Finding Locally Densest Subgraphs: A Convex Programming Approach

Chenhao Ma
The University of Hong Kong
Hong Kong SAR, China
chma2@cs.hku.hk

Reynold Cheng
Guangdong–Hong Kong-Macau Joint Laboratory for Smart Cities
HKU Musketeers Foundation Institute of Data Science
The University of Hong Kong
Hong Kong SAR, China
ckcheng@cs.hku.hk

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

Xiaolin Han*
The University of Hong Kong
Hong Kong SAR, China
xlhan@cs.hku.hk

ABSTRACT

Finding the densest subgraph (DS) from a graph is a fundamental problem in graph databases. The DS obtained, which reveals closely related entities, has been found to be useful in various application domains such as e-commerce, social science, and biology. However, in a big graph that contains billions of edges, it is desirable to find more than one subgraph cluster that are not necessarily the densest, yet they reveal closely-related vertices. In this paper, we study the locally densest subgraph (LDS), a recently-proposed variant of DS. An LDS is a subgraph which is the densest among the "local neighbors". Given a graph G, a number of LDS's can be returned, which reflect different dense regions of G and thus give more information than DS. The existing LDS solution suffers from low efficiency. We thus develop a convex-programming-based solution that enables powerful pruning. Extensive experiments on seven real large graph datasets show that our proposed algorithm is up to four orders of magnitude faster than the state-of-the-art.

PVLDB Reference Format:
Chenhao Ma, Reynold Cheng, Laks V.S. Lakshmanan, and Xiaolin Han. Finding Locally Densest Subgraphs: A Convex Programming Approach. PVLDB, 15(11): 2719 - 2732, 2022.
doi:10.14778/3551793.3551826

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chenhao-ma/LDScvx.

1 INTRODUCTION

In modern systems and platforms that manage intricate relationship among objects, graphs have been widely used to model such relationship information [13, 18, 25, 27–31, 37, 38]. For example, the Facebook friendship network can be treated as a graph by mapping users to vertices and friendships among users to edges connecting vertices [13]. Figure 1 depicts a graph of friendship, where a and f have an edge meaning that they are friends of each other. In biology, graphs can be used to capture complex interactions among different proteins [51] and relationships in genomic DNA [23].

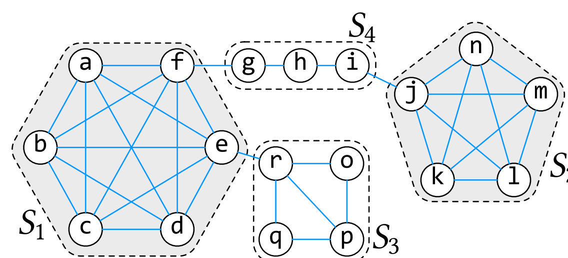

Figure 1: An undirected graph, G

Lying at the core of large scale graph data mining, the densest subgraph (DS) problem [5, 21, 26, 45] is about the finding of a "dense" subgraph from a graph G with n vertices and m edges. For example, the DS of Figure 1 is the subgraph induced by vertex subset S_1, because its density, i.e., the average number of edges over the number of vertices in the subgraph, is the highest among all possible subgraphs of G. The DS has been found useful to various application domains [56]. For example, dense subgraphs can be used to detect communities [12, 55] and discover fake followers [32] in social networks. In biology, the DS found can be used to identify regulatory motifs in genomic DNA [23] and find complex patterns in gene annotation graphs [47]. In graph databases, a DS is used to construct index structures for supporting reachability and distance queries [33]. In system optimization, DS plays an important role in social piggybacking [25, 56], which improves the throughput of social networking systems (e.g., Twitter).

LDS model. Most existing studies [5, 10, 21, 26] on the DS problem focus on finding the densest subgraph. In fact, it is not uncommon to find more than one "dense subgraphs". For example, in community detection, it is interesting to explore multiple communities in a social network, even if not all communities have the highest density. Motivated by this, Qin et al. [45] proposed the locally densest subgraph (LDS) model [45, 48, 56], which identifies LDS's of the graph. An LDS needs to be *dense* and *compact*. Conceptually, an LDS is a subgraph with the highest density in its vicinity. Moreover, LDS is "compact", in the sense that any subset of its vertices is highly connected to each other. (We will discuss the formal definition of LDS in Section 3.) For Figure 1, the two subgraphs induced by vertex subsets S_1 and S_2, denoted as $G[S_1]$

*Xiaolin Han is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551826

and $G[S_2]$ respectively, are two LDS's of G. On the other hand, the subgraph $G[S_1 \cup S_3]$ is not an LDS.

The LDS problem has three important properties [45]:

• The LDS is *parameter-free*. As we will explain later, the two LDS's $G[S_1]$ and $G[S_2]$ can be obtained from G in Figure 1 without setting density threshold or other parameters.

• For any pair of LDS's on a given graph, they do not have common vertices (i.e., *disjoint*). In the above example, the two LDS's $G[S_1]$ and $G[S_2]$ are disjoint. This allows us to identify all the non-overlapping "dense regions" of a graph.

• The set of subgraphs found by the LDS problem is a superset of the subgraph found by the DS problem. For example, DS only identifies $G[S_1]$ in Figure 1, with a density of 2.5. However, LDS returns $\{G[S_1], G[S_2]\}$. Notice that $G[S_2]$, which has a density of 2, does not overlap with $G[S_1]$.

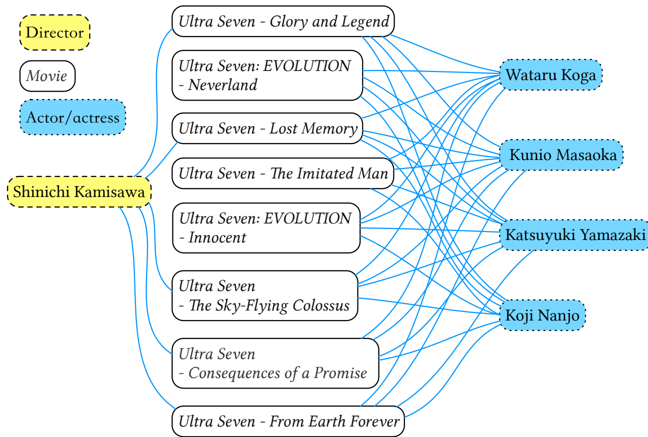

Figure 2: An LDS about "Ultraman".

We have conducted a case study on a large movie graph provided by TCL, an electronic product provider in China. We found that the LDS's found are about different topics (e.g., western films, Chinese martial fiction, Danish comedies). Figure 2 shows an LDS, with the third highest density, about the Japanese sci-fi series "Ultraman". The DS model can only find a subgraph about western films. The case study shows that the LDS model can find representative dense subgraphs of a graph.

Prior work. Based on the LDS model, Qin et al. proposed a max-flow-based LDS solution with pruning techniques built on k-core, named `LDSflow` [45]. Here k-core [50] is a cohesive subgraph, which requires each vertex to have at least k neighbors in the subgraph. To find LDS's, `LDSflow` adopts a prune-and-verify framework. Specifically, the pruning bounds for vertices based on k-core are derived, which are then used to prune vertices. The LDS's are then obtained by verifying the remaining vertices through max-flow computation on the flow network. The main problem of `LDSflow` is that it may not scale well on large graphs. For example, `LDSflow` needs around 17 hours to output 15 LDS's with the highest densities from a graph with around 3 million edges.

Contributions. We have developed an efficient and scalable LDS solution, as detailed below:

(1) *Propose the compact number.* Given a vertex v, we define the *compact number* of v, which depicts the degree of compactness of the most compact subgraph containing v. We further use compact numbers to link the LDS problem and a convex program for the DS problem theoretically, leveraging the fact that vertices in an LDS share the same compact number, and compact numbers can be computed by solving the convex program.

(2) *An efficient convex-programming-based algorithm with elegant pruning techniques.* We propose a prune-and-verify algorithm based on the convex programming, named `LDScvx`. For pruning, we show that an iterative Frank-Wolfe algorithm [22] can provide the tight upper and lower bounds for compact numbers, which can be used to prune vertices not contained by any LDS from the graph. For verification, `LDSflow` [45] performs min-cut computation based on a specific k-core of G. In `LDScvx`, we only need to compute the min-cut based on a smaller subgraph of the k-core by exploiting the upper and lower bounds of compact numbers.

(3) *Extensive experiments.* We have experimentally compared our LDS algorithm with the state-of-the-art algorithm on seven real large datasets, with sizes up to 1.6 billion edges. Our results show that `LDScvx` is up to four orders of magnitude faster than the state-of-the-art. A case study on a large movie graph has also been performed.

Outline. The rest of the paper is organized as follows. We review the related work in Section 2. In Section 3, we formally present the LDS problem. Section 4 defines the compact number, and discuss its relationship with LDS and convex programming. We present our LDS algorithm in Section 5 and empirical results in Section 6. Section 7 concludes the paper.

2 RELATED WORK

The densest subgraph is regarded as one kind of cohesive subgraph. Tsourakakis and Chen [56] provides a comprehensive study of techniques and applications of the DS problem. Other related topics include k-core [50], k-truss [57], clique, and quasi-clique [14]. More details on them can be found in [19, 20]. In the following, we focus on densest subgraph discovery and its variants.

Densest subgraph discovery (DS). Goldberg [26] introduced the densest subgraph (DS) problem on undirected graphs, which aims to find the subgraph whose edge-density is the highest among all the subgraphs where the edge-density of a graph $G = (V, E)$ is defined as $\frac{|E|}{|V|}$. To solve the DS problem, Goldberg [26] proposed an exact algorithm based on flow network via solving $O(\log(n))$ min-cut problems. Later, Fang et al. [21] improved the efficiency of the flow-based exact algorithm by locating the DS in a specific k-core. Exact algorithms can process small graphs reasonably, but they cannot scale well to large graphs. To remedy this issue, several approximation algorithms [5, 8, 10, 21] have been developed.

Kannan and Vinay [34] extended the DS problem definition to directed graphs. Charikar [10] developed an exact polynomial-time algorithm to find the directed densest subgraph via $O(n^2)$ linear programs. Khuller and Saha [35] presented a max-flow-based exact algorithm. Ma et al. [40–42] improved the max-flow-based algorithm by introducing the notion of $[x, y]$-core and exploiting a divide-and-conquer strategy. To further boost the efficiency for

large-scale graphs, approximation algorithms [10, 35, 39, 40, 49] for the directed DS problem are also developed.

Further, there are some variants of the DS problem focusing on different aspects. Asahiro et al. [3] studied the DS problem with constraints on the size of the DS. However, the size constraints (e.g., at least k vertices or at most k vertices to be included) make the DS problem NP-hard [3]. Hence, Andersen and Chellapilla [2], Khuller and Saha [35], and Chekuri et al. [11] studied efficient approximation algorithms on the variants of the size-constrained DS. Tsourakakis extended the notion of density based on edges to k-clique-density, and studied the DS problem with k-clique-density [43, 52, 54]. Chang and Qiao [9] proposed a novel and efficient index to report all minimal DS's and enumerate all DS's, where the minimal DS is strictly denser than all of its proper subgraphs. Tatti et al. [53] and Danisch et al. [15] studied the density-friendly decomposition problem, which decomposes a graph into a chain of subgraphs, where each subgraph is nested within the next one, and the inner one is denser than the outer ones. Galbrun et al. [24] and Dondi et al. [16, 17] studied the densest subgraphs with overlaps.

Locally densest subgraph (LDS). Qin et al. [45] proposed a new DS model, named locally densest subgraph (LDS). Based on this model, users can identify all the locally densest regions of a graph. Qin et al. have shown that such subgraphs cannot be found by other DS models theoretically and empirically. Compared to the original DS model, Qin et al. [45] showed that the subgraphs provided by the LDS model best represent different local dense regions of the graph, while the DS model only provides the DS. Furthermore, they compared the LDS model with a straightforward greedy approach based on the DS, which finds the DS [26] at a time, removes it from the graph, and repeats the process for k times. Nevertheless, this greedy approach has several shortcomings [45]: (1) The top-k results may not fully reflect the top-k densest regions. Especially when the graph has a vast dense region, subgraphs in other dense regions may have a low chance to appear. (2) A subgraph returned by the greedy approach can be partial and contained by a better subgraph. (3) This greedy approach is essentially a heuristic and does not allow for a formal characterization of the result. The above claims are also verified by the experimental results in [45]. With the LDS model justified, Qin et al. [45] proposed a max-flow-based algorithm, `LDSflow`, to find the top-k LDS's from a graph. However, we found that `LDSflow` [45] does not scale to large graphs because `LDSflow` needs to run the max-flow algorithm on the graph several times to find an LDS candidate and verify it, and the max-flow computation can be quite time-consuming. For example, the state-of-the-art max-flow algorithm is proposed by Orlin with time complexity of $O(nm)$ [44]. In this paper, we propose an efficient and scalable LDS algorithm for finding the top-k LDS's from large graphs. We note that Samusevich et al. [48] extended the LDS model from edge-based density to triangle-based density. We focus on edge-based density, and leave triangle density based LDS for future work.

3 PRELIMINARIES

This section formally defines the locally densest subgraph problem (LDS problem). Table 1 lists the notations used in this paper.

Table 1: Notations and meanings.

Notation	Meaning
$G = (V, E)$	a graph with vertex set V and edge set E
$G[S]$	the subgraph induced by S
$d_G(u)$	the degree of a vertex u in G
$\mathsf{density}(G)$	the density of graph G, i.e., $\frac{\lvert E\rvert}{\lvert V\rvert}$
$\phi(u), \phi_{G'}(u)$	compact number of u in G and G' respectively
$\overline{\phi}(u)$	the upper bound of $\phi(u)$
$\underline{\phi}(u)$	the lower bound of $\phi(u)$
$\mathsf{core}_G(u)$	the core number of u in G
$\mathsf{CP}(G)$	the convex program of G for densest subgraph
$\boldsymbol{\alpha}$	the weights distributed from edges to vertices
$\boldsymbol{r}$	the weights received by each vertex

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. For each vertex $u \in G$, we use $d_G(u)$ to represent its degree in G. Given a subset $S \subseteq V$, $E(S)$ denotes the set of edges induced by S, i.e., $E(S) = E \cap (S \times S)$. Hence, the subgraph induced by S is denoted by $G[S] = (S, E(S))$. Following the classic graph density definition [4, 5, 10, 26, 45], the density of a graph $G = (V, E)$, denoted by $\mathsf{density}(G)$, is defined as:

$$\mathsf{density}(G) = \frac{|E|}{|V|}. \tag{1}$$

Based on the density definition, the densest subgraph problem is to find the subgraph $G[S]$ of G such that $\mathsf{density}(G[S])$ is maximized.

Densest subgraph discovery is widely applied in many graph mining tasks (e.g., [23, 32, 40, 42, 47]). However, as pointed out by [45], it is usually not sufficient to find one dense subgraph, in many applications such as community detection [18]. Qin et al. [45] proposed a locally densest subgraph model to provide multiple dense subgraphs, which are dense and compact. Equation (1) gives the density of a subgraph. For compactness, Definition 3.1 defines what is a ρ-compact subgraph, which comes from the intuition that a graph is compact if any subset of vertices is highly connected to others in the graph [45].

Definition 3.1 (ρ-compact [45]). A graph $G = (V, E)$ is ρ-compact if and only if G is connected, and removing any subset of vertices $S \subseteq V$ will result in the removal of at least $\rho \times |S|$ edges in G, where ρ is a non-negative real number.

Definition 3.2 (Maximal ρ-compact subgraph [45]). A ρ-compact subgraph $G[S]$ of G is a maximal ρ-compact subgraph of G if and only if there does not exist a supergraph $G[S']$ of $G[S]$ ($S' \neq S$) in G such that $G[S']$ is also ρ-compact.

Remark. The locally dense subgraph in [53] is similar to the maximal ρ-compact subgraph. But they are different because the ρ-compact subgraph needs to be connected.

Definition 3.3 (Locally densest subgraph [45]). A subgraph $G[S]$ of G is a locally densest subgraph (LDS) of G if and only if $G[S]$ is a maximal $\mathsf{density}(G[S])$-compact subgraph in G.

From Definition 3.3, we can find that (1) the definition itself is parameter-free; (2) an LDS is compact; (3) any supergraph of an LDS cannot be more compact than the LDS itself; (4) any subgraph $G[S']$

of an LDS $G[S]$ cannot be denser than the LDS itself [45]. These properties show that an LDS is indeed a *locally densest* subgraph. The third one comes from that an LDS $G[S]$ is a maximal ρ-compact subgraph. The last one can be proven by contradiction. Suppose that $G[S']$ has a density satisfying $\text{density}(G[S']) > \text{density}(G[S])$. If we remove vertex set $U = S \setminus S'$ from $G[S]$, the number of edges removed is $\text{density}(G[S]) \times |S| - \text{density}(G[S']) \times |S'| < \text{density}(G[S]) \times (|S| - |S'|) = \text{density}(G[S]) \times |U|$, which contradicts that $G[S]$ is $\text{density}(G[S])$-compact. We further illustrate the LDS definition with the following example.

Example 3.4 (LDS). Consider the graph G shown in Figure 1. The subgraph $G[S_1]$ with density $\frac{5}{2}$ is a maximal $\frac{5}{2}$-compact subgraph. Hence, $G[S_1]$ is an LDS. Similarly, $G[S_2]$ with density 2 is also an LDS as it is a maximal 2-compact subgraph. The subgraph $G[S_3]$ with density $\frac{5}{4}$ is a $\frac{5}{4}$-compact subgraph. But $G[S_3]$ is not an LDS because it is contained in $G[S_1 \cup S_3]$ which is $\frac{3}{2}$-compact (and also $\frac{5}{4}$-compact). $G[S_1 \cup S_3]$ is also not an LDS, because its density is $\frac{21}{10}$ but it is not a $\frac{21}{10}$-compact subgraph. The compactness of $G[S_1 \cup S_3]$ is $\frac{3}{2} = \frac{6}{4}$, because removing S_3 from $G[S_1 \cup S_3]$ will result in removing of 6 edges. ◻

The LDS has a useful property that all the LDS's in a graph G are pairwise disjoint.

Lemma 3.5 (Disjoint property [45]). *Suppose $G[S]$ and $G[S']$ are two LDS's in G, we have $S \cap S' = \emptyset$.*

According to Lemma 3.5, the LDS model can be used to find all dense regions of a graph. However, most real-world applications (e.g., community detection) usually require finding the top-k dense regions of a graph [45]. Hence, we focus on finding the top-k LDS's with the largest densities following [45].

***Problem* 1** (LDS problem [45])**:** Given a graph G and an integer k, the LDS problem is to compute the top-k LDS's with the largest density in G.

4 FROM COMPACTNESS TO CP

In this section, we first propose a new concept named *compact number*, inspired by the core number related to k-core [50], which is a cohesive subgraph model. Then, we present some interesting properties of LDS from the perspective of compact numbers. Afterward, we show that the compact numbers can be computed via a convex programming (CP) formulation of the densest subgraph problem. Hence, the compact number acts as a bridge between the LDS problem and the CP formulation, as shown in Figure 3.

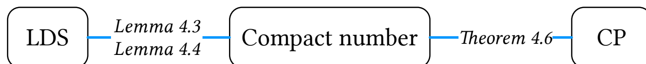

Figure 3: Relation among LDS, Compact number, and CP.

4.1 Compact Number and LDS

Inspired by the core number of k-core [50], we define the *compact number* of each vertex in graph G with respect to ρ-compact subgraphs of G.

Definition 4.1 (Compact number). Given a graph $G = (V, E)$, the compact number of each vertex $u \in V$, denoted by $\phi(u)$, is the largest ρ such that u is contained in a ρ-compact subgraph of G.

We use $\overline{\phi}(u)$ and $\underline{\phi}(u)$ to denote the upper and lower bounds of $\phi(u)$ in G, respectively. Besides, $\overline{\phi}_{G'}(u)$ denotes the upper bound of $\phi_{G'}(u)$ in G', G' is a subgraph of G.

Example 4.2 (Compact number). Consider vertex q of G in Figure 1. The compact number of q is $\frac{3}{2}$, i.e., $\phi(\mathsf{q}) = \frac{3}{2}$, because $G[S_1 \cup S_3]$ is a $\frac{3}{2}$-compact subgraph containing q and there is no other subgraph containing q with a larger ρ. Removing S_3 from $G[S_1 \cup S_3]$ will remove 6 edges, so $G[S_1 \cup S_3]$ is $\frac{3}{2}$-compact.

Next, we discuss the relationship between the LDS and the compact numbers of vertices within or adjacent to the LDS via the following lemmas.

Lemma 4.3. *Given an LDS $G[S]$ in G, $\forall u \in S$, we have $\phi(u) = \text{density}(G[S])$.*

Proof. As $G[S]$ is a maximal $\text{density}(G[S])$-compact subgraph, for each $u \in S$, there exists no other subgraph $G[S']$ containing u such that $G[S']$ is a ρ-compact subgraph with $\rho > \text{density}(G[S])$. We prove the claim by contradiction. Suppose $G[S']$ is a ρ-compact subgraph with $\rho > \text{density}(G[S])$ and $u \in S'$, we have $\text{density}(S') \geq \rho > \text{density}(G[S])$. First $S' \subseteq S$, because $G[S]$ is a maximal $\text{density}(G[S])$-compact subgraph and $S' \cap S \neq \emptyset$. If we remove $U = S \setminus S'$ from $G[S]$, the number of edges removed is $|E(S)| - |E(S')| = \text{density}(G[S]) \times |S| - \text{density}(G[S']) \times |S'| < \text{density}(G[S]) \times (|S| - |S'|) = \text{density}(G[S]) \times |U|$. This contradicts that $G[S]$ is $\text{density}(G[S])$-compact. Hence, $\text{density}(G[S])$ is the compact number of all vertices in S. ◻

Lemma 4.4. *Given an LDS $G[S]$ in G, $\forall (u, v) \in E$, if $u \in S$ and $v \in V \setminus S$, we have $\phi(u) > \phi(v)$.*

Proof. The lemma follows from the LDS definition. Suppose $\exists (u, v) \in E, u \in S, v \in V \setminus S, \phi(u) \leq \phi(v)$, which contradicts that $G[S]$ is a maximal $\text{density}(G[S])$-compact subgraph. ◻

Lemmas 4.3 and 4.4 indicate that the compact numbers of all vertices in an LDS $G[S]$ are exactly $\text{density}(G[S])$ and the compact numbers of all vertices adjacent to vertices in $G[S]$ but not in $G[S]$ are less than $\text{density}(G[S])$, respectively. We further illustrate the two lemmas via the following example.

Example 4.5. Consider the LDS $G[S_1]$ of G in Figure 1. We can see that $\forall u \in S_1$, $\phi(u) = \text{density}(G[S_1]) = \frac{5}{2}$, which fulfills Lemma 4.3. For vertices locating outside $G[S_1]$ but adjacent to some vertices in S_1, i.e., g, r, and q, their compact numbers satisfies Lemma 4.4: $\phi(\mathsf{g}) = \frac{4}{3} < \frac{5}{2}$, $\phi(\mathsf{r}) = \phi(\mathsf{q}) = 2 < \frac{5}{2}$.

According to Lemmas 4.3 and 4.4, compact numbers are powerful to extract and verify LDS's from a graph G. If the compact numbers of all vertices are ready, we can partition the graphs into different subgraphs, where the vertices within the same subgraph share the same compact number and are connected, based on Lemma 4.3. Then, Lemma 4.4 can be used to select LDS's from all subgraphs. Hence, the efficient computation of compact numbers is a key issue. To tackle this issue, we show that compact numbers can be obtained by solving a convex program in the next subsection.

4.2 Compact Number and CP

In this subsection, we first review the convex program (CP) for densest subgraphs by Danisch et al. [15]. Next, we theoretically prove that compact numbers can be obtained by solving the convex program.

$$
\begin{aligned}
\mathsf{CP}(G) \qquad & \min \sum_{u \in V} r_u^2 & \\
& r_u = \sum_{(u,v)\in E} \alpha_{u,v}, & \forall u \in V \\
& \alpha_{u,v} + \alpha_{v,u} \geq 1, & \forall (u,v) \in E \\
& \alpha_{u,v}, \alpha_{v,u} \geq 0. & \forall (u,v) \in E
\end{aligned}
\tag{2}
$$

The intuition of $\mathsf{CP}(G)$ is that each edge $(u,v) \in E$ tries to distribute its weight, i.e., 1, between its two endpoints u and v such that the weight sum received by the vertices are as even as possible. Because in the DS $G[S]$ of G, it is possible to distribute all edge weights such that the weight sum received by each vertex in S is exactly $\mathsf{density}(G[S]) = \frac{|E(S)|}{|S|}$. Following the intuition, $\alpha_{u,v}$ in $\mathsf{CP}(G)$ indicates the weight assigned to u from edge (u,v), and r_u is the weight sum received by u from its adjacent edges. Danisch et al. [15] used r and α of $\mathsf{CP}(G)$ to tentatively decompose the graph into a chain of subgraphs and applied max-flow to fine-grain and confirm the partitions such that each subgraph is nested within the next one with densities in descending order.

Before proving the compact numbers can be obtained via solving $\mathsf{CP}(G)$, we briefly review the Frank-Wolfe-based iterative algorithm proposed by Danisch et al. [15] for optimizing $\mathsf{CP}(G)$. `Frank-Wolfe` (Algorithm 1) outlines the steps to optimize $\mathsf{CP}(G)$. `Frank-Wolfe` first initializes $\boldsymbol{\alpha}$ and $\boldsymbol{r}$ (lines 2–3). Then, in each iteration, each edge $(u,v) \in E$ attempts to distribute its weight, i.e., 1, to the endpoint with a smaller r value (lines 4–10).

Algorithm 1: Frank-Wolfe-based algorithm [15]

```
1  Function Frank-Wolfe(G = (V, E), N ∈ Z+):
2      foreach (u, v) ∈ E do α(0)_{u,v} ← 1/2, α(0)_{v,u} ← 1/2;
3      foreach u ∈ V do r(0)_u ← Σ_{(u,v)∈E} α(0)_{u,v};
4      for i = 1, ..., N do
5          γ_i = 2/(i+2);
6          foreach (u, v) ∈ E do
7              if r(i-1)_u < r(i-1)_v then α̂_{u,v} ← 1, α̂_{v,u} ← 0;
8              else α̂_{u,v} ← 0, α̂_{v,u} ← 1;
9          α(i) ← (1 − γ_i) · α(i-1) + γ_i * α̂;
10         foreach u ∈ V do r(i)_u ← Σ_{(u,v)∈E} α(i)_{u,v};
11     return (r(i), α(i))
```

Next, we prove that the compact numbers can be extracted from the optimal solution of $\mathsf{CP}(G)$.

Theorem 4.6. *Suppose $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ is an optimal solution of $\mathsf{CP}(G)$. Then, each r_u^* in $\boldsymbol{r}^*$ is exactly the compact number of u, i.e., $\forall u \in V, r_u^* = \phi(u)$.*

Proof. For a vertex $u \in V$, let $X = \{v \in V | r_v^* > r_u^*\}$, $Y = \{v \in V | r_v^* = r_u^*\}$, and $Z = \{v \in V | r_v^* < r_u^*\}$. Clearly, $u \in Y$.

We first prove $G[X \cup Y]$ is a r_u^*-compact subgraph. Removing Y from $G[X \cup Y]$ will result in the removal of $r_u^* \times |Y|$ edges in $G[X \cup Y]$. The optimality of $\boldsymbol{r}^*$ implies that

(1) $\forall (x,y) \in E \cap (X \times Y), r_x^* > r_y^*$ and $\alpha_{x,y} = 0$;
(2) $\forall (y,z) \in E \cap (Y \times Z), r_y^* > r_z^*$ and $\alpha_{y,z} = 0$.

Otherwise, suppose $\exists (x,y) \in E \cap (X \times Y)$ such that $\alpha_{x,y} > 0$ without loss of generality. There exists $r_x^* - r_y^* > \epsilon > 0$ such that we can reduce $\alpha_{x,y}$ and increase $\alpha_{y,x}$ by ϵ, respectively. Hence, r_x^* is reduced and r_y^* is increased by ϵ, respectively. After such modification, the objective function be decreased by $2\epsilon(r_x^* - r_y^* - \epsilon)$, which contradicts the optimality of r^*. Hence, $r_u^* \times |Y| = \sum_{(y,x) \in E \wedge y \in Y} \alpha_{y,x} = |((X \times Y) \cup (Y \times Y)) \cap E|$, which is exactly the number of edges to be removed when removing Y from $G[X \cup Y]$. Besides, removing any $Q \subseteq X \cup Y$ from $G[X \cup Y]$ will result in the removal of at least $r_u^* \times |Q|$ edges, because $\sum_{(s,t) \in E(X \cup Y) \wedge s \in Q} 1 \geq \sum_{(s,t) \in E \wedge s \in Q} \alpha_{s,t} \geq r_u^* \times |Q|$, where the second inequality follows from the first condition of $\mathsf{CP}(G)$ (Equation (2)). Hence, $G[X \cup Y]$ is a r_u^*-compact subgraph.

For any other subgraph $G[S]$ containing u, $G[S]$ is a ϕ-compact subgraph, where $\phi \leq r_u^*$. Clearly, $S \cap (Y \cup Z) \neq \emptyset$. Removing $S \cap (Y \cup Z)$ from $G[S]$ will result in the removal of no more than $r_u^* \times |S \cap (Y \cup Z)|$ edges, which can be proved by contradiction analogously. ◻

Theorem 4.6 shows that given an optimal solution $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ to $\mathsf{CP}(G)$, the weight received by each vertex $\boldsymbol{r}_u^*$ is exactly the compact number of u. Example 4.7 further depicts Theorem 4.6 concretely.

Example 4.7. Consider the convex program $\mathsf{CP}(G)$ for G in Figure 1. We list the optimal solution $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ values in Table 2. Some values of $\boldsymbol{\alpha}^*$ are omitted, as they can be inferred from other r_u^* and $\alpha_{u,v}^*$ values. For each $u \in V$, r_u^* is exactly the compact number of u, $\phi(u)$.

Table 2: $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ to $\mathsf{CP}(G)$ for G in Figure 1.

Vertices	r_u^*	Edges	$\alpha_{u,v}^*$
a, b, c, d, e, f $\in S_1$	$\frac{5}{2}$	$(u,v) \in E(S_1) \cup E(S_2)$	$\frac{1}{2}$
j, k, l, m, n $\in S_2$	2	(g, f), (i, j), (r, e)	1
o, p, q, r $\in S_3$	$\frac{3}{2}$	(g, h), (i, h)	$\frac{1}{3}$
g, h, i	$\frac{4}{3}$	...	

According to Corollary 4.9 of [15], it may need many iterations for `Frank-Wolfe` to obtain the optimal $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$. Fortunately, we found that the approximate solution provided by `Frank-Wolfe` already helps prune the vertices not contained in LDS's and extract LDS's, which will be discussed in the next section.

5 OUR LDS ALGORITHM

In this section, we introduce our convex programming based LDS algorithm, named `LDScvx`. Figure 4 presents the workflow of `LDScvx`. First, we compute an approximate solution $(\boldsymbol{r}, \boldsymbol{\alpha})$ via `Frank-Wolfe`; next, we extract stable groups, which can be used to bound the compact numbers, from G based on $(\boldsymbol{r}, \boldsymbol{\alpha})$ via `ExtractSG`; afterward, we prune invalid vertices according to their compact numbers and generate LDS candidates via `Pruning`; finally, we verify the LDS candidates via `IsLDS`. If the verification failed, we repeat the above

process to provide higher-quality $(\boldsymbol{r}, \boldsymbol{\alpha})$ and compact number estimation.

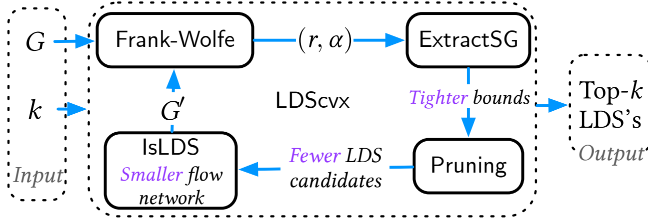

Figure 4: Algorithm workflow.

Comparison with `LDSflow`. `LDSflow` [45] also follows a prune-and-verify framework. Specifically, `LDSflow` prunes vertices, according to pruning bounds that are derived via k-cores. The LDS's are obtained by verifying the remaining vertices via max-flow computation. Compared to `LDSflow`, our `LDScvx` provides *tigher* bounds via convex programming, which allows more vertices to be pruned. Hence, *fewer* LDS candidates need to be verified. Further, we construct *smaller* flow networks to verify LDS's via exploiting lower bounds of compact numbers. The above techniques will be introduced in detail and empirically evaluated.

5.1 Extract Stable Groups

In this subsection, we first introduce a new concept *stable group*, which can be used to provide upper and lower bounds of compact numbers, inspired by the stable subset in [15]. Then, we discuss how to extract stable groups from the approximate solution $(\boldsymbol{r}, \boldsymbol{\alpha})$ provided by `Frank-Wolfe`.

Definition 5.1 (Stable group). Given a feasible solution $(\boldsymbol{r}, \boldsymbol{\alpha})$ to $\mathsf{CP}(G)$, a stable group with respect to $(\boldsymbol{r}, \boldsymbol{\alpha})$ is a non-empty subset $S \in V$, if the following conditions hold.

(1) For any $v \in V \setminus S$, r_v satisfies either $r_v > \max_{u\in S} r_u$ or $r_v < \min_{u\in S} r_u$;
(2) For any $v \in V$, if $r_v > \max_{u\in S} r_u$, we have that $\forall (v, u) \in E \cap (\{v\} \times S)$, $\alpha_{v,u} = 0$;
(3) For any $v \in V$, if $r_v < \min_{u\in S} r_u$, we have that $\forall (u, v) \in E \cap (S \times \{v\})$, $\alpha_{u,v} = 0$.

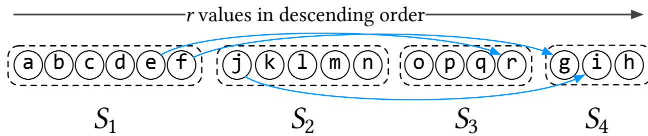

Figure 5: Stable groups w.r.t. $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$.

Definition 5.1 defines stable group. We further use the stable groups w.r.t. $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ (depicted in Figure 5) as an example to illustrate the properties of the stable groups. The definition indicates that if we sort all vertices $u \in V$ w.r.t. their r values in descending order, we can observe the following properties:

(1) The vertices within the same stable group S form a consecutive subsequence of the whole sequence. For example, the stable groups in Figure 5 give a partition to the entire sequence.

(2) The weights of edges whose endpoints fall into different stable groups are assigned to the endpoints with smaller r values. In Figure 5, we use arrows to denote weight assignments of edges across different stable groups. Note that edges within the same stable group are omitted. We can find that all arrows are pointed to the vertices with smaller r values.

Before discussing how to extract stable groups from $(\boldsymbol{r}, \boldsymbol{\alpha})$, we first prove by the following lemma that the stable groups help derive the upper and lower bounds of compact numbers.

Lemma 5.2. *Given a feasible solution $(\boldsymbol{r}, \boldsymbol{\alpha})$ to $\mathsf{CP}(G)$ and a stable group S w.r.t. $(\boldsymbol{r}, \boldsymbol{\alpha})$, for all $u \in S$, we have that $\min_{v\in S} r_v \leq \phi(u) \leq \max_{v\in S} r_v$.*

Proof. We prove the lemma by contradiction. According to Theorem 4.6, $\forall u \in V, \phi(u) = r_u^*$. Suppose there exists a vertex $u \in S$ such that $r_u^* = \phi(u) < \min_{v\in S} r_v \leq r_u$. There must exist another vertex $x \in V$ such that $r_x^* = \phi(x) > r_x$. Here $r_x \geq \min_{v\in S} r_v$ according to the 3-rd condition in Definition 5.1. There exists $\epsilon > 0$ such that we could increase r_u^* by ϵ and decrease r_x^* by ϵ, via manipulating the corresponding $\boldsymbol{\alpha}^*$ values, to strictly decrease $\|\boldsymbol{r}^*\|_2^2$ (i.e., the objective function). This contradicts that $\boldsymbol{r}^*$ is the optimal solution to $\mathsf{CP}(G)$. □

According to Lemma 5.2, the minimum and maximum r values in the stable group S are the lower and upper bounds of compact numbers of vertices in S, respectively.

Now the key issue becomes how to extract stable groups from the approximate solution $(\boldsymbol{r}, \boldsymbol{\alpha})$ provided by `Frank-Wolfe`. Our stable groups closely connect to the stable subsets in [15]. Compared to our stable group, the stable subset does not allow vertices not in the subset having larger r values than vertices in the subset. For example, S_2 cannot be a stable subset, but $S_1 \cup S_2$ is a stable subset. Hence, the stable subset [15] can be treated as the union of several stable groups with largest r values. Due to the close relationship between our stable groups and stable subsets in [15], we adapt the stable subset extraction method [15] to extract our stable groups. In general, we first extract the tentative stable groups from $\boldsymbol{r}$, and then verify or merge the tentative ones via Definition 5.1 to give the stable groups. For the optimal solution $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ to $\mathsf{CP}(G)$, we can just group the vertices with the same r values to form the stable groups (refer Table 2 and Figure 5). Although we cannot perform such aggregation based on $(\boldsymbol{r}, \boldsymbol{\alpha})$, the stable groups obtained from $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ can still provide useful heuristic for us.

Consider the stable groups in Figure 5. After sorting the vertices according to their r values, let $V_{[1:i]}$ denote the first i vertices in the sequence. We can find that the index i of the last vertex in each stable group is exactly $\arg\max_{j\geq i} \mathsf{density}(G[V_{[1:j]}])$, i.e., the subgraph induced by more vertices than the first i vertices cannot have a larger density. For example, the index of the last element in S_1 is 6, and the density of $G[V_{[1:6]}]$ is 3 and is the maximum among all subgraphs induced by $G[V_{[1:j]}]$, where $j \geq 6$. Hence, we use $\arg\max_{j\geq i} \mathsf{density}(G[V_{[1:j]}])$ to find indices for extracting stable group candidates. Note we break the tie via taking a larger index value for j, when two index values give the same density. Then, we verify each candidate by Definition 5.1. Specifically, we restrict all edges adjacent to the candidate stable group satisfying conditions (2) and (3) of Definition 5.1 by modifying $\boldsymbol{\alpha}$ and $\boldsymbol{r}$ and

then check whether condition (1) of Definition 5.1 is fulfilled. If so, the candidate becomes a stable group. Otherwise, it may need to be merged with the next candidate.

Algorithm 2: Extract stable groups from $(\boldsymbol{r}, \boldsymbol{\alpha})$

1 **Function** ExtractSG($G = (V, E)$, $\boldsymbol{r}$, $\boldsymbol{\alpha}$):
2 sort vertices in V according to $\boldsymbol{r}$: $r_{u_1} \geq r_{u_2} \geq \cdot \geq r_{u_n}$;
3 $I \leftarrow \{i \mid i = \arg\max_{i \leq j \leq n} \text{density}(G[V_{[1:j]}])\}$;
4 $\hat{\mathcal{S}} \leftarrow$ partition V according to I;
5 $\mathcal{S} \leftarrow \emptyset, S \leftarrow \emptyset$;
6 **while** $\hat{\mathcal{S}}$ *is not empty* **do**
7 $S' \leftarrow$ pop out the first candidate from $\hat{\mathcal{S}}$;
8 $S \leftarrow S \cup S'$;
9 **if** S *is a stable group* **then** `// via Definition 5.1`
10 put S into $\mathcal{S}$, $S \leftarrow \emptyset$;
11 **foreach** $S \in \mathcal{S}$ **do**
12 **foreach** $u \in S$ **do** $\overline{\phi}(u) \leftarrow \min\{\overline{\phi}(u), \max_{v \in S} r_v\}$;
13 **foreach** $u \in S$ **do** $\underline{\phi}(u) \leftarrow \max\{\underline{\phi}(u), \min_{v \in S} r_v\}$;
14 **return** $\mathcal{S}, \overline{\phi}, \underline{\phi}$

Based on the above discussion, we present the algorithm to extract stable groups from $(\boldsymbol{r}, \boldsymbol{\alpha})$, named ExtractSG, in Algorithm 2. ExtractSG first sorts the vertices in V according to their r values descendingly (line 2). Then, ExtractSG finds the indices I and extracts stable group candidates $\hat{\mathcal{S}}$ following the above heuristic (lines 3–4). Next, we check the candidate in $\hat{\mathcal{S}}$ one-by-one via Definition 5.1 (lines 6–10): if the candidate is a stable group, push it into the list of stable groups $\mathcal{S}$ (lines 9–10); otherwise, the current candidate S will be merged with the next candidate S' in the next iteration (line 8). After all stable groups in $\mathcal{S}$ are obtained, we update the upper and lower bounds of compact numbers according to Lemma 5.2 (lines 11–14). Finally, ExtractSG returns the stable groups $\mathcal{S}$ and updated upper and lower bounds of compact numbers (line 14).

5.2 Prune Invalid Vertices

In this subsection, we present how to prune the vertices, which are certainly not contained by any LDS, based on compact number bounds derived in Section 5.1.

We begin with a powerful corollary.

Corollary 5.3 (Pruning rule 1). *For any $u \in V$, if $\exists(u, v) \in E$, such that $\underline{\phi}(v) > \overline{\phi}(u)$, u is not contained by any LDS in G.*

Proof. The corollary directly follows Lemma 4.4. □

Example 5.4 (Pruning rule 1). Reconsider the graph G in Figure 1 and the stable groups in Figure 5. For vertices in S_3, we can prune r, as shown in Figure 6. Because for edge (e, r), we have $\overline{\phi}(\text{r}) = \frac{3}{2} < \underline{\phi}(\text{e}) = \frac{5}{2}$, respectively. Similarly, the two vertices g and i in S_4 are also pruned by Pruning rule 1 (Corollary 5.3).

Following Example 5.4, we can find that after removing r and q from G, denoting the graph after removal by G', there is only one edge adjacent to o and q, as shown in Figure 6. We can find that $\overline{\phi}_{G'}(\text{o}) = \overline{\phi}_{G'}(\text{q}) = 1 < \underline{\phi}(\text{o}) = \underline{\phi}(\text{q}) = \frac{3}{2}$. It means that

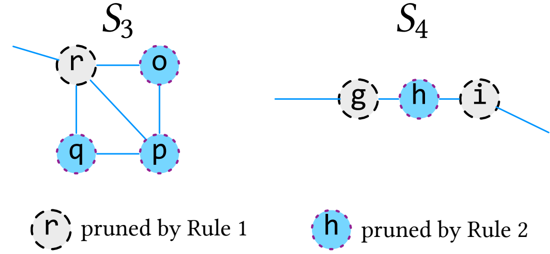

Figure 6: Pruning rules illustration.

any LDS in G cannot contain o and q, because it needs to include some already pruned vertices, such as r, to obtain a $\underline{\phi}(\text{o})$-compact subgraph containing o or q in G. Hence, we derive another pruning rule from the above case.

Lemma 5.5 (Pruning rule 2). *Let G' denote the graph after pruning vertices according to Corollary 5.3 and Lemma 5.5. For any vertex u in G', if $\overline{\phi}_{G'}(u) < \underline{\phi}(u)$, u is not contained by any LDS in G.*

Proof. $\overline{\phi}_{G'}(u) < \underline{\phi}(u)$ means that only relying on the vertices in G' cannot form a $\underline{\phi}(u)$-compact subgraph containing u, which means that some already pruned vertices are needed. Hence, u cannot be contained by any LDS in G. □

To efficiently compute $\overline{\phi}_{G'}(u)$ in G', we use k-core [50], which is a cohesive subgraph model, following [45].

Definition 5.6 (k-core and core number [50]). The k-core of G is the maximal subgraph $G[S]$ such that for any $u \in S$, $d_{G[S]}(u) \geq r$. For any $u \in V$, the *core number* of u, denoted by $\text{core}_G(u)$, is the largest k such that u is contained in the k-core of G.

Lemma 5.7. *Let G' denote the graph after pruning invalid vertices. $\text{core}_{G'}(u)$ provides an upper bound of $\phi_{G'}(u)$.*

Proof sketch. The lemma follows from Lemma 4.7 of [45]. □

Following the above discussion about Example 5.4, Lemma 5.7 provides a useful approach to obtain the upper bounds of compact numbers of o and p after r and q are removed.

Example 5.8 (Pruning rule 2). After r is pruned from G in Example 5.4, we obtain the upper bounds of compact numbers of o, q, and p in the residual graph G' via Lemma 5.7: $\overline{\phi}_{G'}(\text{o}) = \overline{\phi}_{G'}(\text{q}) = \overline{\phi}_{G'}(\text{p}) = 1$. Then, we apply Pruning rule 2 (Lemma 5.5) to remove o, q, and p from the graph, as shown in Figure 6. Analogically, h in S_4 is also pruned.

Following the above two examples, we further compare our pruning rules with those in LDSflow [45]. LDSflow mainly used core numbers for pruning: for vertex u, if $(u, v) \in E$ and $\text{core}_G(u) < \frac{\text{core}_G(v)}{2}$, or $\text{core}_{G'}(u) < \frac{\text{core}_G(u)}{2}$, u can be pruned, where G' denotes the graph with some vertices already pruned. From the perspective of compact numbers, the rationale behind the pruning in LDSflow is that they actually used core numbers to provide relatively loose upper and lower bounds for compact numbers.

Based on the two pruning rules, i.e., Corollary 5.3 and lemma 5.5, we present our pruning algorithm, named Pruning, in Algorithm 3.

Algorithm 3: Prune invalid vertices

```
1  Function Pruning(G = (V, E), 𝒮, φ̄, φ̲):
2      G′ = (V′, E′) ← G;
3      foreach (u, v) ∈ E do
4          if φ̄(u) < φ̲(v) then remove u from G′;
5      compute core_{G′}(u) for all vertices in G′;
6      while ∃u ∈ V′, core_{G′}(u) < φ̲(u) do
7          remove u from G′;
8          update core numbers of vertices adjacent to u;
9      foreach stable group S ∈ 𝒮 do S ← S ∩ V′;
10     return 𝒮
```

We first replicate the graph G to G' (line 1). Then, we apply Pruning rule 1 (Corollary 5.3) to remove invalid vertices (lines 3–4). Next, we compute the core numbers for all vertices in G' (line 5). Afterwards, Pruning rule 2 (Lemma 5.5) is applied (lines 6–8). Finally, Pruning updates the stable groups by intersecting them with the vertices not been pruned (line 9), and returns the updated stable groups (line 10).

The following subsection will introduce how to extract and verify LDS from the updated stable groups.

5.3 Extract and Verify LDS

Here, we discuss how to extract and verify the LDS from the stable groups after pruning. First, we can find that the vertices within the stable group S with largest r values in $\mathcal{S}$ satisfy Lemma 4.4 w.r.t. the current valid vertices, otherwise they are pruned in Pruning. But, we are not sure whether $G[S]$ is the densest among all its subgraphs (i.e., $G[S]$ is self-densest) and whether there exists another larger density($G[S]$)-compact subgraph of G containing $G[S]$, according to Definition 3.3.

We first examine whether $G[S]$ is self-densest because the computation cost for self-densest examination is smaller than checking whether it is a maximal density($G[S]$)-compact subgraph.

Verifying whether $G[S]$ is the densest among all subgraphs of $G[S]$ is one step in the binary search of computing densest subgraph [26], i.e., checking whether there is a subgraph with higher density than density($G[S]$). Generally, we use IsDensest to verify the self-densest via computing the max-flow on the flow network generated based on $G[S]$ following [52].

If $G[S]$ is the DS of itself (i.e., IsDensest returns True), we need to further verify whether $G[S]$ is the maximal density($G[S]$)-compact subgraph in G. We first review how $G[S]$ is verified as an LDS in [45], and next we give our improved verification algorithm.

Qin et al. [45] first use breadth-first-search starting from $G[S]$ to traverse each vertice u with $\overline{\phi}(u) \geq$ density($G[S]$). Recall that they use $\text{core}_G(u)$ as $\overline{\phi}(u)$ (briefed in Section 5.2). We use G^t to denote the subgraph traversed. If there does not exist an already computed LDS in G^t, $G[S]$ is an LDS. Otherwise, Qin et al. construct a flow network based on G^t, then use the min-cut algorithm to find all maximal density($G[S]$)-compact subgraphs in G^t, and check whether $G[S]$ is maximal.

We can observe that the verification algorithm in [45] needs to compute the min-cut on the flow-network based on the vertices with $\overline{\phi}(u) \geq$ density($G[S]$). We will show that only the vertices with $\overline{\phi}(u) \geq$ density($G[S]$) and $\underline{\phi}(u) \leq$ density($G[S]$), which form a subset of the set of vertices needed in [45], are needed to verify whether $G[S]$ is an LDS of G.

Algorithm 4 presents our improved verification algorithm, named IsLDS. IsLDS first initializes an empty queue Q, an empty vertex set T, an empty edge set L, and ρ with density($G[S]$) (lines 2–3). Next, the algorithm performs a breadth-first search starting from S (lines 4–13). Specifically, IsLDS uses Q to store the vertices to be traversed. Each time, it pops out the first vertex v from Q (line 5), and iterates all neighbors of v (lines 8–13). For each neighbor w, if $\underline{\phi}(w) > \rho$, w will not be added to T and Q, but a self loop of v is added to L (lines 10–11). If $\underline{\phi}(w) \leq \rho \leq \overline{\phi}(w)$, w is added into Q and T (lines 12–13). If IsLDS does not encounter a vertex with $\underline{\phi}(w) > \rho$ during the traversal, we can return True (line 14), which means that there does not exist an already computed LDS in the traversed subgraph. Otherwise, we construct a subgraph G^t with all edges induced by T and self loops in L (lines 15). Afterward, we compute all ρ-compact subgraphs in G^t via min-cut following [45] (line 16). Finally, we return True if $G[S]$ is maximal ρ-compact; otherwise, False is returned (line 17). We can observe that G^t only contains vertices with $\overline{\phi}(w) \geq$ density($G[S]$) $\geq \underline{\phi}(w)$. Hence, the flow network generated in our algorithm is much smaller than that generated in [45].

Algorithm 4: Check whether $G[S]$ is an LDS of G

```
1  Function IsLDS(S, φ̄, φ̲, G = (V, E)):
2      Q ← an empty queue, ρ ← density(G[S]);
3      T ← ∅, L ← ∅, isLDS ← True;
4      foreach u ∈ S do
5          if u ∉ T then push u to Q, insert u into T;
6          while Q is not empty do
7              v ← pop out the front vertex in Q;
8              foreach (v, w) ∈ E do
9                  if w ∉ T then
10                     if φ̲(w) > ρ then
11                         add edge (v, v) to L, isLDS ← False
12                     else if φ̄(w) > ρ then
13                         push w to Q, add w into T
14     if isLDS then return True;
15     G^t ← (T, E(T) ∪ L);
16     G′ ← all ρ-compact subgraphs in G^t via min-cut;
17     return G[S] is a connected component in G′
```

Before proving the correctness of Algorithm 4, we use an example to illustrate the traversed subgraph G^t.

Example 5.9. Consider the graph in Figure 7. Suppose $S = \{\mathsf{f}, \mathsf{g}, \mathsf{h}\}$ and we want to verify whether $G[S]$ is an LDS of G. Clearly, $G[S]$ is the DS of itself. We illustrate the scope of the subgraph G^t in Algorithm 4. Following Algorithm 4, T contains f, g, h, e and L consists of edge (e, e) because $\underline{\rho}$(b) > 1 = density($G[S]$). Hence, G^t in our IsLDS contains 4 vertices and 5 edges, while the traversed subgraph in LDSflow [45] contains all eight vertices (i.e., a, b, c, d,

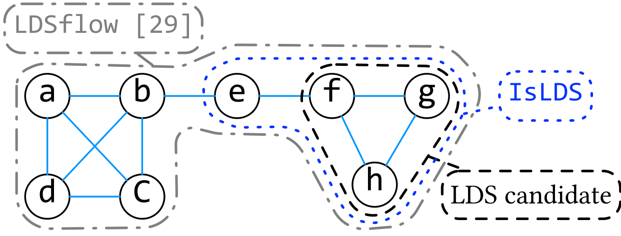

Figure 7: LDS verification illustration.

e, f, g, h) as shown in Figure 7, because core numbers of all vertices are larger than 1, i.e., density($G[S]$).

THEOREM 5.10. *Given a graph G and a subgraph $G[S]$, where $G[S]$ is the DS of itself, $G[S]$ is an LDS of G if and only if Algorithm 4 returns* True.

PROOF. First, if $G[S]$ is an LDS, Algorithm 4 returns True. Because only the loops in L might increase the compact numbers in G^t compared to the compact numbers in G. Hence, $G[S]$ is still an LDS in G^t. Otherwise the maximal density($G[S]$)-compact subgraph containing $G[S]$ must contain a vertex u with self loop, and then we can construct a larger density($G[S]$)-compact subgraph in G by including vertices with $\underline{\phi}(w) >$ density($G[S]$) connected to u. Hence, the contradiction proves the claim.

On the other direction, if $G[S]$ is not an LDS of G, we will find a larger density($G[S]$)-compact subgraph containing $G[S]$ in G^t. Hence, $G[S]$ is also not an LDS in G^t. Thus, the algorithm returns False. □

By now, we have introduced all building blocks of our LDS algorithm. In the next subsection, we will present our LDS algorithm LDScvx by combining these components.

5.4 The Overall Algorithm LDScvx

Combining Algorithms 1 to 4 with reference to Figure 4, we will obtain our LDS algorithm, named LDScvx in Algorithm 5.

Algorithm 5: Our LDS algorithm, LDScvx

```
  Input  :A graph G = (V, E) and two integers k and N
  Output :LDS's with top-k densities
1 G' ← G;
2 stk ← an empty stack;
3 while k > 0 do
4     (r, α) ← Frank-Wolfe(G', N);
5     𝒮, φ̄, φ̲ ← ExtractSG(G', r, α);
6     𝒮 ← Pruning(G', 𝒮, φ̄, φ̲);
7     foreach S ∈ 𝒮 reversely do push S into stk;
8     S ← pop out the top stable group from stk;
9     if IsDensest(G[S]) then
10        if IsLDS(S, φ̄, φ̲, G) then output G[S], k ← k − 1;
11        if stk is empty then break;
12        S ← pop out the top stable group from stk;
13    G' ← G[S];
```

In LDScvx, we first assign G to G' (line 1) and initialize an empty stack stk (line 2). Next, we extract the stable groups from the graph G' via Frank-Wolfe, ExtractSG, and Pruning (lines 4-6). Then, the algorithm pushes the stable groups in $\mathcal{S}$ reversely into stk (line 7). For stable groups in stk, the corresponding $\underline{\phi}$ value is decremented from top to bottom. Afterward, the first stable group in stk, which is also the one with the highest $\underline{\phi}$ value, is poped out (line 8) and is examined by IsDensest and IsLDS (line 9–10). If $G[S]$ is an LDS, we output it and decrease k by 1 (line 10). If $G[S]$ is not an LDS but is the DS of itself, we update S as the top stable group from stk (line 12). Next, we assign $G[S]$ to G' for the next iteration (line 13). The above process is repeated until top-k LDS's are found (line 3), or the stack is empty (line 11).

Next, we use an example to explain further the overall procedure of LDScvx (Algorithm 5).

Example 5.11. We still use the graph G in Figure 1 as the example. Suppose we want to find top-2 LDS's from G. Assume we obtain $(\boldsymbol{r}^*, \boldsymbol{\alpha}^*)$ in Table 2 after Frank-Wolfe. Then, we will obtain the stable groups shown in Figure 5, as well as the upper and lower bounds of compact numbers via ExtractSG. Next, in the Pruning process, the vertices in S_3 and S_4 will be pruned according to Corollary 5.3 and Lemma 5.5, which means that $\mathcal{S}$ contains S_1 and S_2. Afterward, S_2 and S_1 will be pushed into the stack stk. Now, the stable groups in stk satisfy that the compact numbers of vertices in the stable group higher in stk are larger than those in the stable group lower in stk. Next, we pop out the top stable group S_1 from stk and verify that it is an LDS via IsDensest and IsLDS. We output $G[S_1]$ as the first LDS, pop out S_2 from stk, and repeat the above process. After S_2 is verified as an LDS, stk is empty, we break while loop (line 10 in Algorithm 5). In the end, we obtain two LDS's, $G[S_1]$ and $G[S_2]$.

Complexity. The time complexity of LDScvx is $O((N_{\mathsf{FW}}+N_{\mathsf{SG}}) \cdot (n+m) + N_{\mathsf{Flow}} \cdot t_{\mathsf{Flow}})$, where N_{FW} is number of iterations that Frank-Wolfe needs, and $N_{\mathsf{SG}} \leq n$ is the number of stable groups in total, N_{Flow} is number of times IsLDS and IsDensest are called, and t_{Flow} denotes the time complexity of max-flow computation. Note that an iteration in Frank-Wolfe and verifying a stable group in ExtractSG both take $O(n+m)$ time cost. The memory complexity is $O(n+m)$.

6 EXPERIMENTS

6.1 Setup

We use nine real datasets [6, 7, 36, 46, 58] which are publicly available[1] except for TL. The TL dataset is provided by a television company TCL Technology. The dataset contains film information provided on its smart TV platform, mainly used for a case study. Other graph datasets cover various domains, including social networks (e.g., LiveJournal), e-commerce (e.g., Amazon), and video platforms (e.g., YouTube). Table 3 summarizes the statistics.

We compare the following LDS algorithms:

- LDScvx is our convex-programming based top-k LDS algorithm (Section 5.4).
- LDSflow [45] is the state-of-the-art top-k LDS algorithm based on max-flow.

[1] https://networkrepository.com/, https://snap.stanford.edu/data/index.html, and https://law.di.unimi.it/datasets.php

Table 3: Graphs used in our experiments.

Dataset	Full name	Category	$\|V\|$	$\|E\|$
TL	movie-TCL	Movie	108K	168K
AM	com-amazon	E-commerce	335K	926K
YT	com-youtube	Video-sharing	1.13M	2.99M
LJ	com-lj	Social	4.00M	34.7M
OR	com-orkut	Social	3.07M	117M
IC	indochina-2004	web	7.41M	194M
AB	arabic-2005	web	22.7M	639M
IT	web-it-2004-all	Web	41.3M	1.03B
LK	links-anon	Hyperlink	52.6M	1.61B

All the algorithms above are implemented in C++ with STL used. The source codes of `LDSflow` are provided by the authors of [45]. We run all the experiments on a machine having an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz processor and 256GB memory, with Ubuntu installed.

6.2 Efficiency Results

To choose the best setting of N, i.e., the number of `Frank-Wolfe` iterations, we tested the running time of `LDScvx` w.r.t. different values of N from 50 to 200 with k fixed to 5. Table 4 reports the average relative running time w.r.t. N over eight large datasets. The relative running time for a specific value of N on each dataset is obtained via dividing the running time by the minimum running time over all N values for the dataset. The mean is then obtained by averaging over all datasets. We can find that when $N = 100$, we obtain the minimum average relative running time. Hence we use $N = 100$ as the default parameter value in other experiments.

Table 4: Relative running time w.r.t. different N

N	50	100	150	200
Average relative time	1.64	1.10	1.12	1.20

Next, we compare our top-k LDS algorithm `LDScvx` with the state-of-the-art `LDSflow` [45] w.r.t. running time.

We first fix $k = 5$ to overview the two algorithms on the nine datasets w.r.t. running time. Figure 8 shows the efficiency results of the two algorithms. The datasets are ordered by graph size on the x-axis. Note that for some datasets, the bars of `LDSflow` touch the solid upper line, which means `LDSflow` cannot finish within 600 hours on those datasets. From Figure 8, we can observe: first, the running time of `LDSflow` increases along with the graph size increasing; second, `LDScvx` is up to four orders of magnitude faster than `LDSflow`. We reckon that the speedup comes from more vertices pruned due to tighter bounds, fewer LDS candidates verified, and smaller flow networks as stated in Section 5.

We further provide the running time trends of the two algorithms w.r.t. different k values on four representative datasets in Figure 9 due to space limit. For `LDSflow`, we do not show its trends on dataset LK because it cannot finish within 600 hours even with $k = 5$. We can see that when k increases, the running time of both `LDScvx` and `LDSflow` increases. Furthermore, there is a significant increase in `LDSflow` (about two orders of magnitude) when k is increased from

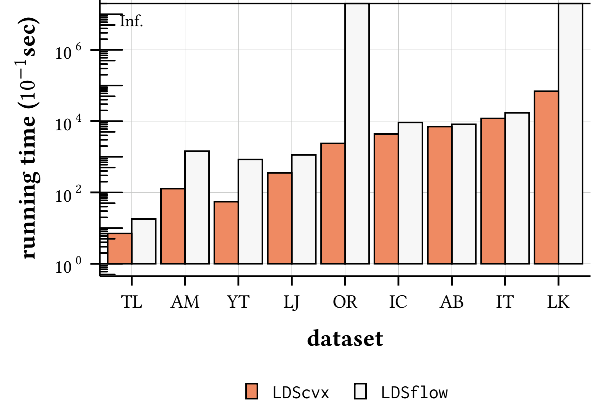

Figure 8: Efficiency of `LDScvx` and `LDSflow` with $k = 5$.

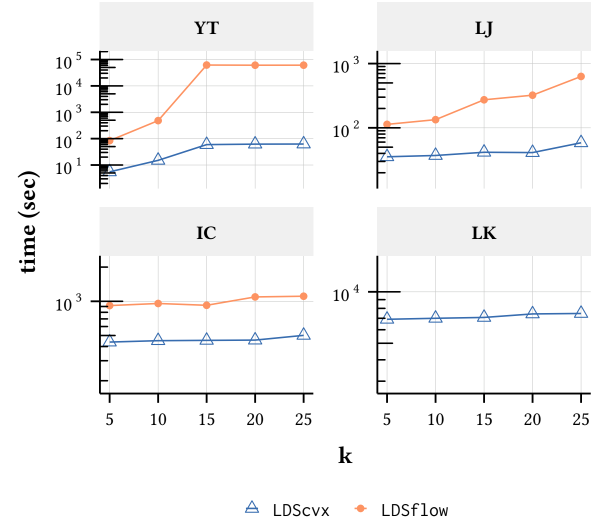

Figure 9: Efficiency of `LDScvx` and `LDSflow` w.r.t. different k.

10 to 15 on dataset YT. To analyze this significant surge, we examine the running process of the two algorithms. We found that many LDS candidates in `LDSflow` do not pass the verification process when k is increased from 10 to 15. Recall that `LDSflow` needs to perform max-flow computation on a flow network to verify each LDS candidate according to Section 5.3. Hence, the number of failed LDS candidates can reveal to some extent the reason of the spike in running time.

Table 5: Numbers of failed LDS candidates on YT w.r.t. k.

Algorithm	$k = 10$	$k = 15$	increased times
`LDScvx`	37	84	2.27×
`LDSflow`	277	18399	66.42×

We report the numbers of failed LDS candidates on YT with k = 10 and 15 for both algorithms in Table 5. We observe that the number of failed candidates in LDSflow increased around 66× when k increases from 10 to 15, which explains the surge of running time in Figure 9. In contrast, the number of failed candidates in LDScvx only increases by about 2×. This is why the running time of LDScvx does not increase much when k is increased from 10 to 15. Another observation from Table 5 is that the failed numbers for LDScvx are smaller than LDSflow on both k values, respectively. The reason is that we provide tight upper and lower bounds for compact numbers via convex programming and stable groups, and the tight bounds further enable more vertices to be pruned, which results in fewer LDS candidates to be examined. Apart from that, to examine an LDS candidate, our LDScvx only needs a subgraph of what is needed in LDSflow to calculate the max-flow by leveraging the lower bounds of compact numbers, according to Section 5.3. We believe the above two improvements explain why we are around three orders of magnitude faster on YT when k = 15.

We further tested the scalability w.r.t. the density via synthetics datasets. The six synthetic graphs are generated by Barabási-Albert (BA) model [1]. The numbers of vertices in all synthetic graphs are fixed to 1,000,000, and the densities are increased linearly from 2 to 12. In other words, the numbers of edges are from 2,000,000 to 12,000,000. We report the running time of LDScvx on the six synthetic datasets in Figure 10. We observe that LDScvx scales well w.r.t. the graph density.

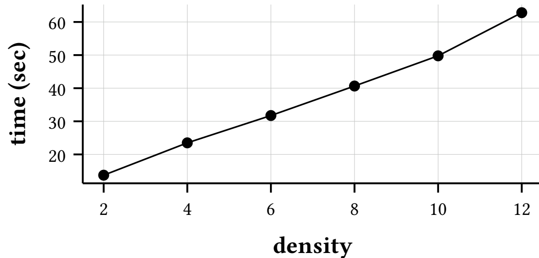

Figure 10: The scalability w.r.t. the density.

6.3 Memory Usage Comparison

In this subsection, we test the memory usage of the two algorithms. The maximum memory usage is tested via the Linux command `/usr/bin/time -v`. For the cases that LDSflow does not finish reasonably, we record the maximum resident memory during the running process. Figure 11 reports the maximum memory usage of the two algorithms. The datasets are sorted on the x-axis in ascending order of graph size. We can observe that the memory usages of both algorithms increase along with the increasing graph size. Besides, the memory costs of LDScvx and LDSflow are around the same scale because the two algorithms take linear memory usage w.r.t. the graph size.

6.4 Time Proportion Analysis

Here, we evaluate how the different building blocks of LDScvx contribute to the whole running time after the graph is loaded and preprocessed. Figure 12 reports the proportion of each part in the total running time: Frank-Wolfe (Algorithm 1), ExtractSG (Algorithm 2), Pruning (Algorithm 3), and VerifyLDS (Algorithm 4) with k = 5 over nine datasets. We can observe that the Frank-Wolfe computation is the most computationally expensive part on most datasets. For LK and YT datasets, the time used by verifying LDS takes the majority. We further examine the number of failed LDS candidates (i.e., on which IsLDS returns False) on the nine datasets. Table 6 reports the results. We can find that the numbers of failed candidates on LK and YT are much higher than other datasets, which means that these two datasets need more time to verify LDS's, which explains the results in Figure 12 to some extent. For IC, the time used by verifying LDS is also high, because the first LDS in IC is quite large and takes relatively long time to verify.

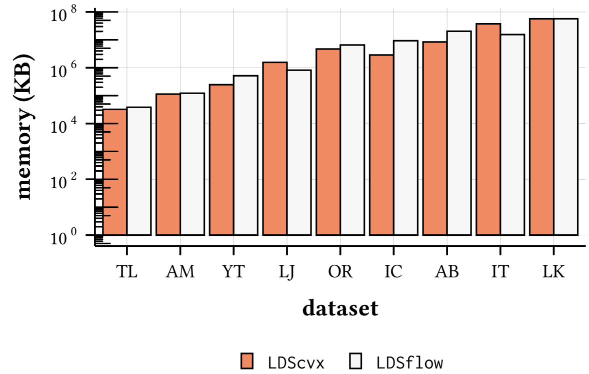

Figure 11: Memory usage of algorithms with k = 5.

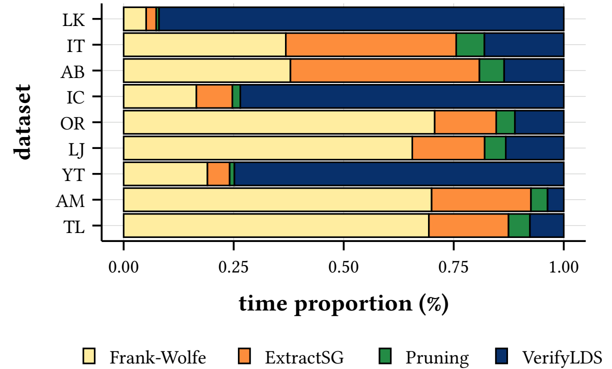

Figure 12: Proportion of each part in total running time.

Table 6: Numbers of failed LDS candidates with k = 5.

Dataset	TL	AM	YT	LJ	OR	IC	AB	IT	LK
#failed	1	0	9	1	1	0	0	0	6

6.5 Ablation Study of `IsLDS`

Here, we conduct an ablation study on `LDScvx` to understand the effectiveness of `IsLDS` (Algorithm 4). Recall that in `IsLDS` we only include the vertices satisfying $\overline{\phi}(u) \geq \text{density}(G[S]) \geq \underline{\phi}(u)$ into the flow network computation, while its counterpart in `LDSflow` [45] includes all vertices satisfying $\overline{\phi}(u) \geq \text{density}(G[S])$, named by `IsLDS-ab` (ablation). We report the time used for verifying LDS's with `IsLDS` and `IsLDS-ab`, respectively, on the nine datasets when $k = 5$ in Table 7. Here, the time of `IsLDS-ab` is measured by replacing `IsLDS` with `IsLDS-ab` in `LDScvx`. The time of `IsLDS-ab` on LK is marked as $\geq$ 259200s because it cannot finish within three days. From the results, we observe that the verification process with `IsLDS` is up to 110× faster than that with `IsLDS-ab`. Hence, reducing the number of vertices by the lower bounds in `IsLDS` does help speed up the LDS verification process.

Table 7: Effect of `IsLDS` with $k = 5$.

Dataset	`IsLDS`	`IsLDS-ab`	Speedup
TL	0.0399s	0.0748s	1.87×
AM	0.3334s	0.3623s	1.09×
YT	2.6575s	80.9994s	30.48×
LJ	2.1204s	2.3924s	1.13×
OR	18.4089s	723.6035s	39.31×
IC	285.4502s	288.9184s	1.01×
AB	60.2669s	62.0416s	1.03×
IT	147.9361s	188.8527s	1.28×
LK	2335.4461s	$\geq$ 259200s	$\geq$ 110.99×

6.6 Subgraph Statistics

In this subsection, Figure 13 reports the densities of the top-15 densest subgraphs w.r.t. the size (number of vertices) returned by three different models on four datasets, where `Greedy` iteratively computes a densest subgraph and removes it from the graph, and `FDS` denotes the density-friendly decomposition model [15, 53]. From Figure 13, we can find that the densest subgraph can be found by all three algorithms, because the densest subgraph is also an LDS. But there are some dense subgraphs found by `Greedy` that do not qualify as LDS's. Hence, the subgraphs found by `LDScvx` have a wide range of densities and sizes. For `FDS`, we can find that the subgraphs have increasing sizes and decreasing densities. This is because `FDS` outputs a chain of subgraphs, where each subgraph is nested within the next one, and the inner one is denser than the outer ones.

6.7 Case Study

Here, we perform a case study on the TL dataset. The TL dataset is provided by a Chinese television company, TCL Technology, which contains three types of vertices: director, movie, and actor.

After examining the top-10 LDS's returned by our LDS algorithm, we found that the LDS's on the TL dataset are about different topics. For example, Figure 2 shows the LDS with the third-highest density. This LDS contains eight movies related to a famous Japanese sci-fiction series "Ultraman" with four actors and one director. In this LDS, the four actors participated in all eight films, and the director directed five of them. The LDS with the second-highest density is a subgraph about Chinese martial fiction. Other LDS's cover topics about western films, Danish comedies, and cartoons, while the DS model can only find a large subgraph about western films. Hence, from the output of the case study, we reckon that the subgraphs returned by the LDS algorithm are good representations of *different* local dense regions in the graph.

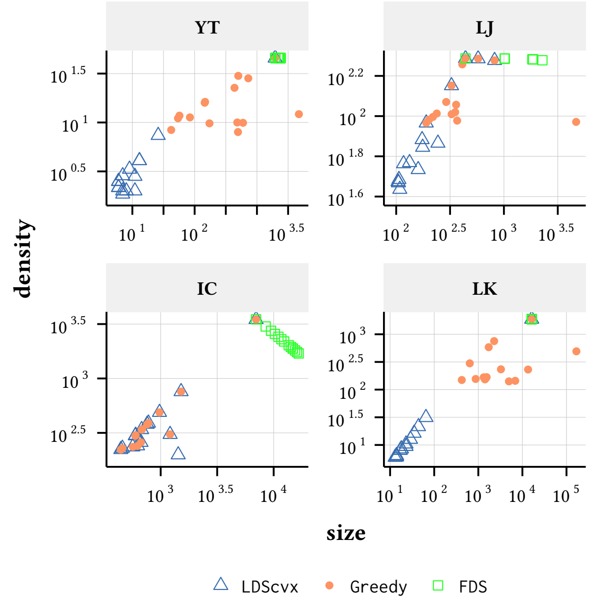

Figure 13: Subgraph statistics: density w.r.t. size.

7 CONCLUSION

In this paper, we study the problem of finding the top-k locally densest subgraphs (LDS's) in a graph to identify the local dense regions. The LDS's are usually compact and dense. To facilitate the LDS discovery, we propose a new concept, named *compact number* for each vertex, which denotes the compactness of the most compact subgraph containing the vertex. By leveraging the compact number and its relations with the LDS problem and a specific convex program, we derive a convex-programming based algorithm `LDScvx` following the pruning-and-verifying paradigm. Extensive experiments on nine datasets show that `LDScvx` is up to four orders of magnitude faster than the state-of-the-art algorithm.

ACKNOWLEDGMENTS

Chenhao Ma and Xiaolin Han were supported by HKU Project 10400599 and the Innovation and Technology Commission of Hong Kong (ITF project MRP/029/18). Reynold Cheng was supported by the HKU-TCL Joint Research Center for Artificial Intelligence (Project no. 200009430) and the Guangdong–Hong Kong-Macau Joint Laboratory Program 2020 (Project No: 2020B1212030009). Lakshmanan's research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

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

[2] Reid Andersen and Kumar Chellapilla. 2009. Finding dense subgraphs with size bounds. In *International workshop on algorithms and models for the web-graph*. Springer, 25–37.

[3] Yuichi Asahiro, Refael Hassin, and Kazuo Iwama. 2002. Complexity of finding dense subgraphs. *Discrete Applied Mathematics* 121, 1-3 (2002), 15–26.

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

[5] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *PVLDB* (2012).

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM, 587–596.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th international conference on World Wide Web*. ACM, 595–602.

[8] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*. 573–583.

[9] Lijun Chang and Miao Qiao. 2020. Deconstruct Densest Subgraphs. In *Proceedings of The Web Conference 2020*. 2747–2753.

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

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

[12] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *IEEE Transactions on knowledge and data engineering* 24, 7 (2010), 1216–1230.

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

[14] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: scalable community detection in massive networks via small-diameter k-plexes. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1272–1281.

[15] Maximilien Danisch, T-H Hubert Chan, and Mauro Sozio. 2017. Large scale density-friendly graph decomposition via convex programming. In *Proceedings of the 26th International Conference on World Wide Web*. 233–242.

[16] Riccardo Dondi, Mohammad Mehdi Hosseinzadeh, and Pietro H Guzzi. 2021. A novel algorithm for finding top-k weighted overlapping densest connected subgraphs in dual networks. *Applied Network Science* 6, 1 (2021), 1–17.

[17] Riccardo Dondi, Mohammad Mehdi Hosseinzadeh, Giancarlo Mauri, and Italo Zoppis. 2021. Top-k overlapping densest subgraphs: approximation algorithms and computational complexity. *Journal of Combinatorial Optimization* 41, 1 (2021), 80–104.

[18] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2007. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international conference on World Wide Web*. 461–470.

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

[20] Yixiang Fang, Wensheng Luo, and Chenhao Ma. 2022. Densest Subgraph Discovery on Large Graphs: Applications, Challenges, and Techniques. *Proceedings of the VLDB Endowment* 15 (2022).

[21] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[22] Marguerite Frank, Philip Wolfe, et al. 1956. An algorithm for quadratic programming. *Naval research logistics quarterly* 3, 1-2 (1956), 95–110.

[23] 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.

[24] Esther Galbrun, Aristides Gionis, and Nikolaj Tatti. 2016. Top-k overlapping densest subgraphs. *Data Mining and Knowledge Discovery* 30, 5 (2016), 1134–1165.

[25] Aristides Gionis, Flavio P Junqueira, Vincent Leroy, Marco Serafini, and Ingmar Weber. 2013. Piggybacking on social networks. In *VLDB 2013-39th International Conference on Very Large Databases*, Vol. 6. 409–420.

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

[27] Xiaolin Han. 2019. Traffic incident detection: a deep learning framework. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 379–380.

[28] Xiaolin Han, Reynold Cheng, Tobias Grubenmann, Silviu Maniu, Chenhao Ma, and Xiaodong Li. 2022. Leveraging contextual graphs for stochastic weight completion in sparse road networks. In *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*. SIAM, 64–72.

[29] Xiaolin Han, Reynold Cheng, Chenhao Ma, and Tobias Grubenmann. 2022. DeepTEA: effective and efficient online time-dependent trajectory outlier detection. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1493–1505.

[30] Xiaolin Han, Daniele Dell'Aglio, Tobias Grubenmann, Reynold Cheng, and Abraham Bernstein. 2022. A framework for differentially-private knowledge graph embeddings. *Journal of Web Semantics* 72 (2022), 100696.

[31] Xiaolin Han, Tobias Grubenmann, Reynold Cheng, Sze Chun Wong, Xiaodong Li, and Wenya Sun. 2020. Traffic incident detection: A trajectory-based approach. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1866–1869.

[32] 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 *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 895–904.

[33] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 813–826.

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

[35] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *International colloquium on automata, languages, and programming*. Springer, 597–608.

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

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

[38] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. LINC: a motif counting algorithm for uncertain graphs. *Proceedings of the VLDB Endowment* 13, 2 (2019), 155–168.

[39] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *Proceedings of the 2022 International Conference on Management of Data*. 845–859.

[40] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1051–1066.

[41] 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.

[42] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On Directed Densest Subgraph Discovery. *ACM Transactions on Database Systems (TODS)* 46, 4 (2021), 1–45.

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

[44] James B Orlin. 2013. Max flows in O (nm) time, or better. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 765–774.

[45] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally densest subgraph discovery. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 965–974.

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[47] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Annual International Conference on Research in Computational Molecular Biology*. Springer, 456–472.

[48] Raman Samusevich, Maximilien Danisch, and Mauro Sozio. 2016. Local triangle-densest subgraphs. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 33–40.

[49] Saurabh Sawlani and Junxing Wang. 2020. Near-optimal fully dynamic densest subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 181–193.

[50] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[51] Ulrich Stelzl, Uwe Worm, Maciej Lalowski, Christian Haenig, Felix H Brembeck, Heike Goehler, Martin Stroedicke, Martina Zenkner, Anke Schoenherr, Susanne Koeppen, et al. 2005. A human protein-protein interaction network: a resource for annotating the proteome. *Cell* 122, 6 (2005), 957–968.

[52] Bintao Sun, Maximilien Danisch, TH Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. *Proceedings of the VLDB Endowment (PVLDB)* (2020).

[53] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly graph decomposition. In *Proceedings of the 24th International Conference on World Wide Web*. 1089–1099.

[54] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *WWW*. 1122–1132.

[55] 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 *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 104–112.

[56] Charalampos Tsourakakis and Tianyi Chen. 2021. Dense subgraph discovery: Theory and application. In *SDM Tutorial*.

[57] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012).

[58] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.