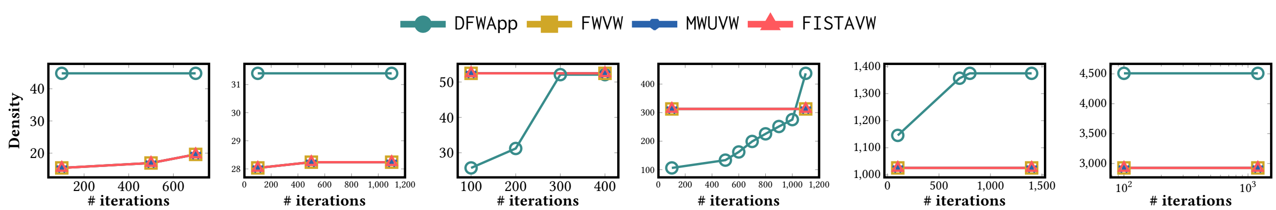
In this section, we present the additional experimental results on the ML and MF datasets in table 18, and Figure 26 - 29. The results for these two datasets align with experimental results from other datasets presented in Section 6.3. Besides, as shown in Figure 29, we can see that, DFW typically takes fewer iterations to obtain the solutions with high densities, which aligns with Lesson 5 (L5) reported in our paper. Moreover, the density of $\mathcal{D}(G)$ returned by VW-based algorithms increases very slowly. This is because they need to enumerate all c values, and there may be many c values corresponding to solutions where the density does not increase. We also report the size of the DDS obtained by various algorithms. The results are reported in Tables 21 - 24 where the edges of the DDS and the ratio of the number of vertices in sets S and T of the DDS (obtained by exact algorithms) to that of the whole graph are also presented. We can make the following observations: (1). For the larger graphs (with more than 100 k vertices), the number of vertices in sets S and T of the DDS comprises no more than 5% of the total vertices in the original graph, which is also a relatively smaller ratio. (2). The numbers of vertices in sets S and T of the DDS are often unbalanced, which inspires us to study how to obtain the densest subgraph on the directed graph with the more balanced vertices distribution on S and T.

Table 15: Results of 2-approx. UDS algorithms

Dataset	BG		EB		DP		YT		LJ		UK		WB		FS	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
FlowApp*	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
CoreApp	99	3,868	187	14,528	114	6,441	425	29,000	377	70,633	944	445,096	1,507	1,134,771	24,528	6,301,881
PKMC	99	3,868	187	14,528	114	6,441	425	29,000	377	70,633	944	445,096	1,507	1,134,771	24,528	6,301,881
FWApp	243	11,124	487	41,686	229	11,661	1,220	94,076	2,229	400,022	1,468	450,598	4,798	3,228,932	98,393	22,425,232
MWUApp	230	10,306	373	34,234	225	8,964	1,193	92,331	2,562	444,562	1,291	446,988	4,564	3,115,806	35,710	9,493,152
FISTApp	309	12,827	380	34,875	133	6,528	1,384	105,563	2,448	320,067	1,384	105,562	4,683	3,461,429	54,158	14,441,496
Greedy	239	12,827	372	34,150	114	6,441	1,219	94,427	386	73,720	944	445,096	1,638	1,316,788	49,730	13,503,584

Table 16: Results of $(1+\epsilon)$-approx. UDS algorithms

Dataset	ϵ	BG		EB		DP		YT		LJ		UK		WB		FS	
		$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
FlowApp*	0.1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	0.01	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	0.001	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	0.0001	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
FWApp	0.1	257	11,942	379	34,790	217	11,656	1,300	100,637	2,380	449,790	944	445,096	3,888	3,157,998	50,015	13,679,449
	0.01	242	11,252	383	35,157	115	6,505	1,202	93,117	1,016	196,294	944	445,096	3,883	3,154,049	49,583	13,561,732
	0.001	242	11,252	375	34,426	115	6,505	1,197	92,730	440	85,146	944	445,096	1,713	1,399,388	49,202	13,457,668
	0.0001	242	11,252	375	34,426	115	6,505	1,197	92,730	440	85,146	944	445,096	1,713	1,399,388	49,202	13,457,668
MWUApp	0.1	257	11,942	387	35,519	114	6,441	1,197	92,730	2,251	426,888	944	445,096	3,857	3,127,340	50,015	13,679,449
	0.01	242	11,252	387	35,519	114	6,441	1,197	92,730	2,251	426,888	944	445,096	3,857	3,127,340	50,015	13,679,449
	0.001	242	11,252	375	34,426	114	6,441	1,197	92,730	440	85,146	944	445,096	1,713	1,399,388	49,165	13,447,549
	0.0001	242	11,252	375	34,426	114	6,441	1,197	92,730	440	85,146	944	445,096	1,713	1,399,388	49,202	13,457,668
FISTAApp	0.1	256	11,896	380	34,884	114	6,441	1,226	94,946	2,249	426,510	944	445,096	3,802	3,066,326	50,015	13,679,449
	0.01	252	11,710	374	34,334	114	6,441	1,206	93,412	1,015	196,101	944	445,096	3,883	3,154,049	49,587	13,562,711
	0.001	242	11,252	375	34,426	114	6,441	1,202	93,117	440	85,146	944	445,096	1,707	1,394,166	49,319	13,489,628
	0.0001	242	11,252	375	34,426	114	6,441	1,197	92,730	440	85,146	944	445,096	1,713	1,399,388	49,193	13,455,207



(a) OF (b) AD (c) AM (d) BA (e) WE (f) SK

Figure 29: The densities of DDS obtained by the $(1+\epsilon)$-approximation algorithms.

Table 18: Comparison of 2-approximation algorithms on directed graphs (Red denotes most efficient algorithm, and Blue denotes the best accuracy).

Dataset	ML		MF	
	time	ratio	time	ratio
DGreedy	130.2 s	1.000	—	—
DFWApp	11.6 ms	1.003	27.5 ms	1
FWVW	13.6 ms	1.099	50.5 ms	1.158
MWUVW	12.5 ms	1.004	81.8 ms	1.029
FISTAVW	20.2 ms	1.003	98.1 ms	1.133
XYCoreApp	4.6 ms	1.197	1.8 ms	1.371
WCoreApp	12.3 ms	1.197	2.0 ms	1.235

Dataset	ϵ	#Vertices		#Edges		Time (s)		Speedup
		Original	Reduced	Original	Reduced	Original	Reduced	
OF	0.01	2,939	464	30,501	12,263	3.5	1.5	2.3
	0.001	2,939	439	30,501	12,473	22.6	9.1	2.5
	0.0001	2,939	426	30,501	12,458	140.3	34.5	4.1
	0	2,939	443	30,501	12,364	3.0	1.0	2.9
AD	0.01	6,541	1,195	51,127	17,137	10.4	3.7	2.8
	0.001	6,541	1,128	51,127	17,541	54.2	21.2	2.6
	0.0001	6,541	1,064	51,127	17,708	296.6	138.8	2.1
	0	6,541	1,161	51,127	16,986	17.4	5.9	2.9
AM	0.01	403,394	11,045	3,387,388	20,058	216.0	1.5	143.6
	0.001	403,394	9,730	3,387,388	15,046	1,791.2	2.9	612.1
	0.0001	403,394	9,066	3,387,388	12,513	7,885.0	17.3	455.7
	0	403,394	10,632	3,387,388	18,483	119.5	1.7	71.1
BA	0.01	2,141,300	137,181	17,794,839	698,422	1,505.5	35.0	43.0
	0.001	2,141,300	137,181	17,794,839	698,422	4,244.5	52.3	81.2
	0.0001	2,141,300	156,486	17,794,839	652,086	13,870.6	156.6	88.6
	0	2,141,300	151,901	17,794,839	711,706	1,319.3	153.5	8.6

Table 19: The effect of graph reduction on directed graphs.

Dataset		$\epsilon = 0.01$		$\epsilon = 0.001$		$\epsilon = 0.0001$		$\epsilon = 0$	
		DFW	DFWReN	DFW	DFWReN	DFW	DFWReN	DFW	DFWReN
AM	#Ratios	6	6	11	10	19	17	7	6
	#Edges	20,058	31,439	15,046	22,476	12,513	17,009	18,483	31,439
	#Iterations	233	267	1,045	1,390	7,974	7,265	200	217
	Product	2.81×10^7	5.03×10^7	1.73×10^8	3.12×10^8	1.90×10^9	2.10×10^9	2.59×10^7	4.09×10^7
	Time (s)	1.0	1.1	2.5	2.6	13.8	11.6	1.3	1.2
BA	#Ratios	16	9	16	9	19	12	16	10
	#Edges	698,422	2,665,311	698,422	2,674,577	652,086	2,099,222	711,706	2,449,856
	#Iterations	112	122	281	322	1,242	2,450	100	100
	Product	1.26×10^9	2.93×10^9	3.14×10^9	7.76×10^9	1.54×10^{10}	6.17×10^{10}	1.14×10^9	2.45×10^9
	Time (s)	28.2	56.4	38.5	61.7	117.7	166.3	119.7	133.0

Table 20: Effectiveness of graph reduction technique on directed graphs.

Dataset	$\lvert V \rvert$	# of vertices S of DDS	ratio of S	# of vertices T of DDS	ratio of T	# of edges in DDS
maayan-lake (ML)	183	70	38.25%	32	17.49%	1,303
maayan-figeys (MF)	2,239	7	0.31%	382	17.06%	1,256
Openflights (OF)	2,939	189	6.43%	187	6.36%	8,432
Advogato (AD)	6,541	453	6.93%	195	2.98%	9,416
Amazon (AM)	403,394	4,944	1.23%	2	0.0005%	5,238
Baidu-zhishi (BA)	2,141,300	99,040	4.63%	2	0.00009%	194,802
Wiki-en (WE)	13,593,032	27,101	0.20%	70	0.0005%	1,893,778
SK-2005 (SK)	50,636,154	4,510	0.01%	4,515	0.01%	20,349,639

Table 21: Comparing the sizes of DDS and entire directed graph.

Dataset	ML		MF		OF		AD		AM		BA		WE		SK	
	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$
`DGreedy`	$\frac{70}{32}$	1,303	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–
`DFWApp`	$\frac{77}{32}$	1,362	$\frac{7}{382}$	1,256	$\frac{193}{193}$	8,652	$\frac{418}{393}$	12,732	$\frac{2,751}{1}$	2,751	$\frac{95,762}{2}$	191,524	$\frac{27,101}{70}$	1,893,778	$\frac{4,510}{4,515}$	20,343,472
`XYCoreApp`	$\frac{23}{23}$	529	$\frac{1}{314}$	314	$\frac{37}{35}$	1,135	$\frac{1}{786}$	786	$\frac{2,751}{1}$	2,751	$\frac{95,762}{2}$	191,524	$\frac{27,040}{69}$	1,865,760	$\frac{4,504}{4,515}$	20,331,057
`WCoreApp`	$\frac{23}{23}$	529	$\frac{4}{141}$	467	$\frac{37}{37}$	1,189	$\frac{1}{786}$	786	$\frac{2,751}{1}$	2,751	$\frac{95,762}{2}$	191,524	$\frac{27,040}{69}$	1,865,760	$\frac{4,504}{4,515}$	20,331,057

Table 22: Results of 2-approx. DDS algorithms.

Dataset	ϵ	ML		MF		OF		AD		AM		BA		WE		SK	
		$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$
`DFWApp`	0.1	$\frac{70}{33}$	1,323	$\frac{7}{382}$	1,256	$\frac{193}{193}$	8,652	$\frac{453}{195}$	9,416	$\frac{2,751}{1}$	2,751	$\frac{95,762}{2}$	191,524	$\frac{27,101}{70}$	1,893,778	$\frac{8,932}{8,942}$	39,914,393
	0.01	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{189}{187}$	8,432	$\frac{453}{195}$	9,416	$\frac{4,944}{2}$	5,238	$\frac{99,040}{2}$	194,802	$\frac{27,101}{70}$	1,893,778	$\frac{4,510}{4,515}$	20,349,639
	0.001	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{189}{187}$	8,432	$\frac{453}{195}$	9,416	$\frac{4,944}{2}$	5,238	$\frac{99,040}{2}$	194,802	$\frac{27,101}{70}$	1,893,778	$\frac{4,510}{4,515}$	20,349,639
	0.0001	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{189}{187}$	8,432	$\frac{453}{195}$	9,416	$\frac{4,944}{2}$	5,238	$\frac{99,040}{2}$	194,802	$\frac{27,101}{70}$	1,893,778	$\frac{4,510}{4,515}$	20,349,639

Table 23: Results of $(1+\epsilon)$-approx. DDS algorithms.

Dataset	ML		MF		OF		AD		AM		BA		WE		SK	
	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$	$\frac{\lvert S\rvert}{\lvert T\rvert}$	$\lvert E\rvert$
`DFlowExact`	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–	$\frac{-}{-}$	–
`DDCExact`	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{189}{187}$	8,432	$\frac{453}{195}$	9,416	$\frac{4,944}{2}$	5,238	$\frac{99,040}{2}$	194,802	$\frac{-}{-}$	–	$\frac{-}{-}$	–
`DFWExact`	$\frac{70}{32}$	1,303	$\frac{7}{382}$	1,256	$\frac{189}{187}$	8,432	$\frac{453}{195}$	9,416	$\frac{4,944}{2}$	5,238	$\frac{99,040}{2}$	194,802	$\frac{27,101}{70}$	1,893,778	$\frac{4,510}{4,515}$	20,349,639

Table 24: Results of exact DDS algorithms.