# Faster and Scalable Algorithms for Densest Subgraph and Decomposition

**Harb, Elfarouk**[*]
eyharb2@illinois.edu
University of Illinois at Urbana-Champaign

**Quanrud, Kent**[†]
krq@purdue.edu
Purdue University

**Chekuri, Chandra**[‡]
chekuri@illinois.edu
University of Illinois at Urbana-Champaign

## Abstract

We study the densest subgraph problem (DSG) and the densest subgraph local decomposition problem (DSG-LD) in undirected graphs. We also consider supermodular generalizations of these problems. For large scale graphs simple iterative algorithms perform much better in practice than theoretically fast algorithms based on network-flow or LP solvers. Boob *et al.* [1] recently gave a fast iterative algorithm called GREEDY++ for DSG. It was shown in [2] that it converges to a $(1-\epsilon)$ relative approximation to the optimum density in $O(\frac{1}{\epsilon^2}\frac{\Delta(G)}{\lambda^*})$ iterations where $\Delta(G)$ is the maximum degree and $\lambda^*$ is the optimum density. Danisch *et al.* [3] gave an iterative algorithm based on the Frank-Wolfe algorithm for DSG-LD that takes $O(\frac{m\Delta(G)}{\epsilon^2})$ iterations to converge to an $\epsilon$-additive approximate local decomposition vector $\hat{b}$, where $m$ is number of edges in the graph.

In this paper we give a new iterative algorithm for both problems that takes at most $O(\frac{\sqrt{m\Delta(G)}}{\epsilon})$ iterations to converge to an $\epsilon$-additive approximate local decomposition vector; each iteration can be implemented in $O(m)$ time. We describe a fractional peeling technique which has strong empirical performance as well as theoretical guarantees. The algorithm is scalable and simple, and can be applied to graphs with hundreds of millions of edges. We test our algorithm on real and synthetic data sets and show that it provides a significant benefit over previous algorithms. The algorithm and analysis extends to hypergraphs.

## 1. Introduction

The densest subgraph problem (DSG) is a classical problem in combinatorial optimization and has many real world applications in data mining, network analysis, and machine learning. The input for DSG is an undirected graph $G = (V, E)$ with $m = |E|$ and $n = |V|$. The goal is to return a subset $S \subseteq V$ that maximizes $\frac{|E(S)|}{|S|}$ where $E(S) = \{\{u, v\} \in E : u, v \in S\}$ is the set of edges with both end points in $S$. DSG has a variety of applications in which dense subgraphs reveal important information about the underlying network such as communities. One can view it as a subroutine in

[*]Supported in part by NSF grant CCF-2028861.
[†]Supported in part by NSF grant CCF-2129816.
[‡]Supported in part by NSF grants CCF-2028861 and CCF-1910149.
We thank Professors Sariel Har-Peled and Ruoyu Sun from the University of Illinois Urbana-Champaign for fruitful discussions in private correspondences.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

an unsupervised clustering procedure. It is a canonical problem in the broad area of dense subgraph discovery which has seen many developments and applications in the past two decades. We point the reader's attention to a (non-exhaustive) list of recent, and some not so recent, important work and the pointers therein [4, 5, 6, 7, 1, 3, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. Constrained versions of DSG such as the densest $k$-subgraph problem DkSG, densest at most $k$-subgraph, and densest at least $k$-subgraph DalkSG are also well-studied and many of these are NP-Hard. See [24, 25, 26, 10, 6, 27] for some positive and negative results on approximation algorithms.

One of the key advantages of DSG is its polynomial-time solvability. The first polynomial-time algorithms were due to Goldberg [4] and Picard and Queuranne [28]. The decision version of DSG is the following: given $G$ and a rational number $\lambda$, is the density in $G$ at least $\lambda$? The algorithms in [4, 28] construct an auxiliary directed flow-network $H$ such that the maximum flow in $H$ allows one to answer the decision version. Binary search over $\lambda$ leads to the final algorithm. Maximum flow is a powerful algorithmic subroutine, however, it is not a practical or scalable algorithm for modern graph data sets with millions and even billions of vertices and edges. For example, authors from [1] noted that the Goldberg's maximum flow algorithm failed on many large scale graphs even though they used a highly optimized maximum flow library. Charikar [5] described a linear programming formulation for DSG that gives an exact solution and has $O(|E|)$ variables and constraints. LP solvers are also unsuitable for large data sets due to memory limitations among others. Charikar [5] described a very simple $\frac{1}{2}$-approximation algorithm for DSG known as the Greedy or Peeling algorithm. The algorithm creates an ordering of the vertices as follows. The first vertex $v_1$ is the one with the smallest degree in $G$ (ties broken arbitrarily). It selects $v_2$ to be the smallest degree vertex in $G - v_1$. Letting $G^i$ be the graph after removing $v_1, v_2, \ldots, v_{i-1}$ (with $G^0 = G$), the algorithm returns the graph among $G^0, \ldots, G^n$ with the highest density. The algorithm can be implemented in $O(m)$ time. The ordering created by the algorithm is the same as the one to compute a $k$-core decomposition of a graph — this is a well-studied graph decomposition procedure with several applications [29] . The simplicity and the efficiency of the Greedy algorithm, and its approximation guarantees, has led to its adoption for a number of other density measures.

Despite Greedy's advantages, its worst-case approximation guarantee is only $\frac{1}{2}$, and is worse for other density measures. The goal is to develop algorithms that obtain a $(1-\varepsilon)$ relative approximation for a given parameter $\varepsilon \in (0, 1)$ while also being scalable to large graphs. One approach to obtain such algorithms is via the dual of Charikar's LP relaxation. It is a mixed packing and covering LP for a given guess of the optimal value. Such LPs can be approximately solved via iterative methods such as the multiplicative weight updates (MWU) or other methods based on convex optimization. Bahmani, Goel and Munagala [30] applied this methodology to obtain an algorithm that yields a $(1-\varepsilon)$-approximation in $\tilde{O}(m/\varepsilon^2)$-time. Boob, Sawlani and Wang [31] described an algorithm that yields a $(1-\varepsilon)$-approximation in $\tilde{O}(m\Delta(G)/\varepsilon)$-time, where $\Delta(G)$ is the maximum degree in $G$. More recently Chekuri, Quanrud and Torres [2] obtained a $(1-\varepsilon)$-approximation in $\tilde{O}(m/\varepsilon)$-time via approximate flow techniques. Some of these nice theoretical developments have not yet led to practically useful algorithms for large scale graphs. In a different direction, Boob *et al.* [1] described a fast iterative algorithm called Greedy++ which builds on the Greedy algorithm and insights from the LP relaxation. It does extremely well in experiments, and the authors conjectured that it yields a $(1-\epsilon)$ relative approximation in $O(\frac{1}{\epsilon^2})$ iterations and each iteration can be implemented in $O(m)$ time. Chekuri *et al.* [2] proved that Greedy++ converges to a $(1-\epsilon)$-approximation in $O(\frac{\Delta(G)}{\lambda^* \epsilon^2})$ iterations where $\lambda^*$ is the optimum density. This gives evidence of the theoretical soundness of the algorithm.

Our goal in this paper is to develop new and scalable algorithms that outperform Greedy++ while having strong theoretical guarantees. In addition, we are interested in an algorithm that finds an approximate *dense subgraph decomposition* which gives information on the density structure of the graph and is different from another such decomposition, namely the $k$-core decomposition. Further, as shown in previous work [32, 33, 2], a dense subgraph decomposition allows one to compactly represent approximate solutions to densest subgraphs of different sizes; in particular the representation yields a 2-approximation for densest at least $k$-subgraph for any given $k$ [32, 2].

**Dense subgraph decomposition.** One can show that every graph $G = (V, E)$ admits a certain structured nested decomposition based on decreasing density. In particular, the vertex set $V$ can partitioned into $S_1, S_2, \ldots, S_k$ such that $S_1$ is the unique maximal densest subgraph in $G$, and $S_i$ is the unique maximal densest set with respect to $S_1 \cup \ldots \cup S_{i-1}$ (a formal definition is deferred to

Section 3). For each vertex $v$ we let $\lambda_v$ denote the density of the set containing $v$. The existence of such a decomposition follows from the supermodularity properties of the function $f(S) = |E(S)|$, and is implicitly known from past work on submodular functions (in particular, the classical result of Fujishige [34]). For graphs this decomposition was rediscovered by Tatti and Gionis [35, 12] under the name of locally-dense decomposition. Tatti showed that one can compute the exact decomposition via $n$ maximum flow computations. Danisch *et al.* [3] showed that computing the $\lambda_v$ values can be cast as solving a quadratic program that extends the LP of Charikar, and applied the Frank-Wolfe algorithm. They showed that their algorithm needs $O(\frac{|E|\Delta(G)}{\epsilon^2})$ iterations (each taking $O(|E|)$ time) to converge to an $\varepsilon$-approximate vector. Their work shows that the Frank-Wolfe algorithm leads to a good approximation to the densest subgraph in a relatively small number of iterations based on experiments. However, they do not describe a systematic way to extract a dense subgraph decomposition from the approximate vector with provable guarantees. See Figure 6.4 in Appendix 6.1 for a table summarizing known results. For DSG-LD, GREEDY++ is not proven to converge to the optimal dense decomposition load vector [4]. The FRANK-WOLFE based algorithm takes $O(\frac{m\Delta(G)}{\epsilon^2})$ iterations, each taking $O(m)$ time to converge to a $(1-\epsilon)$ dense decomposition. Our algorithm requires $O(\frac{\sqrt{m\Delta(G)}}{\epsilon})$ iterations, each taking $O(m)$ time, to converge to a $(1-\epsilon)$ dense decomposition vector. Finally, the multiplicative weight update algorithm takes $O(\frac{m\Delta(G)}{\epsilon^2})$ iterations, each taking $O(m)$ time. We note that the multiplicative weight update algorithm has several variants, and the one we implement is discussed in Section 5.5. Except for GREEDY++ due to the peeling nature of the algorithm, each iteration of the other algorithms are easy to parallelize.

**Densest Supermodular Set.** Efficient solvability of DSG can also be seen via a connection to a more general problem called the densest supermodular set problem, which we refer to as DSS. A real-valued set function $f : 2^V \to \mathbb{R}_+$ is said to be supermodular iff $f(A) + f(B) \leq f(A \cup B) + f(A \cap B)$ for all $A, B \subseteq V$. The goal in DSS is to return $S \subseteq V$ that maximizes $f(S)/|S|$. For any graph $G = (V, E)$, the function $f : 2^V \to \mathbb{R}_+$ defined as $f(S) = |E(S)|$ for each $S \subseteq V$, is known to be supermodular. Hence DSS generalizes DSG. Several algorithms and structural features for DSG are easier to understand via supermodularity, as shown in recent work [2].

*Notation:* For an undirected graph $G = (V, E)$ and $S_1, S_2 \subseteq V$, we use $E(S_1, S_2)$ for the edge set $\{\{u, v\} \in E : u \in S_1, v \in S_2\}$. For a vector $b \in \mathbb{R}^V$, and a subset $S \subseteq V$, we let $b(S)$ denote $\sum_{i \in S} b_i$. We let $ord(E)$ be the set of $2|E|$ edges that are ordered edges $uv$ and $vu$ for each unordered edge $\{u, v\} \in E$. Finally, $m$ and $n$ denote $|E|$ and $|V|$ respectively.

## 2. Technical Contributions

We summarize the main contributions of this paper. **A fast and scalable algorithm based on projections:** We describe an iterative algorithm for DSG and dense subgraph decomposition based on solving a quadratic objective with linear constraints that is derived from the dual of Charikar's LP. Unlike previous work [36] that relied on the Frank-Wolfe method, we use a projection-based approach. The algorithm is extremely simple and highly parallelizable. We show that it converges to an $\varepsilon$-additive approximation in $O(\sqrt{m\Delta(G)}/\varepsilon)$ iterations which is significantly better than the bound for the Frank-Wolfe method. The algorithm scales to extremely large graphs and outperforms existing algorithms in both running time and the approximation quality on almost all data sets. The algorithm and analysis generalizes to hypergraphs.

**Fractional Peeling**: We introduce the idea of fractional peeling to round a fractional solution to the underlying LP/QP relaxation and show its effectiveness in theory and practice. In experiments it significantly outperforms ordering based rounding algorithms considered previously. We use it to obtain a provably approximate dense subgraph decomposition from a fractional solution.

**Connections to DSS:** We explicitly connect the result of Fujishige [34] on the existence of a lexicographically optimal base in a polymatroid with the problem of computing a densest decomposition of a supermodular function. We show that the a vertex load vector is feasible for the dual of Charikar's LP for DSG iff it is feasible for the base contrapolymatroid associated with the supermodular func-

[4] Very recently the authors of this paper were able to prove the convergence of GREEDY++ to the dense decomposition. A proof will appear in a followup manuscript.

tion $f(S) = |E(S)|$ induced by the graph. These connections clarify past results such as the one in [36, 2] and allow us to compare Frank-Wolfe versus projection based methods.

**Experimental evaluation:** We compare five algorithms for DSG, namely GREEDY++, FRANK-WOLFE, MWU, FISTA, and FISTA-PARALLEL a parallel version of FISTA. We test the algorithms on real world and synthetic data sets and show the effectiveness of FISTA when combined with fractional peeling.

To make the paper self contained we provide all proofs in the appendix, even for results implicitly contained in past literature. We focus on DSG (unweighted), however, the ideas generalize to hypergraphs and weighted graphs, and we leave a more in-depth exploration for future work.

## 3. Densest Subgraph Decomposition and Supermodularity

Let $G = (V, E)$ be an undirected graph. Each graph admits a unique nested decomposition of decreasing densities. This property is more transparently seen via supermodularity. Let $f : 2^V \to \mathbb{R}_+$ be a non-negative supermodular set function[5]. For example, fix $f(S) = |E(S)|$ which is supermodular. Supermodularity implies that there is a unique inclusion-wise *maximal* densest set $S_1$. It is convenient to describe the decomposition in an algorithmic fashion. The algorithm calculates the maximal set $S_1 \subseteq V_0 = V$ that maximizes $\frac{|E(S)|}{|S|}$ in $G$ with density $\lambda_1 = \lambda^* = \frac{|E(S_1)|}{|S_1|}$. For DSS this corresponds to finding the unique maximal set $S_1$ that achieves the maximum density $\max_S f(S)/|S|$.

In iteration $i$, letting $U_{i-1} = \cup_{1 \le j < i} S_j$, it calculates the maximal set $S_i \subseteq V_i = V - U_{i-1}$ that maximizes $(|E(S \cup U_{i-1}| - |E(U_{i-1})|)/|S|$. The algorithm is described formally in Algorithm 1.

**Algorithm 1** Dense graph decomposition (left) and dense supermodular set decomposition (right)

$U_0 \leftarrow \emptyset, V_0 \leftarrow V, k \leftarrow 0$
**while** $V_k \neq \emptyset$ **do**
  $k \leftarrow k + 1$
  $S_k \leftarrow \arg\max_{\substack{S \subseteq V_{k-1} \\ S \text{ maximal}}} \frac{|E(S)| + |E(S, U_{k-1})|}{|S|}$
  $U_k \leftarrow U_{k-1} \cup S_k, V_k \leftarrow V_{k-1} - S_k$
**return** $S_1, ..., S_k$

$U_0 \leftarrow \emptyset, V_0 \leftarrow V, k \leftarrow 0$
**while** $V_k \neq \emptyset$ **do**
  $k \leftarrow k + 1$
  $S_k \leftarrow \arg\max_{\substack{S \subseteq V_{k-1} \\ S \text{ maximal}}} \frac{f(S \cup U_{k-1}) - f(U_{k-1})}{|S|}$
  $U_k \leftarrow U_{k-1} \cup S_k, V_k \leftarrow V_{k-1} - S_k$
**return** $S_1, ..., S_k$

The algorithm outputs a partition of $V$ into $S_1, ..., S_k$, where $k$ is the *dense decomposition depth* of $G$ (or any supermodular function $f$). With each set $S_i$, we associate a density defined as $\lambda_i = (|E(S_i)| + |E(S_i, U_{i-1})|)/|S_i|$ (or in the case of $f, \lambda_i = (f(S_i \cup U_{i-1}) - f(U_{i-1}))/|S_i|$). In addition, for any $u \in S_i$, we say the density of $u, \lambda_u = \lambda_i$. Refer to Lemma 6.2 (Appendix 6.1) for some basic properties of the dense decomposition. Specifically, the densities monotonically decrease, that is, $\lambda_1 > \lambda_2 > ... > \lambda_k$.

## 4. LP and QP for DSG and Decomposition

Charikar's [5] exact LP relaxation for DSG 4.1, and its dual 4.2 (in a slightly modified form), are given below. The primal has a variable $y_u$ for each vertex $u \in V$ which indicates whether $u$ is chosen in the densest set. For each edge $e = \{u, v\} \in E$ there is a variable $z_e$ to indicate whether it is chosen. An edge $\{u, v\}$ can be chosen only if both $u$ and $v$ are chosen which explains the constraints linking $z_e$ to $y_u, y_v$. The LP normalizes $\sum_u y_u$ to 1 for linearity, and maximizes chosen edges.

[5]We restrict attention to non-negative supermodular set functions that satisfy $f(\emptyset) = 0$ and this automatically also implies that they are monotone, that is, $f(A) \le f(B)$ for $A \subset B$

$$\begin{aligned}
\max \sum_{e\in E} z_e & \qquad (4.1)\\
\text{s.t. } z_e &\le y_u \qquad \forall e=\{u,v\}\in E\\
z_e &\le y_v \qquad \forall e=\{u,v\}\in E\\
\sum_{u\in V} y_u &\le 1\\
z_e, y_u &\ge 0
\end{aligned}$$

$$\begin{aligned}
\min \max_{u\in V} b_u & \qquad (4.2)\\
\text{s.t. } \sum_{v\in\delta(u)} x_{uv} &= b_u \qquad \forall u\in G\\
x_{uv}+x_{vu} &= 1 \qquad \forall\{u,v\}\in E\\
x_{uv}, x_{vu}, b_u &\ge 0
\end{aligned}$$

The dual can be viewed as orienting each edge $\{u,v\}$ fractionally towards $u$ and $v$. The orientation induces loads on the vertices and the goal is to find an orientation that minimizes the maximum load over vertices. It is not hard to see that LP 4.2 is actually the same as the dual of 4.1; the variables $b_u$ can be replaced by a single variable $b$. The optimum value of LP 4.1 and LP 4.2 is the density $\lambda^* = \lambda_1$. However, we show that the $b_u$ values have additional information.

**Theorem 4.1.** *Let $S_1, ..., S_k$ and $\lambda_1, ..., \lambda_k$ be the densest subgraph decomposition of a graph $G$, and for any $u \in S_i$, let $\lambda_u = \lambda_i$. There is an optimal solution $(x^*, b^*)$ to 4.2 such that $b^*_u = \lambda_u$.*

We give two proofs, one a direct proof using Rado's theorem [37] in Appendix 6.1, and a second by relating 4.2 to another relaxation and using a known result of Fujishige that is captured in Theorem 4.3. We say a vector $a \in \mathbb{R}^n$ is *lexicographically* smaller than vector $b \in \mathbb{R}^n$ if the sorted vector $a$ (in descending order) is lexicographically smaller than the sorted vector $b$. Theorem 4.1 suggests that there exists a lexicographically least optimal solution to LP 4.2 where each vertex load $b_u$ is precisely $\lambda_u$. To obtain the lexicographic solution, it suffices to introduce some strict convexity into the objective. Let $P(x, b)$ denote the polyhedron defined by the constraints in 4.2. Consider the quadratic program 4.3:

$$\min \sum_{u\in V} b_u^2 \text{ such that } (x, b) \in P(x, b) \qquad (4.3)$$

The theorem below was shown by Danisch *et al.* [3] but as we will show later, this is a special case of Fujishige's result [34] from 1980 via Theorem 4.3.

**Theorem 4.2.** *There is a unique optimum solution $b^*$ to 4.3 and for each $u \in V(G), b^*_u = \lambda_u$.*

### 4.1. LP and QP for DSS and densest decomposition

We will now generalize Theorem 4.1 and Theorem 4.2 for the DSS problem. We start by recapping the notion of a contrapolymatroid (see [38]) which is the relevant notion for supermodular functions (as polymatroids are for submodular functions). For a normalized non-negative supermodular function $f : 2^V \to \mathbb{R}_+$, the contrapolymatroid with it is the following polyhedron

$$P_f = \{x \in \mathbb{R}^V \mid x \ge 0,\ x(S) \ge f(S) \text{ for all } S \subseteq V\} \qquad (4.4)$$

A vector $x \in P_f$ is a base if $x(V) = f(V)$. The base contrapolymatroid is defined as:

$$B_f = \{x \in \mathbb{R}^V \mid x \ge 0,\ x(S) \ge f(S) \text{ for all } S \subseteq V,\ x(V) = f(V)\} \qquad (4.5)$$

Now consider problem 4.6 where we are given a monotone non-negative supermodular function $f : 2^V \to \mathbb{R}_+$ and want to find the lexicographically minimal solution $b^*$ for Problem 4.6. We will show that we can do this by instead solving Problem 4.7:

$$\begin{aligned}
&\text{minimize} && \max_{u\in V} b_u\\
&\text{subject to} && b \in B_f
\end{aligned} \qquad (4.6)$$

$$\begin{aligned}
&\text{minimize} && \sum_{u\in V} b_u^2\\
&\text{subject to} && b \in B_f
\end{aligned} \qquad (4.7)$$

**Theorem 4.3.** *Let $S_1, ..., S_k$ and $\lambda_1, ..., \lambda_k$ be the densest supermodular set decomposition of $f$, and for any $u \in S_i$, let $\lambda_u = \lambda_i$. Then the following must hold*

1. *The solution $b$ where $b_u = \lambda_u$ is feasible in the base polytope (i.e $b \in B_f$)*

2. *The lexicographically minimal solution $b^*$ for Problem 4.6 satisfies $b_u = \lambda_u$*
3. *The optimal solution of 4.7, $b^*$ is unique, and for each $u \in V, b^*_u = \lambda_u$*

We give a self-contained proof in Appendix 6.1 and note that the theorem is implied by (essentially equivalent to) Fujishige's result [34] on the existence of a lexicographically optimal base of a polymatroid with respect to a weight vector. LP 4.2 and LP 4.6 can be related via the following theorem.

**Theorem 4.4.** *Consider a graph $G = (V, E)$ and the associated supermodular function $f : 2^V \to \mathbb{R}_+$ where $f(S) = |E(S)|$. A vector $b \in B_f$ if and only if there is an $x \in \mathbb{R}^{ord(E)}, x \geq 0$ such that the pair $(x, b)$ satisfy the constraints of the LP 4.2.*

See Appendix 6.1 for a proof. This implies that Theorem 4.2 is a corollary of Theorem 4.3.

## 5. Solving the Quadratic Program using proximal projections, and rounding

In this section we show how to approximately solve 4.3. We let $f(x) = \sum_{u \in V} \left( \sum_{v \in \delta(u)} x_{uv} \right)^2$. Note that this is simply the objective function rewritten in terms of $x$. Similarly, let $h(x)$ be an indicator function where $h(x) = 0$ if $x_{uv} \geq 0$ and $x_{uv} + x_{vu} = 1, \forall (u, v) \in E$ and $+\infty$ otherwise. Then Problem 4.3 can be rewritten as minimizing the unconstrained objective $f(x) + h(x)$ for $x \in \mathbb{R}^{2m}$. We will use a proximal gradient method to solve the problem. For that, we need two lemmas whose proofs are in Appendix 6.2.

**Lemma 5.1.** *The Lipschitz constant of $\nabla f$ is at most $2\Delta(G)$ where $\Delta(G)$ is the max degree of $G$.*

**Lemma 5.2.** *Let $x \in \mathbb{R}^{2m}$. Define the* proximal mapping *$prox_h(x)$ as the point $p \in \mathbb{R}^{2m}$ that minimizes $\|p - x\|^2$ such that $h(p) = 0$. Then we have that for $u < v$, $prox_h(x)_{uv} = \frac{x_{uv} - x_{vu} + 1}{2}$ if $|x_{uv} - x_{vu}| \leq 1$, $prox_h(x)_{uv} = 1$ if $x_{uv} - x_{vu} > 1$, and $prox_h(x)_{uv} = 0$ if $x_{uv} - x_{vu} < -1$. Additionally, $prox_h(x)_{vu} = 1 - prox_h(x)_{uv}$.*

We present the algorithm now. We are interested in the unconstrained optimization problem of minimizing $f(x) + h(x)$ where $f$ is convex, and $h$ has an easy to compute proximal mapping. This type of problem can be solved using proximal gradient methods. From a high level, the (basic) algorithm is described Algorithm 2. At any iteration $t$ it has a guess for the minimizer $x^{(t)}$. It then calculates the gradient of $f$ and moves slightly against it. However, since this might make the new guess infeasible, it uses the proximal mapping to project the new guess to a feasible solution.

**Algorithm 2** Basic Proximal Gradient Method (left) and accelerated FISTA (right)

Input: $f$ and $h$ with $\text{prox}_h(x)$, learning rate $\alpha$ and iterations $T$. Initialize $x^{(0)}$ with $h(x^{(0)}) = 0$

**for** $t \in [1, T]$ **do**
  $x^{(t)} = \text{prox}_h(x^{(t-1)} - \alpha \nabla f(x^{(t-1)}))$
**return** $x^{(T)}$

$y^{(0)} = x^{(0)}$
**for** $t \in [1, T]$ **do**
  $x^{(t)} = \text{prox}_h(y^{(t-1)} - \alpha \nabla f(y^{(k-1)}))$
  $y^{(t)} = x^{(t)} + \frac{t-1}{t+2}(x^{(t)} - x^{(t-1)})$
**return** $x^{(T)}$

While the basic proximal gradient method works, we will use an even faster (both theoretically and practically) version known as the accelerated proximal gradient method which incorporates Nesterov-like momentum terms [39] in the projection step. It has other names in the literature such as proximal gradient method with extrapolation and FISTA [40]. The algorithm is outlined in Algorithm 2. We have the following known result on the FISTA algorithm.

**Lemma 5.3.** *[40]. Let $x^*$ be the minimizer of $f$. Suppose that the learning rate satisfies $\alpha \leq \frac{1}{L(f)}$ where $L(f)$ is the Lipschitz constant of $\nabla f$. Then after $k$ iterations, $f(x^{(k)}) - f(x^*) \leq \frac{2\left\|x^{(0)} - x^*\right\|^2}{\alpha k^2}$.*

In our case, we bounded the Lipschitz constant to $2\Delta(G)$ and an easy upper bound on $\left\|x^{(0)} - x^*\right\|^2$ is $2m = 2|E(G)|$. Combining this with a learning rate of $\frac{1}{2\Delta(G)}$, we get the following result

**Lemma 5.4.** *If FISTA is applied to the objective $f(x) + h(x)$ as previously defined, then in the $k$th iteration, we have $f(x^{(k)}) - f(x^*) \leq \frac{8\Delta(G)m}{k^2}$.*

The final FISTA algorithm for our problem is shown in Algorithm 3 in Appendix 6.2 with the correct gradient computation, projective mappings, and iteration updates. Lastly, to get a good additive approximation on each $\lambda_u$, we have the following result whose proof is in Appendix 6.3

**Theorem 5.5.** *Let $b_u^* = \lambda_u$. After $t = O(\frac{\sqrt{\Delta(G)m}}{\epsilon})$ iterations of FISTA, we must have $\left\|b^{(t)} - b^*\right\| \leq \epsilon$. This implies $\left|b_u^{(t)} - \lambda_u\right| \leq \epsilon$ for all $u \in V$*

### 5.1. Fractional Peeling.

Given an approximate solution $(b, x)$ to Problem 4.3, how do we round it to obtain a good densest decomposition? The natural approach is to sort the vertices in non-increasing values of $b$ and take suffixes. This is used in [36]. This is an exact rounding algorithm when $b$ is an *optimum* solution, however one can construct approximate solutions for which this rounding is not ideal. We describe a peeling algorithm inspired by GREEDY++ [1] that takes advantage of the auxiliary information provided by the vector $x$. First, set $b' = b$ and $G^{(0)} = G$. In iteration, $t \geq 1$, peel the vertex $u$ with minimum current load $b'_u$. Then, for each $v \in \delta_{G^{t-1}}(u)$, set $b'_v \leftarrow b'_v - x_{vu}$ (i.e subtract the fractional value of $x_{vu}$ from $v$'s load). Update $G^t = G^{t-1} - u$. Repeat this process to obtain $G^{(0)}, ..., G^{(n)}$. Finally, return a graph $G^{(i)}$ with maximum density. The algorithm can be implemented in $O(m + n \log n)$ using a Fibonacci heap. We will refer to this process as *fractional peeling*. We show both theoretically and experimentally that fractional peeling leads to better algorithms.

### 5.2. $\epsilon$-dense local decomposition

We show that fractional peeling can be used on an approximate solution to 4.3 to obtain an approximate dense decomposition with a theoretical guarantee. This is in contrast to previous work in [36]. We define a strong notion of approximate decomposition. Given a solution $(b, x)$ to 4.3, we say $\hat{b}$ is an *$\epsilon$-load-vector* if $\left\|\hat{b} - b^*\right\| \leq \epsilon$. Given a partition of the vertices $T_1, ..., T_r$, we say the partition is an *$\epsilon$-approximate dense decomposition* to $S_1, ..., S_k$ (the true dense decomposition) if

$$u \in S_i, u \in T_h \implies \frac{|E(T_h)| + |E(T_h, \cup_{j<h} T_j)|}{|T_h|} \geq \frac{|E(S_i)| + |E(S_i, \cup_{j<i} S_j)|}{|S_i|} - \epsilon$$

Intuitively, what this says is that every $u \in V$ belongs to a set $T_h$ that has a density that is not much less than $\lambda_u$.

**Theorem 5.6.** *Given an $\epsilon$ load vector $b$ and an edge vector $x$ that induces $b$, we can calculate an $\epsilon(\sqrt{n} + 1)$-approximate dense decomposition in $\tilde{O}(mn)$ time.*

The proof is in Appendix 6.4. Note that the notion of error is additive and holds for every vertex $u$. Although the $\sqrt{n}$-factor is large, the analysis shows that one can usually obtain a much stronger bound. Qualitatively it shows that fractional peeling leads to good dense decomposition as $\varepsilon \to 0$.

### 5.3. Projections vs Frank-Wolfe vs MWU for DSG and DSS

The Frank-Wolfe method is natural to apply to solve 4.7 since each iteration requires optimizing a linear objective over the base contrapolymatroid $B_f$; this is easy and fast via the greedy algorithm, as shown originally by Edmonds in the context of polymatroids (see [38]). Danisch *et al.* work with 4.3. However, as we observed earlier, 4.3 is a compact way to represent the associated base polyhedron, and hence the algorithm in [36] is the same as the Frank-Wolfe algorithm applied to 4.7. The Fujishige-Wolfe minimum norm point algorithm for submodular function minimization is related but is based on Wolfe's method (see [41, 42]). The fact that the optimum point in the base polytope has additional information was already pointed out in [34], and also explored in the context of size-constrained submodular function minimization by Nagano *et al.* [33]. The advantage of proximal gradient methods such as FISTA is their faster convergence rates when compared to the Frank-Wolfe method, although each iteration requires a projection oracle for $B_f$. Our algorithm is

based on the observation that there is an $O(m)$-time projection oracle for DSG due to the alternate characterization of $B_f$ via the edge variable LP 4.2. This same observation holds for hypergraphs and we obtain a fast algorithm for them. We outline the formal details in Appendix 6.5. The multiplicative weight update (MWU) method is another broad methodology for solving linear and convex programs and there are several variants. One variant of MWU for solving 4.2 can be interpreted as a Frank-Wolfe method with a certain convex objective and with a certain step size; each iteration of this algorithm is again a greedy optimization over $B_f$. We provide more details in the Appendix 6.7.

### 5.4. A theoretically fast algorithm for approximate load vector via min-cost flow

While Algorithm 3 (detailed in Appendix 6.2) works extremely well in practice, there is room to improve the theoretical run time. Each iteration takes $O(m)$ time and can easily be parallelized. However, in the worst-case it would need $O(m\sqrt{\Delta(G)m}/\epsilon)$ time to get an additive $\epsilon$-approximation. Can we obtain a theoretically faster algorithm? In Appendix 6.6 we describe a reduction to computing *exact* minimum quadratic-cost flow in a directed network. Very recently, in a breakthrough, [43] developed near-linear time algorithm for min-cost flow and also convex-cost flows. Via their algorithm we obtain an $O(m^{1+o(1)})$ time algorithm to get an optimal dense decomposition vector. We hope that this theoretical result will inspire the development of new algorithms that are provably faster in theory while also having good empirical performance.

### 5.5. Frank-Wolfe and MWU

The Frank-Wolfe algorithms is part of the broader family of algorithms referred to as conditional linear gradient methods. These algorithms approximately minimize a convex function $f(x)$ over a polytope $P$ on which one can do efficient linear optimization. Generally speaking, each iteration $t$, these algorithms start with the current point $x_{t-1} \in P$ and solve the linear optimization problem $\min_{y \in P}(\nabla f(x_{t-1})^T y$. Let $p_t$ be optimum solution to this; the algorithm sets $x_t$ to be a convex combination of the current point and this vertex, $x_t = \alpha_t p_t + (1 - \alpha_t)x_{t-1}$. The parameter $\alpha_t$, called the step size, controls the convergence rate. The Frank-Wolfe method often refers to the specific step size of $\alpha_t = 2/(t + 2)$ [44]. Other possibilities including $\alpha_t = 1/t$, which implies that each $x_t$ will be a uniform combination of all previous vertices $p_1, p_2, \ldots, p_t$. This gives a slower rate of convergence proportional to $\ln t/t$, rather than $1/t$ (see for instance [45]).

An alternative continuous approach, popular for solving obtaining multiplicative approximations to LP's, is the *multiplicative weight update* framework [46]. There are several ways to leverage the MWU framework for the densest subgraph problem (e.g., [30, 1, 2]). We test one variation that applies the MWU framework to solve LP 4.2 that minimizes the maximum load. In this case, the MWU framework implicitly tries to minimize a potential function that exponentiates these loads and sums them together. More formally, for a parameter $\eta > 0$, consider the problem $\min \frac{1}{\eta} \ln(\sum_{u \in V} \exp(\eta b_u))$ over $b \in B_f$. As $\eta \to \infty$ one can see that the optimum solution to this problem converges to the minimum load vector $b^*$. Each iteration of the MWU framework involves solving a simpler linear optimization problem induced by the gradient of this potential. In the specific context of DSS the linear optimization problem corresponds to the greedy algorithm over $B_f$. The greedy algorithm only depends on the ordering of $V$ based on the current loads (and not their specific values). For this reason, when the MWU method is applied with a fixed step size (in the so-called "width-dependent MWU framework"), the MWU algorithm ends up solving the exact same sequence of optimization problems, in the exact same way, as the conditional linear gradient method with step size $\alpha_t = 1/t$ would for the objective $\sum_u b_u^2$! Thus the parameter $\eta$ does not a play a role in this specific case, and it also follows that the MWU algorithm converges to an $\epsilon$-approximate load vector, although at a slightly lower rate than the Frank-Wolfe method.

## 6. Experimental Evaluation

**Datasets & Implementation Details.** We ran experiments on 7 real world datasets (6 from the SNAP database [47], and 1 from [48]), and one tailored synthetic dataset (used for clarifying an important difference between all algorithms) for a total of 8 datasets. The dataset information is summarized in the table below. The CLOSE-CLIQUES dataset consists of the complete bipartite graph $K_{d,D}$ for $d = 30$ and $D = 2000$, and 20 copies of the complete graph $K_h$ where $h = 60$.

| Dataset | Vertices | Edges | Source |
|---|---|---|---|
| cit-Patents | 3,774,768 | 16,518,947 | [47] |
| com-Amazon | 334,863 | 925,872 | [47] |
| orkut | 3,072,441 | 117,185,083 | [48] |
| roadNet-PA | 1,088,092 | 1,541,898 | [47] |

| Dataset | Vertices | Edges | Source |
|---|---|---|---|
| roadNet-CA | 1,965,206 | 2,766,607 | [47] |
| Close-Cliques | 3,230 | 95,400 | Synthetic |
| dblp-author | 317,080 | 1,049,866 | [47] |
| wiki-topcats | 1,791,489 | 28,508,141 | [47] |

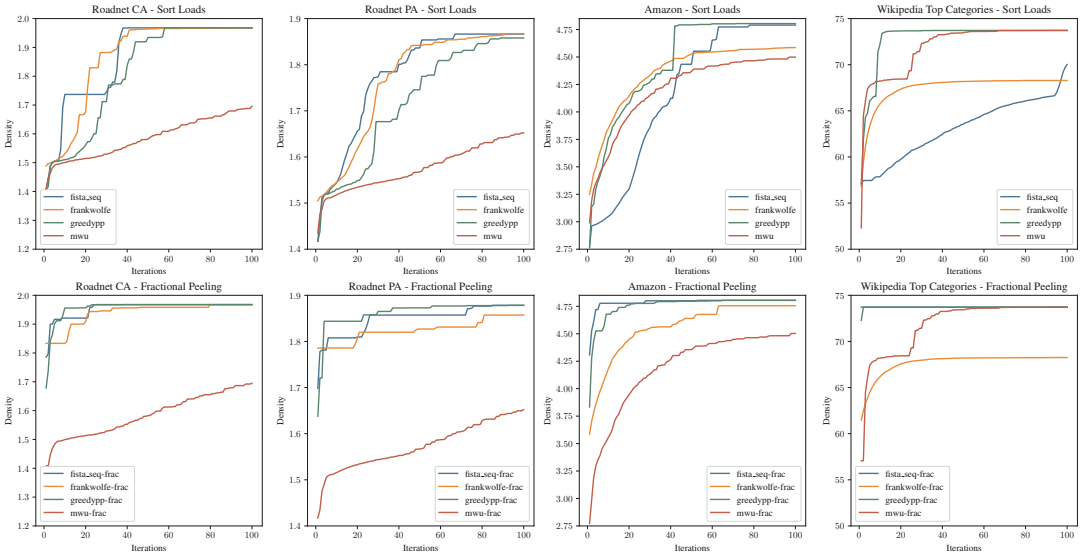


Figure 6.1: Density based on sorting loads vs fractional peeling.

We tested 5 algorithms to approximate DSG: The FISTA based algorithm (sequential, no parallelism), FISTA-PARALLEL, GREEDY++ from [1], the FRANK-WOLFE based algorithm from [3], and the MWU algorithm respectively. See supplementary section for implementation details of the 5 algorithms. Also see Appendix 6.7 for details on the specific variant of MWU we tested. All algorithms were implemented in C++17 and were compiled with O3 and UNROLL-LOOPS optimizations. The implementations of Algorithms FISTA, FISTA-PARALLEL, FRANK-WOLFE, and MWU are the authors' implementations, but we used the original implementation for GREEDY++ [1] as it was extremely well optimized. We modified their implementation minimally to log basic information needed for the evaluation. FISTA-PARALLEL used Open MPI [49] for parallelism. We ran our experiments on a Slurm-based university campus cluster. For all machines, we requested 1 node and 16 cores per experiment. The nodes had 64 GB of RAM and Xeon PHI 5100 CPUs.

**Densities**. In the first experiment, we ran all the algorithms for a number of iterations, and monitored the maximum density reached by the algorithm until iteration $t$. The density in each iteration was calculated by statically sorting the vertices by the value of their load vector, and peeling in that order and returning the maximum density subgraph in that iteration. The result is shown in Figure 6.1 (Top 4 plots, we only show 4 datasets, remaining plots in Appendix 6.7). Specifically, this does not use the fractional peeling technique we discussed. We also plot the maximum density reached by using fractional peeling instead (Bottom 4 plots in 6.1, we show 4 datasets, remaining plots in Appendix 6.7).

When the peeling order is changed from a static peel (based on the load vector value) to a fractional peel, the results improve dramatically. See Figure 6.1 and Appendix 6.7. In all algorithms (except MWU), fractional peeling leads to a dramatic speedup in terms of the number of iterations needed to get the densest subgraph on almost all datasets. Fractional peeling provides little benefit to MWU. We can observe that for all the real world datasets, GREEDY++ and FISTA are extremely competitive and reach near-optimal densities in just a few iterations. Meanwhile, the FRANK-WOLFE based algorithm lagged behind in the beginning, but steadily made progress towards the maximum density in later iterations. MWU strongly lagged behind all algorithms in several datasets. The only excep-

tion is for the CLOSE-CLIQUES dataset (see Appendix 6.7). The component in the densest subgraph has density $\approx 29.5566$. Meanwhile, the remaining components have density $\binom{60}{2}/60 = 29.5$. This is a similar but nonidentical to the example from [1]. Note that for $K_{d,D}$ and $r$ copies of $K_{2d}$ (for sufficiently large $r, D$), $\exists \epsilon$ where GREEDY++ requires $\Theta(\frac{1}{\epsilon})$ iterations to converge to a $(1-\epsilon)$ approximation for DSG in the worst case. MWU and FRANK-WOLFE did better for this synthetic example where the densities are very close and it is worth understanding in more detail.

**Wall clock time and run-time per iteration.** In the second experiment we examine the wall-clock time and time per iteration. We study the maximum density reached by an algorithm after $T$ seconds of wall-clock time (cumulative time of all iterations). Figure 6.13 (Appendix 6.7) and last 4 plots of Figure 6.2 shows the result. As can be seen, FISTA-PARALLEL and FISTA (sequential) finds the maximum density on almost all datasets in the least wall-clock time (albeit sequential FISTA and FRANK-WOLFE are close runners if we only restrict to non-parallel implementations). On the other hand, MWU performed poorly in wall-clock time, often taking orders of magnitude longer than FISTA or GREEDY++ to find suboptimal dense subgraphs. Overall, the number of iterations of FISTA is the lowest, and each iteration is fast and can be parallelized, so it is the best performer.

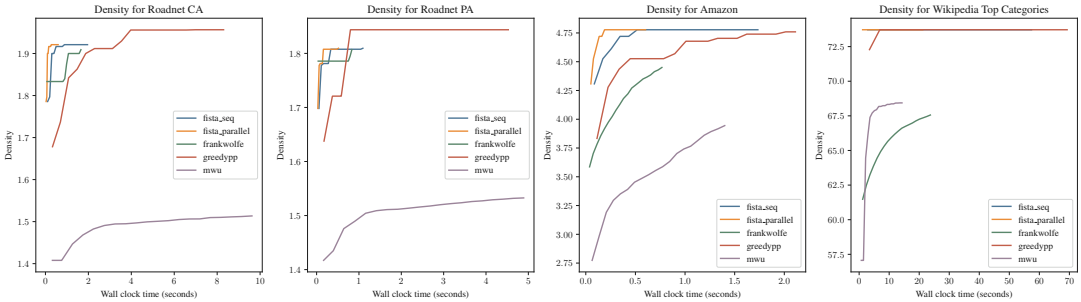


Figure 6.2: Wall clock time of algorithms.

We also considered the run time per iteration for the algorithms. Figure 6.9 (appendix) shows the result. The per iteration time depends on several factors. Although all algorithms have linear or near-linear in $m$ running time, the specific data structure used and cache performance can have substantial impact. Overall, FRANK-WOLFE (due to its simple implementation) and FISTA-PARALLEL had the best per-iteration performance. The average speedup *per iteration* from FISTA-PARALLEL over GREEDY++ was roughly 5 fold. Note that due to the peeling nature of GREEDY++, it cannot be parallelized like FISTA.

**Convergence to optimal load vector.** Recall that we are minimizing $\sum_u b_u^2$, and hence the norm $\left\|b^{(t)}\right\|$ gives a proxy for the convergence of the errors in $b^{(t)}$. In this experiment, we plotted the norm of the vector $b^{(t)}$ for each algorithm in each dataset. The result is shown in Figure 6.10 (Appendix 6.7). GREEDY++ does very well in the first few iterations, but slows down in its improvement on $\left\|b^{(t)}\right\|$ as the number of iterations increases. In comparison, FISTA eventually surpasses GREEDY++ and reduces the error at a faster rate than GREEDY++. In comparison, FRANK-WOLFE and MWU start with a substantial error in the $b$ vector but quickly reduce it, however, their error was always worse that both GREEDY++ and FISTA in all iterations even when left to run 200 iterations.

**Conclusion.** We introduced a new iterative algorithm for the densest subgraph and densest decomposition problems. We also described a new fractional peeling technique which has strong empirical performance as well as theoretical guarantees and showed experimentally how it improved almost all existing algorithms compared to static load sorting. The new algorithm is scalable and simple, and can be applied to graphs with hundreds of millions of edges. Our experiments support the theory established on the utility of the new algorithm and fractional peeling. Our work also adds value via a detailed comparison of the practical performance of existing algorithms and the new algorithm. A few limitations remain for the paper. First, reducing the $\sqrt{n}$ dependency in Theorem 5.6 is important, even if fractional peeling shows strong experimental bounds. Finally, we need to evaluate empirically the quality of the decomposition achieved by algorithms (with and without fractional peeling), and also algorithms for approximating at least $k$ DSG and related problems via the decomposition.

## References


[1] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. *Flowless: Extracting Densest Subgraphs Without Flow Computations*, page 573–583. Association for Computing Machinery, New York, NY, USA, 2020.

[2] Chandra Chekuri, Kent Quanrud, and Manuel R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1531–1555.

[3] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. Large scale density-friendly graph decomposition via convex programming. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 233–242, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[4] A. V. Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD-84-171, EECS Department, University of California, Berkeley, 1984.

[5] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In Klaus Jansen and Samir Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, pages 84–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[6] Samir Khuller and Barna Saha. On finding dense subgraphs. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, pages 597–608, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V.S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1051–1066, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Charalampos Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1122–1132, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[9] Bintao Sun, Maximilien Danisch, T-H. Hubert Chan, and Mauro Sozio. Kclist++: A simple algorithm for finding k-clique densest subgraphs in large graphs. *Proc. VLDB Endow.*, 13(10):1628–1640, jun 2020.

[10] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In Konstantin Avrachenkov, Debora Donato, and Nelly Litvak, editors, *Algorithms and Models for the Web-Graph*, pages 25–37, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[11] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 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*, KDD '13, page 104–112, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Nikolaj Tatti. Density-friendly graph decomposition. *ACM Trans. Knowl. Discov. Data*, 13(5), sep 2019.

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

[14] Francesco Bonchi, David García-Soriano, Atsushi Miyauchi, and Charalampos E. Tsourakakis. Finding densest k-connected subgraphs. *Discrete Appl. Math.*, 305(C):34–47, dec 2022.

[15] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 300–310, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[16] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. Densest subgraph in dynamic graph streams. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald T. Sannella, editors, *Mathematical Foundations of Computer Science 2015*, pages 472–482, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[17] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 173–182, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Polina Rozenshtein, Nikolaj Tatti, and Aristides Gionis. Discovering dynamic communities in interaction networks. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 678–693, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[19] Oana Denisa Balalau, Francesco Bonchi, T-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. Finding subgraphs with maximum total density and limited overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, page 379–388, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Yuko Kuroki, Atsushi Miyauchi, Junya Honda, and Masashi Sugiyama. Online dense subgraph discovery via blurred-graph feedback. In *ICML*, 2020.

[21] Albert Angel, Nikos Sarkas, Nick Koudas, and Divesh Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *Proc. VLDB Endow.*, 5(6):574–585, feb 2012.

[22] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Corescope: Graph mining using k-core analysis — patterns, anomalies and algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 469–478, 2016.

[23] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. Flowscope: Spotting money laundering based on graphs. In *AAAI*, 2020.

[24] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. Detecting high log-densities – an O(n 1/4) approximation for densest k-subgraph.

[25] U. Feige, D. Peleg, and G. Kortsarz. The dense k -subgraph problem. *Algorithmica*, 29(3):410–421, mar 2001.

[26] Subhash Khot. Ruling out ptas for graph min-bisection, dense $k-$subgraph, and bipartite clique. *SIAM Journal on Computing*, 36(4):1025–1071, 2006.

[27] Pasin Manurangsi. Inapproximability of maximum biclique problems, minimum k-cut and densest at-least-k-subgraph from the small set expansion hypothesis. *ArXiv*, abs/1705.03581, 2018.

[28] Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982.

[29] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.

[30] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. Efficient primal-dual graph algorithms for mapreduce. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 59–78. Springer, 2014.

[31] Digvijay Boob, Saurabh Sawlani, and Di Wang. Faster width-dependent algorithm for mixed packing and covering lps. *Advances in Neural Information Processing Systems 32 (NIPS 2019)*, 2019.

[32] Samir Khuller and Barna Saha. On finding dense subgraphs. In *International Colloquium on Automata, Languages, and Programming*, pages 597–608. Springer, 2009.

[33] Kiyohito Nagano, Yoshinobu Kawahara, and Kazuyuki Aihara. Size-constrained submodular minimization through minimum norm base. In *ICML*, 2011.

[34] Satoru Fujishige. Lexicographically optimal base of a polymatroid with respect to a weight vector. *Mathematics of Operations Research*, 5(2):186–196, 1980.

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

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

[37] R. Rado. Theorems on linear combinatorial topology and general measure. *Annals of Mathematics*, 44(2):228–270, 1943.

[38] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

[39] Yurii Nesterov. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

[40] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[41] Philip Wolfe. Finding the nearest point in a polytope. *Mathematical Programming*, 11(1):128–149, 1976.

[42] Satoru Fujishige and Shigueo Isotani. A submodular function minimization algorithm based on the minimum-norm base. *Pacific Journal of Optimization*, 7(1):3–17, 2011.

[43] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time, 2022.

[44] Martin Jaggi. Revisiting frank-wolfe: Projection-free sparse convex optimization. In *International Conference on Machine Learning*, pages 427–435. PMLR, 2013.

[45] Yurii Nesterov et al. *Lectures on convex optimization*, volume 137. Springer, 2018.

[46] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of computing*, 8(1):121–164, 2012.

[47] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[48] Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, page 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery.

[49] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open mpi: A flexible high performance mpi. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[50] Satoru Fujishige. Theory of principal partitions revisited. In *Research Trends in Combinatorial Optimization*, pages 127–162. Springer, 2009.

[51] Weiran Wang and Miguel Á. Carreira-Perpiñán. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application, 2013.

[52] Franck Iutzeler and Laurent Condat. Distributed projection on the simplex and $\ell_1$ ball via admm and gossip. *IEEE Signal Processing Letters*, 25(11):1650–1654, 2018.

## Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] In the experimental discussion, we describe the limitations of different algorithms (including ours) on the datasets
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Work is mainly theoretical with an applied flavor and doesn't involve inherent potential societal impacts.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] All proofs are included in Appendix
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Code is in supplemental section, Data available online (not our data to include)
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] The algorithm isn't randomized, no random seed or error plots applicable.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] Code for Greedy++ was cited appropriately together with used libraries. All the sources data were also cited.
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] No personal data of people was used
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Personal data is not used
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] Crowdsourcing is not used
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] Crowdsourcing is not used
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] Crowdsourcing is not used

# Appendix

### 6.1. Proofs for Section 4

Before we prove Theorem 4.1, we first recap the $b$-transportation problem. Given a bipartite graph $B(V', E')$, and a required load value $b_u \geq 0, \forall u \in V'$, a *$b$-transportation* is an assignment $x_{uv} \geq 0$ to the edges of the bipartite graph such that $\sum_{v \in \delta(u)} x_{uv} = b_u$ for all $u \in V'$. Hence, it can be thought of as a fractional $b-$matching problem. Rado characterized the existence of a $b-$transportation in bipartite graphs using the following result

**Theorem 6.1.** *Rado, 1948 [37] Let $B(V', E')$ be a biparite graph. Then there exists a $b$-transportation for $B$ if and only if $b(C) \geq \frac{1}{2} b(V')$ for each vertex cover $C$ of $B$.*

In addition, we need the following lemma on the dense decomposition properties.

**Lemma 6.2.** *For a dense decomposition $S_1, ..., S_k$ of $G$ with densities $\lambda_1, ..., \lambda_k$ obtained by Algorithm 1, we have: (i) $\lambda_1 > \lambda_2 > ... > \lambda_k \geq \frac{1}{2}$, (ii) For $S \subseteq S_i$, we must have $E(S, U_i) \geq \lambda_i |S|$.*

*Proof:* (1): Suppose for the sake of a contradiction that this is not the case, and let $i$ be the first index where $\lambda_i \geq \lambda_{i-1}$. Then consider the set $S_{i-1} \cup S_i$ when the algorithm selected $S_{i-1}$. We have that

$$\lambda' = \frac{E(S_{i-1} \cup S_i) + E(S_{i-1} \cup S_i, \bigcup_{1 \leq t < i-1} S_t)}{|S_{i-1} \cup S_i|}$$

Note that $S_j$ are disjoint by construction. So we have the simplification

$$\lambda' = \frac{E(S_{i-1}) + E(S_i) + E(S_{i-1}, \bigcup_{1 \leq t < i-1} S_t) + E(S_i, \bigcup_{1 \leq t < i} S_t)}{|S_{i-1}| + |S_i|} = \frac{\lambda_{i-1} |S_{i-1}| + \lambda_i |S_i|}{|S_{i-1}| + |S_i|} \geq \lambda_{i-1}$$

If $\lambda' = \lambda_{i-1}$ then this would be a contradiction to the maximality of $S_{i-1}$. If $\lambda' > \lambda_{i-1}$ then that would be a contradiction that $S_{i-1}$ was the densest subgraph when it was chosen. Finally, $\lambda_k \geq 1/2$ is clear, the minimum density for a connected component is that of just a single edge which has density $\frac{1}{2}$.

(2): Suppose for the sake of a contradiction that this is not the case for some $S_i$ and $S \subset S_i$, then consider the set $S_i - S$. Then we have that

$$\frac{E(S_i - S) + E(S_i - S, U_{i-1})}{|S_i - S|} = \frac{E(S_i) + E(S_i, U_{i-1}) - E(S, U_i)}{|S_i| - |S|} > \frac{\lambda_i |S_i| - \lambda_i |S|}{|S_i| - |S|} = \lambda_i$$

A contradiction to optimality of $S_i$. ■

We are now ready to prove Theorem 4.1.

*Proof of* Theorem 4.1 We construct a bipartite graph $B(V', E')$ as follows. $V' = L' \cup R'$, where the left side vertices $L' = V(G)$ and the right side vertices are the edges $R' = E(G)$. We set $b_u = \lambda_u$ for $u \in L'$, and $b_e = 1$ for $e \in R'$. We connect a vertex $i \in L'$ to $e \in E'$ if $i$ is incident on $e$ in $G$. It is clear that a $b-$transportation for $B$ induces a feasible solution for LP 4.2. We will now show that there is a feasible $b-$transport using Theorem 6.1. First, note that $b(V') = \left(\sum_{i=1}^{k} \lambda_i |S_i|\right) + m \times 1$ But note that $m = \sum_{i=1}^{k} \lambda_i |S_i|$ (as each edge in $G$ gets counted exactly once) which implies that $\frac{1}{2} b(V') = \sum_{i=1}^{k} \lambda_i |S_i|$. Now we will show that any vertex cover $C$ of the bipartite graph must satisfy $b(C) \geq \sum_{i=1}^{k} \lambda_i |S_i|$ which would imply the theorem. Let $C_L = C \cap L'$ and $C_R = C \cap R'$ be the vertices in the vertex cover on the left and right respectively. Further, subdivide $C_L, C_R$ into $C_{L1}, ..., C_{Lk}$ and $C_{R1}, ..., C_{Rk}$ where $C_{Li} = C_L \cap S_i$ and $C_{Ri} = C_R \cap (E(S_i) \cup E(S_i, \bigcup_{t<i} S_t))$. See Figure 6.3.

Consider $S_1 - C_{L1}$, we must have that $E(S_1 - C_{L1}) + E(S_1 - C_{L1}, C_{L1}) \geq \lambda_1 |S_1 - C_{L1}|$ using Lemma 6.2. But note that $E(S_1 - C_{L1}) \cup E(S_1 - C_{L1}, C_{L1})$ are precisely the edges that are not covered by $C_{L1}$ and hence must be covered by $C_{R1}$, so it must be that $E(S_1 - C_{L1}) \cup E(S_1 - C_{L1}, C_{L1}) \subseteq C_{R1}$ which implies $|C_{R1}| \geq |E(S_1 - C_{L1})| + |E(S_1 - C_{L1}, C_{L1})| \geq \lambda_1 |S_1 - C_{L1}|$

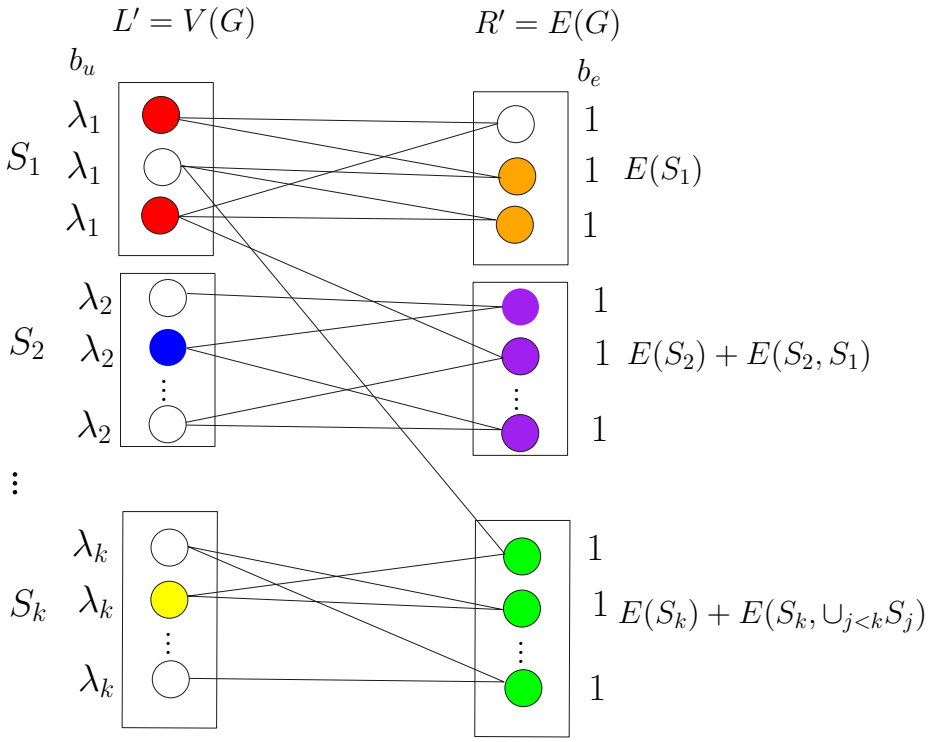


Figure 6.3: $B(V', E')$ bipartite graph in Proof of Theorem 4.1. The colored vertices are the vertex cover $C$. The red vertices are $C_{L_1}$, the blue vertices are $C_{L_2}$, and the yellow vertices are $C_{L_k}$. Similarly, the orange vertices are $C_{R_1}$, the purple vertices are $C_{R_2}$, and the green vertices are $C_{R_k}$.

So we have that $b(C_{L1} \cup C_{R1}) \geq \lambda_1 |C_{L1}| + \lambda_1 |S_1 - C_{L1}| = \lambda_1 |S_1|$. This analysis holds inductively to show that $b(C_{Li} \cup C_{Ri}) \geq \lambda_i |S_i|$. Summing up, we obtain the following,

$$b(C) = \sum_{i=1}^{k} b(C_{Li} \cup C_{Ri}) \geq \sum_{i=1}^{k} \lambda_i |S_i| = \frac{1}{2} b(V').$$

This finishes the proof. ■

Fujishige proved Theorem 4.3 [34] and more general versions of his theorem are also known (see [50]). Here we give a proof for the sake of completeness following the algorithmic definition of the decomposition.

*Proof of* Theorem 4.3: (1): Clearly $b_u \geq 0$ for all $u \in V$. Consider an arbitrary set $R \subseteq V$ and let $R_i = S_i \cap R$. Since $R_i \subseteq S_i$ and $S_i$ is the densest set chosen during iteration $i$, then it must be that

$$\frac{f(R_i \cup S_1 \cup ... \cup S_{i-1}) - f(S_1 \cup ... \cup S_{i-1})}{|R_i|} \leq \frac{f(S_i \cup S_1 \cup ... \cup S_{i-1}) - f(S_1 \cup ... \cup S_{i-1})}{|S_i|} = \lambda_i$$

And hence we have that

$$b(R) = \sum_{u \in R} b_u = \sum_{i=1}^{k} \lambda_i |R_i| \geq \sum_{i=1}^{k} (f(R_i \cup S_1 \cup ... \cup S_{i-1}) - f(S_1 \cup ... \cup S_{i-1}))$$

$$\geq \sum_{i=1}^{k} (f(R_i \cup R_1 \cup ... \cup R_{i-1}) - f(R_1 \cup ... \cup R_{i-1})) = f(R)$$

Where the second inequality is by supermodularity of $f$ and the last equality is because of the telescoping sum. Finally, note the chain of inequalities above hold with equality if $R = V$ since $R_i = S_i$. This implies $b(V) = f(V)$.

| Problem | Algorithm | Convergence (Current worst case number of iters.) | Time/iter | Iteration parallelizable? |
|---|---|---|---|---|
| DSG-LD | Frank-Wolfe based Algorithm | $O(\frac{m\Delta(G)}{\epsilon^2})$ iterations for $\epsilon$-load vector | $O(m)$ | Yes |
| DSG-LD | Greedy++ | Load vector not proven to converge, but experimentally does. | $O(m \log n)$ | No |
| DSG-LD | MWU based Algorithm | $O(\frac{m\Delta(G)}{\epsilon^2})$ iterations for $\epsilon$-load vector | $O(m)$ | Yes |
| DSG-LD | **FISTA based Algorithm** | $O(\frac{\sqrt{m\Delta(G)}}{\epsilon})$ iterations for $\epsilon$-load vector | $O(m)$ | Yes |
| DSG | Bahmani *et al.* [30] primal-dual | $O(\frac{\log m}{\epsilon^2})$ for $(1-\epsilon)$ *multiplicative* DSG. | $O(m)$ | Yes |
| DSG | Boob *et al.* [31] via mixed packing-covering LP solver | $\tilde{O}(\frac{m\Delta(G)}{\epsilon})$ for $(1-\epsilon)$ *multiplicative* DSG. | NA | NA |
| DSG | Greedy++ | $O(\frac{\Delta(G)}{\lambda^*\epsilon^2})$ for $(1-\epsilon)$ *multiplicative* DSG | $O(m \log n)$ | No |
| DSG | Chekuri *et al.* [2] via approximate flow | $O(\frac{\log m}{\epsilon})$ for $(1-\epsilon)$ *multiplicative* DSG. | $\tilde{O}(m)$ | No |
| DSG | Frank-Wolfe based Algorithm | $O(\frac{mn\Delta(G)}{\epsilon^2})$ for $\epsilon$ *additive* DSG using fractional peeling from this paper. | $O(m)$ | Yes |
| DSG | **FISTA based Algorithm** | $O(\frac{\sqrt{mn\Delta(G)}}{\epsilon})$ for $\epsilon$ *additive* DSG using fractional peeling from this paper. | $O(m)$ | Yes |

Figure 6.4: Summary of currently known bounds on different iterative algorithms for DSG and DSG-LD including results in this paper.

(2) : Let $b_u^* = \lambda_u$ with $b^* \in B_f$ from (1). Let $b \in B_f$ be a lexicographically minimal base. We will prove that $b = b^*$ by inductively proving that for all $i$, $b_u = \lambda_u$ if $u \in S_i$. Consider $i = 1$ for the base case. Since $b \in B_f$ we have

$$b(S_1) \geq f(S_1) = \lambda_1 |S_1|$$

Hence the maximum load in $b$ is at least $\lambda_1$. Since the maximum load in $b^*$ is $\lambda_1$, then it forces $b_u = \lambda_u$ for $u \in S_1$. Now we proceed inductively, assuming that $b_u = \lambda_u$ for $u \in S_1 \cup ... \cup S_i$. Since $b \in B_f$,

$$\sum_{h=1}^{i+1} b(S_h) = b(S_1 \cup ... \cup S_{i+1}) \geq f(S_1 \cup ... \cup S_{i+1}) = \sum_{h=1}^{i+1} (f(S_1, ..., S_h) - f(S_1, ..., S_{h-1}))$$

$$= f(S_1 \cup ... \cup S_{i+1}) - f(S_1 \cup ... \cup S_i)) + \sum_{h=1}^{i} b(S_h) = \lambda_{i+1} |S_{i+1}| + \sum_{h=1}^{i} b(S_h)$$

This implies that $b(S_{i+1}) \geq \lambda_{i+1} |S_{i+1}|$. We have $b_u^* = b_u$ for $u \in S_1 \cup ... \cup S_i$ by induction hypothesis. Since $b_u^* = \lambda_{i+1}$ for all $u \in S_{i+1}$ and $b(S_{i+1}) \geq \lambda_{i+1}|S_{i+1}|$ it follows that $b_u = \lambda_{i+1}$ for $u \in S_{i+1}$ for otherwise $b$ is not lexicographically minimal.

Hence, by induction, $b_u = \lambda_u$ for all $u \in V$.

(3): The function $g(b) = \sum_{u \in V} b_u^2$ is strictly convex. $B_f$ is a bounded polyhedron and hence a closed convex set. In addition, we showed that $B_f$ is feasible. Any strictly convex function with a feasible convex constraint set must have a unique solution, and so $b^*$ must be unique.

Now consider the constraint polytope $\mathcal{C}$ defined as the intersection of the following $k$ inequalities

$$\mathcal{C} = \{b \in \mathbb{R}^{|V|} : \forall 1 \leq r \leq k, \sum_{l=1}^{r} b(S_l) \geq \sum_{l=1}^{r} \lambda_l |S_l|\}$$

Recall if $b \in B_f$ then

$$\sum_{i=1}^{r} b(S_i) = b(S_1 \cup ... \cup S_r) \geq f(S_1 \cup ... \cup S_r) = \sum_{i=1}^{r} f(S_1 \cup ... \cup S_i) - f(S_1 \cup ... \cup S_{i-1}) = \sum_{i=1}^{r} \lambda_i |S_i|$$

And so $b \in B_f \implies b \in \mathcal{C}$. Hence $\min_{b \in \mathcal{C}} g(b) \leq \min_{b \in B_f} g(b)$.

We aim to prove that for the unique optimal solution of $g$ under $\mathcal{C}$, all inequalities are active (i.e hold with equality). Let $b$ be the optimal solution of $g$ under $\mathcal{C}$. From (1), we have that $b(V) = f(V)$ and so the last inequality has to be active. Now suppose for the sake of contradiction that not all inequalities are active, and let $i$ be the first index of an inequality that is not active. Since $\sum_{h=1}^{i-1} b(S_h) = \sum_{h=1}^{i-1} \lambda_h |S_h|$ and $\sum_{h=1}^{i} b(S_h) > \sum_{h=1}^{i} \lambda_h |S_h|$, then it must be that $b(S_i) > \lambda_i |S_i|$. Similarly, let $j > i$ be the first index of an inequality after $i$ which is active (this must exist since the last inequality holds with equality). Then it must be that $b(S_j) < \lambda_j |S_j|$. Now let

$$\epsilon = \min \left( \min_{i \leq t < j} \left( \sum_{r=1}^{t} b(S_r) - \sum_{r=1}^{t} \lambda_r s_r \right), \frac{1}{2}(\lambda_i - \lambda_j) \right) > 0$$

be the minimum "excess" from inequalities $i$ to $j-1$ and half the difference between $\lambda_i, \lambda_j$. Since $b(S_i) > \lambda_i |S_i|$, then there exists $b_u > \lambda_i$ for $u \in S_i$. Similarly, there exists $b_v < \lambda_j$ for $v \in S_j$. Now consider the solution of $b'$ where $b'_u = b_u - \epsilon, b'_v = b_v + \epsilon$, and $b'_x = b_x$ otherwise. The solution is feasible in $\mathcal{C}$ because inequalities $1, ..., i-1$ stay the same (no $b_u, b_v$ variable), inequalities $i, ..., j-1$ stay feasible (LHS decreases by $\epsilon$), and the effect by inequality $j$ is cancelled. However, we have that since $b_u > \lambda_i > \lambda_j > b_v$ and $\epsilon \leq \frac{1}{2}(\lambda_i - \lambda_j)$ that $(b_u - \epsilon)^2 + (b_v + \epsilon)^2 < b_u^2 + b_v^2$ So the solution $b'$ has a strictly smaller cost, contradicting optimality of $b$.

This implies that all inequalities have to hold with equality in an optimal solution. So $\sum_{u \in S_i} b_u = \lambda_i |S_i|$. The sum of squares is minimized if and only if all variables have equal weight, and so it must be that $b_u = \lambda_i = \lambda_u$ for all $u \in S_i$. This shows $g(b)$ is minimized at $b_u = \lambda_u$ under $\mathcal{C}$. But recall from (1) that $b \in B_f$, and so $b$ is the unique optimal solution for Problem 4.7 ■

We now show the equivalence of the two LPs 4.2 and 4.6 for DSG.

*Proof of* Theorem 4.4: Suppose $b \in B_f$. Then $b(V) = m$ and $b(S) \geq |E(S)|$ for all $S \subseteq V$. We need to prove the existence of $x \geq 0$ such that $x_{uv} + x_{vu} = 1$ for all edge $\{u, v\} \in E$ and such that the total load on each vertex is at most $b$. The idea from the proof of Theorem 4.1 goes through to show this. The only fact we used in the proof of Theorem 4.1 is that $b(S) \geq |E(S)|$ for certain sets $S \subseteq V$ and $b(V) = m$ which hold as we mentioned.

Suppose $x, b$ satisfy the constraints of 4.2. Since $x_{uv} + x_{vu} = 1$ for each edge $\{u, v\}$, for any $S \subseteq V$,

$$b(S) = \sum_{u \in S} b_u = \sum_{u \in S} \sum_{v \in \delta(u) \cap S} x_{uv} + \sum_{u \in S} \sum_{v \in \delta(u) \setminus S} x_{uv} \geq |E(S)|$$

Similarly, $b(V) = \sum_{u \in V} \sum_{v \in \delta(u)} x_{uv} = m = |E(V)|$. Thus $b \in B_f$. ■

### 6.2. Details of FISTA based Algorithm

*Proof of* Lemma 5.1 $\nabla f_{uv} = 2 \sum_{w \in \delta(u)} x_{uw}$. So for $x, y \in \mathbb{R}^{2m}$,

$$\|\nabla f(x) - \nabla f(y)\|^2 = 4 \sum_{uv \in ord(E)} \left( \sum_{w \in \delta(u)} x_{uw} - y_{uw} \right)^2 = 4 \sum_{u \in V} \sum_{v \in \delta(u)} \left( \sum_{w \in \delta(u)} x_{uw} - y_{uw} \right)^2$$

$$\leq 4\Delta(G) \sum_{u \in V} \left( \sum_{w \in \delta(u)} x_{uw} - y_{uw} \right)^2 \leq 4\Delta(G)^2 \sum_{u \in V} \sum_{w \in \delta(u)} (x_{uw} - y_{uw})^2 = 4\Delta(G)^2 \|x - y\|^2$$

Where the last inequality holds by Cauchy-Schwarz inequality. ■

*Proof of* Lemma 5.2 Fix $u < v$ and let $p = \text{prox}_h(x)$. Then we aim to minimize

$$\sum_{uv \in ord(E)} (p_{uv} - x_{uv})^2 = \sum_{uv \in E} \left( (p_{uv} - x_{uv})^2 + (1 - p_{uv} - x_{vu})^2 \right)$$

So it is sufficient to minimize $(p_{uv} - x_{uv})^2 + (1 - p_{uv} - x_{vu})^2$ individually for each unordered edge $u < v$ subject to $0 \leq p_{uv} \leq 1$. Proving that the described proximal mapping minimizes this quadratic is a standard exercise so we omit it.

We describe below the full details of the FISTA based algorithm for DSG.

**Algorithm 3** FISTA Algorithm for Densest Subgraph

Input is Graph $G$ and number of iterations $T$. Assume $E = E(G)$ is ordered (so includes $(u, v)$ and $(v, u)$ for every edge).
$\Delta = \max_{u \in G} |\delta(u)|$ ▷ Maximum Degree
$\alpha \leftarrow \frac{1}{2\Delta}$ ▷ Learning Rate
$x^{(0)}(u, v) = 1 \quad \forall (u, v) \in E, u < v$
$x^{(0)}(v, u) = 0 \quad \forall (u, v) \in E, u < v$
$y^{(0)} = x^{(0)}$
**for** $t \in [1, T]$ **do**
  $b^{(t)}(u) = 0 \quad \forall u \in V$ ▷ Calculate Load with respect to $y^{(t-1)}$
  **for** $u \in G$ **do**
    **for** $v \in \delta(u)$ **do**
      $b^{(t)}(u) = b^{(t)}(u) + y^{(t-1)}(u, v)$

  $g^{(t)}(u, v) = 0 \quad \forall (u, v) \in E$ ▷ Calculate Gradient with respect to $y^{(t-1)}$
  **for** $(u, v) \in E, u < v$ **do**
    $g^{(t)}(u, v) = g^{(t)}(u, v) + 2b^{(t)}(u)$
    $g^{(t)}(v, u) = g^{(t)}(v, u) + 2b^{(t)}(v)$

  $z^{(t)}(u, v) = y^{(t)}(u, v) - \alpha g^{(t)}(u, v) \quad \forall (u, v) \in E$ ▷ Descent direction
  $x^{(t)}(u, v) = 0 \quad \forall (u, v) \in E$ ▷ Calculate New $x^{(t)}$, which is projected descent direction
  **for** $(u, v) \in E, u < v$ **do**
    $diff \leftarrow z^{(t-1)}(u, v) - z^{(t-1)}(v, u)$
    **if** $diff \geq -1$ and $diff \leq 1$ **then**
      $x^{(t)}(u, v) \leftarrow \frac{diff+1}{2}$
    **else if** $diff > 1$ **then**
      $x^{(t)}(u, v) \leftarrow 1$
    **else**
      $x^{(t)}(u, v) \leftarrow 0$
    $x^{(t)}(v, u) \leftarrow 1 - x^{(t)}(u, v)$

  $y^{(t)}(u, v) \leftarrow x^{(t)}(u, v) + \frac{t-1}{t+2}(x^{(t)}(u, v) - x^{(t-1)}(u, v)) \quad \forall (u, v) \in E$ ▷ Calculate New $y^{(t)}$

**return** $x^{(T)}$

### 6.3. Approximate densest decomposition via fractional peeling

*Proof of* Theorem 5.5

$$f(b) - f(b^*) = \sum_{u \in V} b_u^2 - \sum_{u \in V} \lambda_u^2 = \sum_{i=1}^{k} \sum_{u \in S_i} (b_u^2 - \lambda_u^2) = \sum_{i=1}^{k} \sum_{u \in S_i} (b_u^2 - \lambda_i^2) \leq \mu$$

Now let $b_u = \lambda_u + \delta_u$, then we have from above that

$$\sum_{i=1}^{k} \sum_{u \in S_i} (2\lambda_i \delta_u + \delta_u^2) \leq \mu$$

We will first show that $\sum_{i=1}^{k} \sum_{u \in S_i} 2\lambda_i \delta_u \geq 0$. Note that for any $l$, we have

$$\sum_{i=1}^{l} \sum_{u \in S_i} (\lambda_i + \delta_u) = \sum_{i=1}^{l} \sum_{u \in S_i} b_u \geq \sum_{i=1}^{l} \lambda_i |S_i|$$

Since the edges with both endpoints in $S_1 \cup ... \cup S_l$ get double counted, and $x_{uv} + x_{vu} = 1$. Which implies that for all $l$,

$$\sum_{i=1}^{l} \sum_{u \in S_i} \delta_u \geq 0$$

Now we prove that for all $l$, and any $a_1 > a_2 > ... > a_l \geq 0$

$$\sum_{i=1}^{l} \sum_{u \in S_i} \delta_u a_i \geq 0$$

by induction. Observe that it holds for $l = 1$ since $a_1 \sum_{u \in S_1} \delta_u \geq 0$. For $l = r$, we have that

$$\sum_{i=1}^{r} \sum_{u \in S_i} \delta_u a_i = \sum_{i=1}^{r-1} \sum_{u \in S_i} \delta_u (a_i - a_r) + a_r \sum_{i=1}^{r} \sum_{u \in S_i} \delta_u \geq 0 + 0 = 0$$

By induction. Since $\lambda_1 > ... > \lambda_k$ by Lemma 6.2, this implies

$$\sum_{i=1}^{k} \sum_{u \in S_i} \lambda_i \delta_u \geq 0$$

Which implies

$$\sum_{i=1}^{k} \sum_{u \in S_i} \delta_u^2 \leq \mu$$

Now let $\mu = \epsilon^2$, which would imply

$$\sum_{i=1}^{k} \sum_{u \in S_i} (b_u - \lambda_u)^2 \leq \epsilon^2$$

And hence $\|b - b^*\| \leq \epsilon$ ■

### 6.4. Proof of Theorem 5.6

We first note the running time follows easily from using a heap in the fractional peeling subroutine to identify the next minimum-load vertex. We focus on proving the approximation factor.

In what follows, refer to Figure 6.5. Let $x^{(0)} = x$ and $b^{(0)} = b$. Each iteration $t = 1, 2, \ldots$, the algorithm runs fractional peeling over the remaining vertices, with respect to the fractional orientation given by $x^{(t-1)}$ and the loads given by $b^{(t)}$. This produces a set of vertices $T_t$. We remove $T_t$ from the vertex set. $x^{(t)}$ is obtained from $x^{(t-1)}$ by assigning all the edges cut by $T_t$ to the endpoint not in $T_t$. That is, for each edge $\{u, v\}$ cut by $T_t$ in the remaining graph, where $u \in T_t$ and $v \notin T_t$ we set $x_{uv}^{(t)} = 0$ and $x_{vu}^{(t)} = 1$. We let $b^{(t)}$ denote the loads induced by $x^{(t)}$. Note that $b_u^{(t)}$ is nondecreasing for $u \notin T_t$, and nonincreasing for $u \in T_t$. In particular we have $b_u^{(t)} \geq b_u^{(0)} \geq \lambda_u - \varepsilon$ for every remaining vertex $u$.

We want to show that for each iteration $t$, and each vertex $u \in T_t$, the density of $T_t$ is at least $\lambda_u - \varepsilon(1 + \sqrt{n})$.

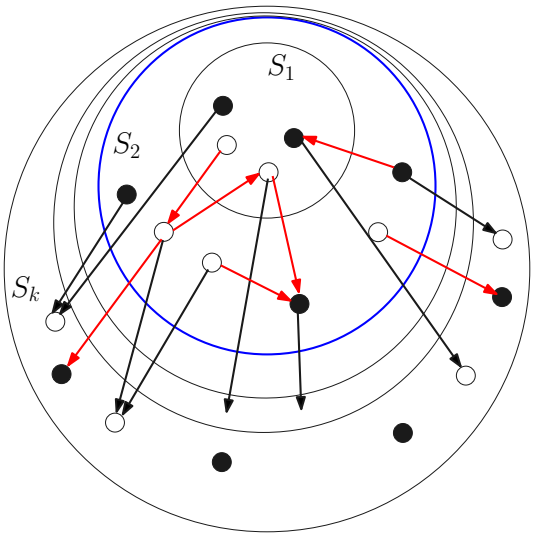


Figure 6.5: Let $S_1, ..., S_k$ be as shown in the figure. Suppose we are in iteration $t$. The dark vertices represent the vertices that were deleted in previous rounds (i.e $T_1 \cup ... \cup T_{t-1}$), while the white vertices are those that were not deleted yet. The sum of $\delta_2^{(t)}$ only includes the black edges leaving $S_1, S_2$ to "outside" white points. The red edges are all examples of edges not included in the sum.

First, for each iteration $t$, and each index $i$ from the dense decomposition, let $\delta_i^{(t)}$ denote the sum,

$$\delta_i^{(t)} = \sum_{\substack{u \in S_1 \cup \cdots \cup S_i \\ v \in S_{i+1} \cup \cdots \cup S_k \\ v \notin T_1 \cup \cdots \cup T_t}} x_{uv}^{(t)}.$$

At a high-level, $\delta_i^{(t)}$ represents the sum of loads in $S_1 \cup \cdots \cup S_i$ from edges cut by $S_1 \cup \cdots \cup S_i$, except omitting the edges where the endpoint outside $S_1 \cup \cdots \cup S_i$ was taken in one of the first $t$ iterations.

We claim that for each index $i$ and iteration $t$, we have

$$\delta_i^{(t)} \leq \varepsilon\sqrt{n}.$$

We first observe that $\delta_i^{(t)}$ is non-increasing in $t$. Indeed, fix $t$, and consider a term $x_{uv}^{(t)}$ appearing in the sum. (That is, $u \in S_1 \cup \cdots \cup S_i$, $v \in S_{i+1} \cup \cdots \cup S_k$, and $v \notin T_1 \cup \cdots \cup T_t$.) In the $(t+1)$th iteration, we select a new set $T_{t+1}$ and $x_{uv}^{(t+1)}$ is bigger than $x_{uv}^{(t)}$ only if $v \in T_{t+1}$. But in this case, $x_{uv}^{(t+1)}$ is omitted from the sum for $\delta_i^{(t)}$.

Since $\delta_i^{(t)}$ is non-increasing in $t$, suffices to prove the claim for $t = 0$, when $b^{(0)} = b$ and $x^{(0)} = x$. To this end, observe that

$$\sum_{u \in S_1 \cup \cdots \cup S_i} b_u = \delta_i^{(0)} + |E(S_1 \cup \cdots \cup S_i)| = \delta_i^{(0)} + \sum_{u \in S_1 \cup \cdots \cup S_i} \lambda_u.$$

Rearranging and applying the Cauchy-Schwarz inequality, we have

$$\delta_i^{(0)} = \sum_{u \in S_1 \cup \cdots \cup S_i} (b_u - \lambda_u) \leq \sqrt{n} \sqrt{\sum_{u \in S_1 \cup \cdots \cup S_i} (b_u - \lambda_u)^2} \leq \varepsilon\sqrt{n}.$$

This establishes the inequality for $t = 0$, hence all $t$ by monotonicity.

Let $u \in T_t$ and suppose $u \in S_i$. We want to show that $T_t$ has density at least $\lambda_u - \varepsilon(1 + \sqrt{n})$. Let $v$ be the first vertex in $S_1 \cup \cdots \cup S_i$ peeled in the $t$th iteration; in particular, $\lambda_v \geq \lambda_u$. Recall that just before $v$ is peeled, $v$ has the lowest load remaining of any vertex, and the sum of loads of

the remaining vertices counts the total number of edges in $T_t$. Thus the load at $v$ is a lower bound on the density of $T_t$. Additionally, before $v$ is peeled, any decrease in $v$'s load is from edges $\{v, w\}$ where $w \in S_{i+1} \cup \cdots \cup S_k$ and $w \notin T_1, \ldots, T_{t-1}$. Thus $v$ has load at least $b_v^{(t-1)} - \delta_i^{(t-1)}$. Putting everything together, we conclude that $T_t$ has density at least

$$b_v^{(t-1)} - \delta_i^{(t-1)} \geq b_v^{(0)} - \varepsilon\sqrt{n} \geq \lambda_v - \varepsilon(1+\sqrt{n}), \geq \lambda_u - \varepsilon(1+\sqrt{n}),$$

as desired. This completes the proof.

### 6.5. Hypergraphs

The projective results for DSG can be generalized for DSS. We say a supermodular function is *projectable* if given a vector $y \in \mathbb{R}^{|V|}$, there is a fast oracle that can calculate $prox_f(y) = \min_{x \in B_f} \|x - y\|_2^2$.

**Theorem 6.3.** *Given $f$, a projectable supermodular function, and an initial load vector $b^{(0)}$ for the guess of $b_u^* = \lambda_u$, there exists an algorithm that returns an approximate load vector $\hat{b}$ that uses $O(\frac{\|b^{(0)} - b^*\|}{\epsilon})$ oracle calls to the projection oracle, and satisfies $\|b - b^*\| \leq \epsilon$ for all $u \in V$.*

*Proof:* Consider applying FISTA [40] on Problem 4.7 where we have unconstrained optimization problem $\sum_{u \in V} b_u^2 + h(b)$ where $h(b)$ is an indicator function for $B_f$. We have that $\nabla f(b) = 2b$, and hence the Lipschtiz constant of $\nabla f$ is 2. Applying Lemma 5.3 with a learning rate of 0.5, we get the desired result. ■

A hypergraph $G = (V, E)$ generalizes the idea of a graph by allowing edges to have size greater than two. For example, if $V = \{a, b, c, d\}$, then a potential "edge" is $e = \{a, b, d\}$. The rank $r$ of a hyper graph is $\max_{e \in E} |e|$. For practical purposes, $r$ is generally "small". Given a hypergraph $G = (V, E)$ we let $E(S)$ denote the set of all hyperedges in $E$ that are fully contained in $S$, that is $E(S) = \{e \in E \mid e \subseteq S\}$. One can easily verify that the function $f : 2^V \to \mathbb{R}_+$ where $f(S) = |E(S)|$ is a monotone nonnegative supermodular function.

We can generalize the approach for DSG to hypergraphs as follows. Charikar's LP relaxation can be generalized to hypergraphs. Here we focus on the dual. For $e \in E$ and $u \in e$, we define a variable $x_{e,u}$ which corresponds to the load that $e$ assigns to $u$. The LP requires each $e$ to be assigned to its end points and hence we have a constraint $\sum_{u \in e} x(e, u) = 1$. The load on $u$, denote by the variable $b_u$, is $\sum_{e : u \in e} x(e, u)$. The goal is to minimize $\max_{u \in V} b_u$. Similar to Theorem 4.4, one can prove in the same way that $b \in B_f$ if and only if $\exists x$ that induces $b$. Now given a vector $y$ that induces $b$, we can project it on $B_f$ by finding $x_{e,u}$ that minimizes $\sum_{u \in e}(x_{e,u} - y_{e,u})^2$ subject to $\sum_{u \in e} x_{e,u} = 1$ and $x_{e,u} \geq 0$. This is known as the simplex projection and it has a simple closed form solution (See [51] for the basic algorithm, and [52] for a recent distributed variant of the algorithm). The algorithm in this case would take $O(|e| \log |e|)$ time for each $e \in E$ to do the projection, and hence overall the projection step takes $O(p \log r)$ for all edges where $p = \sum_{e \in E} |e|$ is the representation size of the hypergraph $G$ ($p$ corresponds to $m$ in graphs). We can apply FISTA analysis in a very similar fashion to that for graphs to obtain the following theorem.

**Theorem 6.4.** *For a hyper graph $G$ with rank $r$, maximum degree $\Delta(G)$ (i.e $\max_{u \in V} |\{e : u \in e\}| = \Delta(G)$), and size $p = \sum_{e \in E} |e|$, there exists an algorithm that takes $O(\frac{\sqrt{r\Delta(G)p}}{\epsilon})$ iterations, each needing $O(p \log r)$ time, to compute an $\epsilon$-approximate load vector $\hat{b}$ satisfying $\left\|\hat{b} - b^*\right\| \leq \epsilon$.*

### 6.6. Approximate load vector via minimum-cost flow

We set up the problem as a quadratic min cost flow problem. Namely, we will set the flow network $\mathcal{V} = \{s\} \cup \{a_v : v \in V\} \cup \{a_e : e \in E\} \cup \{t\}$. We add an edges to $\mathcal{E}$ of the form $(s, a_v)$ of capacity $deg_G(v)$. We add an edge $(a_v, a_e)$ is $v$ if one of the endpoints of $e$ of capacity 1. Finally, we add an edge $(a_e, t)$ of capacity 1 for all edges $e \in E$. See Figure 6.6. One can verify that the maximum flow has cost $m = |E|$ using max flow min cut theorem. Let $\mathcal{F}$ be the set of valid maximum flows

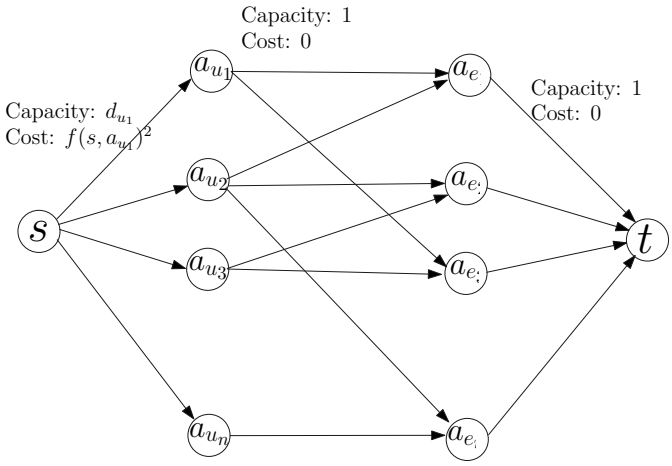


Figure 6.6: Quadratic min cost flow network

for $(\mathcal{V}, \mathcal{E})$. We are interested in the problem of

$$\begin{aligned} &\text{minimize} \quad \sum_{u \in V} f(s, a_u)^2 \\ &\text{subject to} \quad f \in \mathcal{F} \end{aligned} \tag{6.1}$$

One can verify that the optimal flow for Problem 6.1 gives the solution for Problem 4.3; any flow $f \in \mathcal{F}$ corresponds to a solution of the same cost to Problem 4.3 and vice versa. Hence the optimal flow $f^*$ is the dense decomposition vector.

Observe that this flow network has size $O(m)$. We will use Theorem 10.14 from [43]. Define $h_{(s,a_{v_i})}(x) = x^2$ and $h_e(x) = 0$ otherwise for $e \neq (s, a_{v_i})$. These are functions which are sums of $\tilde{O}(1)$ $p-$norms. Hence, in $m^{1+o(1)}$ time, we can compute a min cost flow $f$ of value $|E|$. Setting $C = 3$ in the Theorem, we get that

$$\sum_{u \in V} f(s, a_u)^2 \leq \sum_{u \in V} f^*(s, a_u)^2 + O(\exp(-\log^3 m)) = OPT + \frac{1}{m^{\log^2 m}} \leq OPT + \frac{1}{m^4} \leq OPT + \epsilon^2$$

Note that the smallest $\epsilon$ we care about is $\epsilon \geq n^{-2}$ (since any smaller epsilon wouldn't change the value from setting $\epsilon = n^{-2}$). The chain of inequalities hold for reasonably large $m$ (say $m \geq 5$). Hence $f$ is an $\epsilon$-approximate load vector.

## 6.7. Further Details of Experimental Section

### 6.7.1. More details on MWU and FRANK-WOLFE based algorithms

We described the theoretical aspects of the FRANK-WOLFE and MWU algorithms in Sections 5.5. Here we discuss some concrete details of our implementation.

We use the FRANK-WOLFE implementation in the context of DSG as described by Danisch *et al.* [3]. The algorithm maintains an edge assignment vector that it updates in each iteration $t$. The edge assignment vector $x^{(t-1)}$ at the start of iteration $t$ naturally induces a vertex load vector $b^{t-1}$. We loop over each edge, and set $y^{(t)}_{uv} = 1, y^{(t)}_{vu} = 0$ if $b^{(t-1)}_u < b^{(t-1)}_v$ and $y^{(t)}_{uv} = 0, y^{(t)}_{vu} = 1$ otherwise. Finally, we let $x^{(t)} = x^{(t-1)} + \frac{2}{t+2} y^{(t)}$ which itself induces a new load vector $b^{(t)}$. Each iteration takes $O(m)$ time. This is the same implementation as described in the Danisch *et al.* [3] paper except that we initialize the starting vector $x^{(0)}$ differently. Danisch *et al.* initialize $x^{(0)}_{uv} = x^{(0)}_{vu} = 0.5$ while we initialize it with the edge assignment obtained from running the Greedy algorithm.

There is an alternative way to implement the algorithm without using edge assignment variables. We maintain a vertex load vector $b^{(t)}$ of size $n$. Each iteration is implemented as follows. We sort the vertices $u_1 < ... < u_n$ in ascending order of $b^{(t-1)}_u$ (ties broken arbitrarily). Then, for each $u_i$,

we set

$$b_u^{(t)} = b_u^{(t-1)} + \frac{2}{t+2} |u_j \in \delta(u_i) : j > i|$$

One can verify that the two algorithms are equivalent with a slightly different implementation. This implementation takes $O(n \log n)$ time for the sort, and an $O(m)$ loop over the graph, but does not have to store an additional array of size $2m$ for each edge. When $n \ll m$, this can make a substantial difference.

The MWU algorithm can also be implemented in two different ways (one edge based, and one vertex based). We choose to implement the algorithm in the vertex based approach in comparison to the FRANK-WOLFE based approach. Specifically, the algorithm maintains a vertex load vector that it updates in each iteration as follows. In iteration $t$, it sorts the vertices as $u_1 < ... < u_n$ in ascending order of $b_u^{(t-1)}$ values. Then, for each $u_i$, it sets

$$b_u^{(t)} = b_u^{(t-1)} + \frac{1}{t+1} |u_j \in \delta(u_i) : j > i|$$

We keep track of the edge assignment vector $x(t)$ (that induces $b^{(t)}$) for the fractional peeling experiment — however, when reporting the running time we do not add this overhead since the algorithm does not require maintaining the $x^{(t)}$ vector.

### 6.7.2. Additional data

See the end of the Appendix for enlarged plots of the main paper plots and additional plots for all datasets.

1. Figure 6.7 shows the density achieved as number of iterations vary for all algorithms on all datasets when static load sorting is used instead of fractional peeling. Figure 6.8 shows the effect of adding fractional peeling to all the algorithms.
2. Figure 6.9 shows the time per iteration histogram for all datasets and all algorithms that we tested.
3. Figures 6.10 and 6.11 show the error plots (i.e sum of $\sum_{u \in V} b_u^2$) of all algorithms on all datatasets. Specifically, Figure 6.11 zooms in on the last few iterations to see what is happening near the end.
4. Figure 6.12 shows the sorted load vector after 100 iterations of FISTA in sorted order, where a vertex rank is its relative order in terms of its load vector in $V$, and load is the value $b_u$. It appears that for each $S_i$, FISTA focuses on adjusting $b_u$ for most vertices in $S_i$, but a few vertices "lag" behind in lower/higher levels, and slowly bubble down as shown the dataset for CLOSE CLIQUES and ROADNET PA. This gives some intuition on why fractional peeling does well in practice as these "trailing" vertices will be peeled first by fractional peeling allowing the dense component to stabilize.
5. Figure 6.13 shows the wall clock time of different algorithms on all 8 datasets. Figure 6.14 shows the same figure but zoomed in on the first 20 percent of the time.

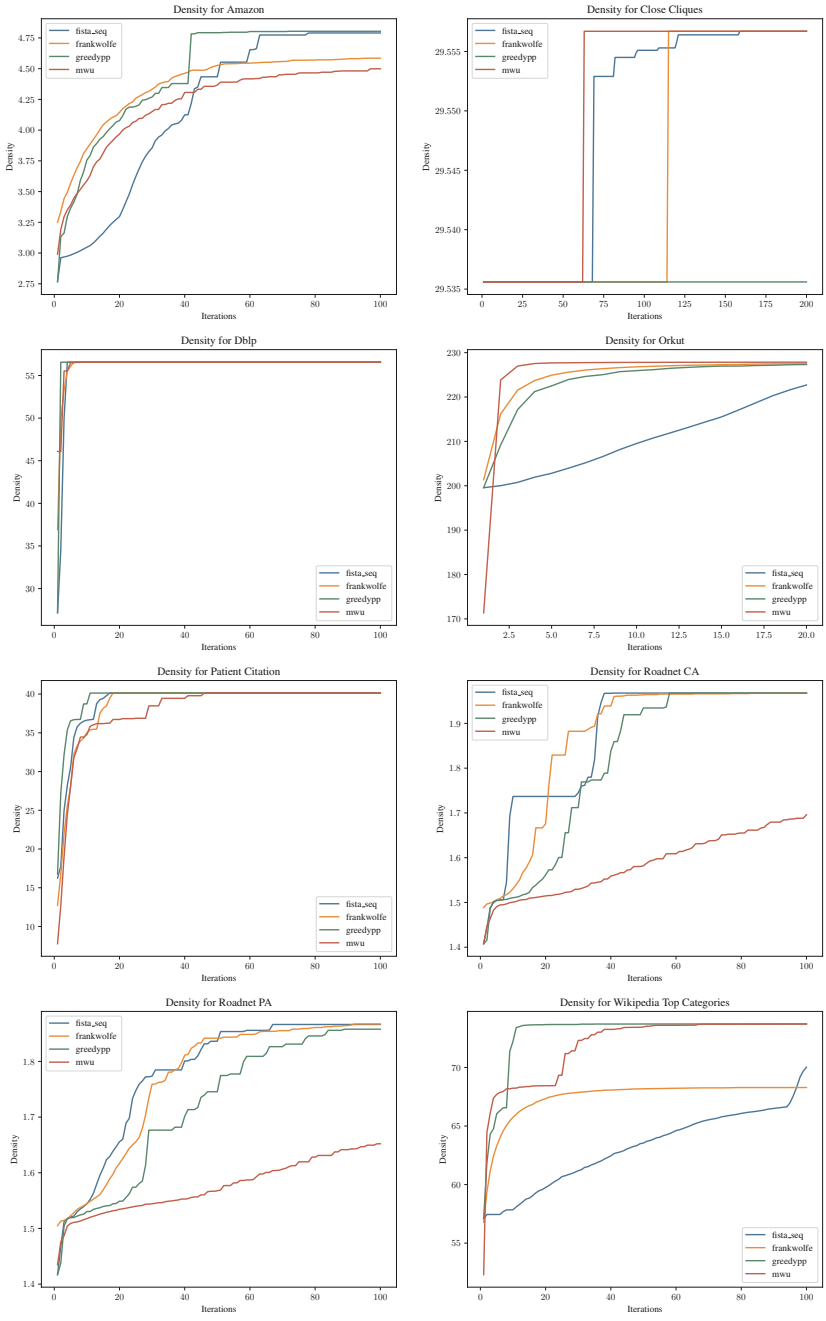


Figure 6.7: Density based on sorting loads

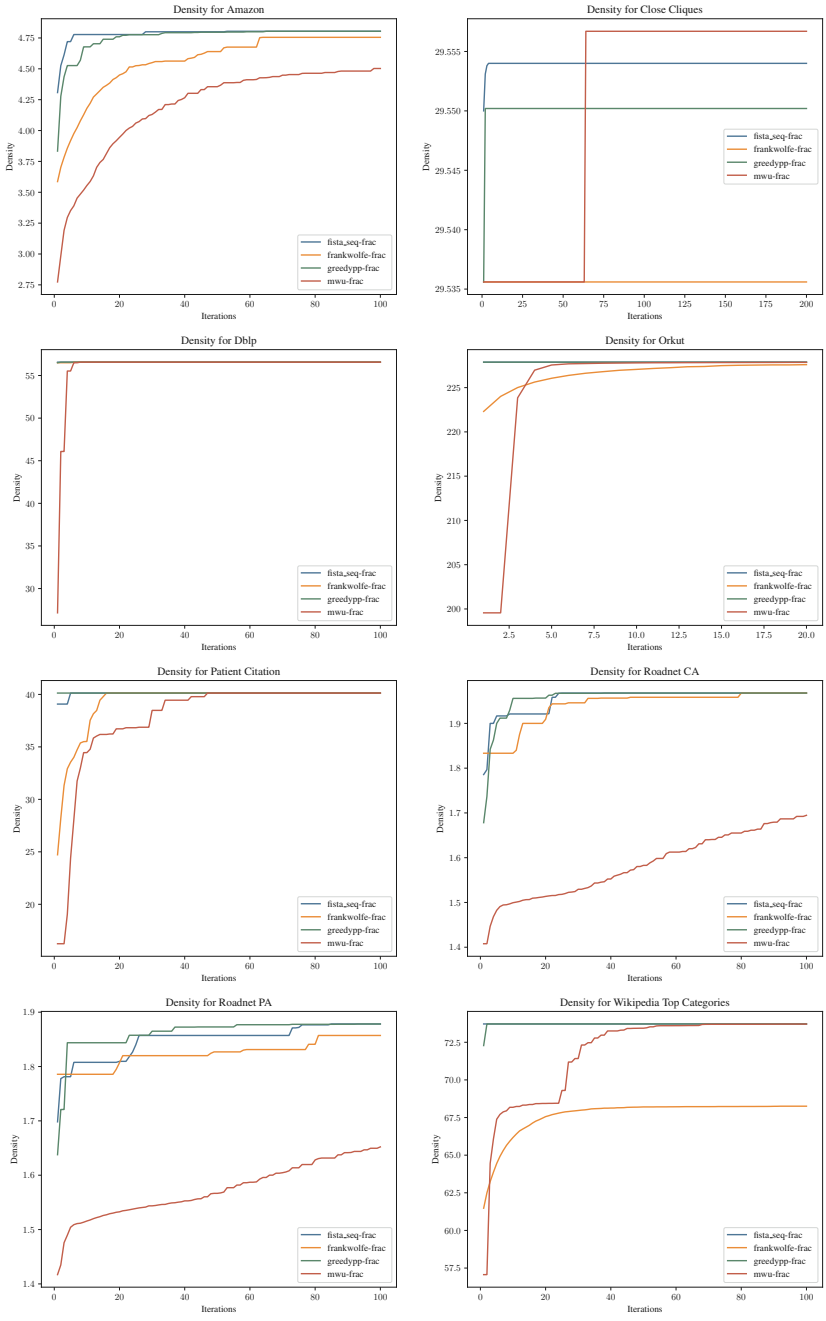


Figure 6.8: Density based on Fractional Peeling

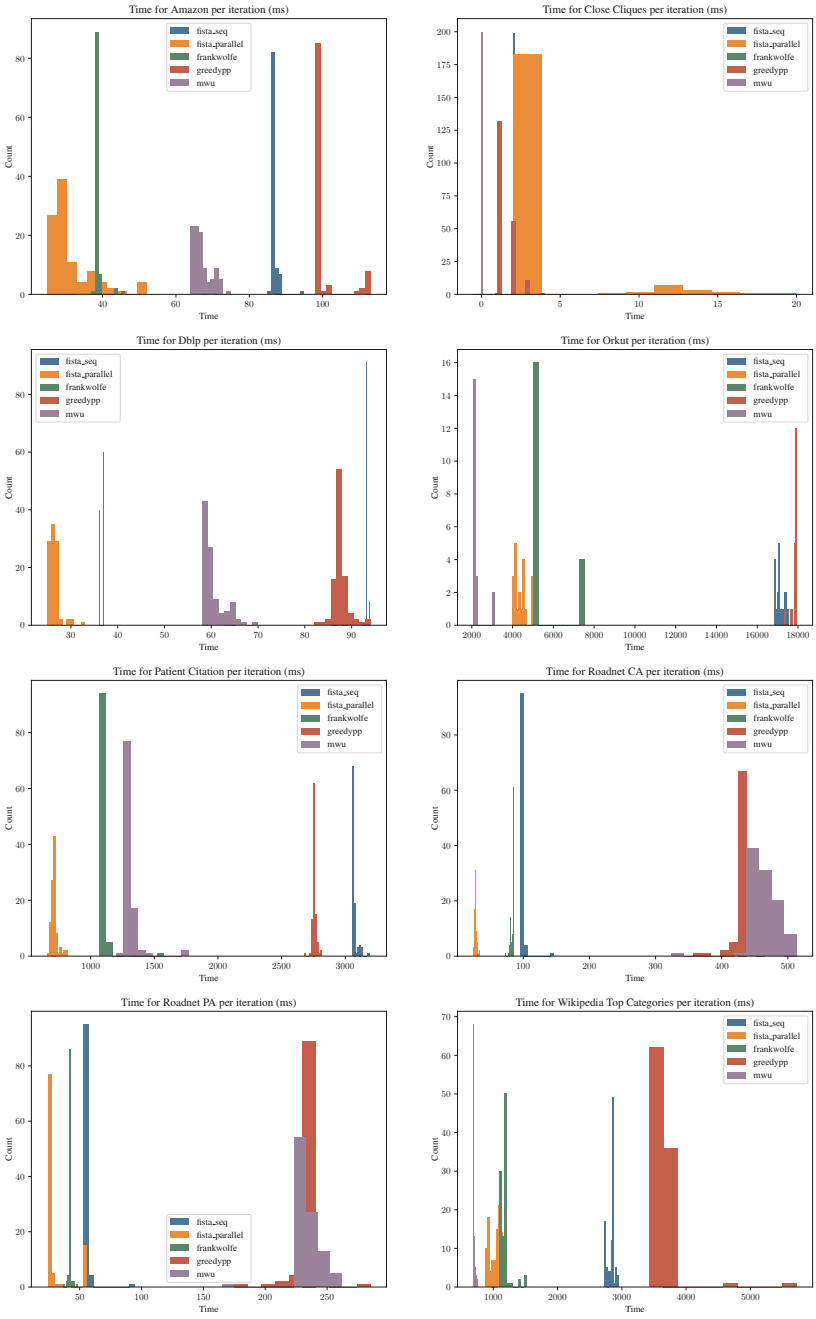


Figure 6.9: Time take per iteration histogram

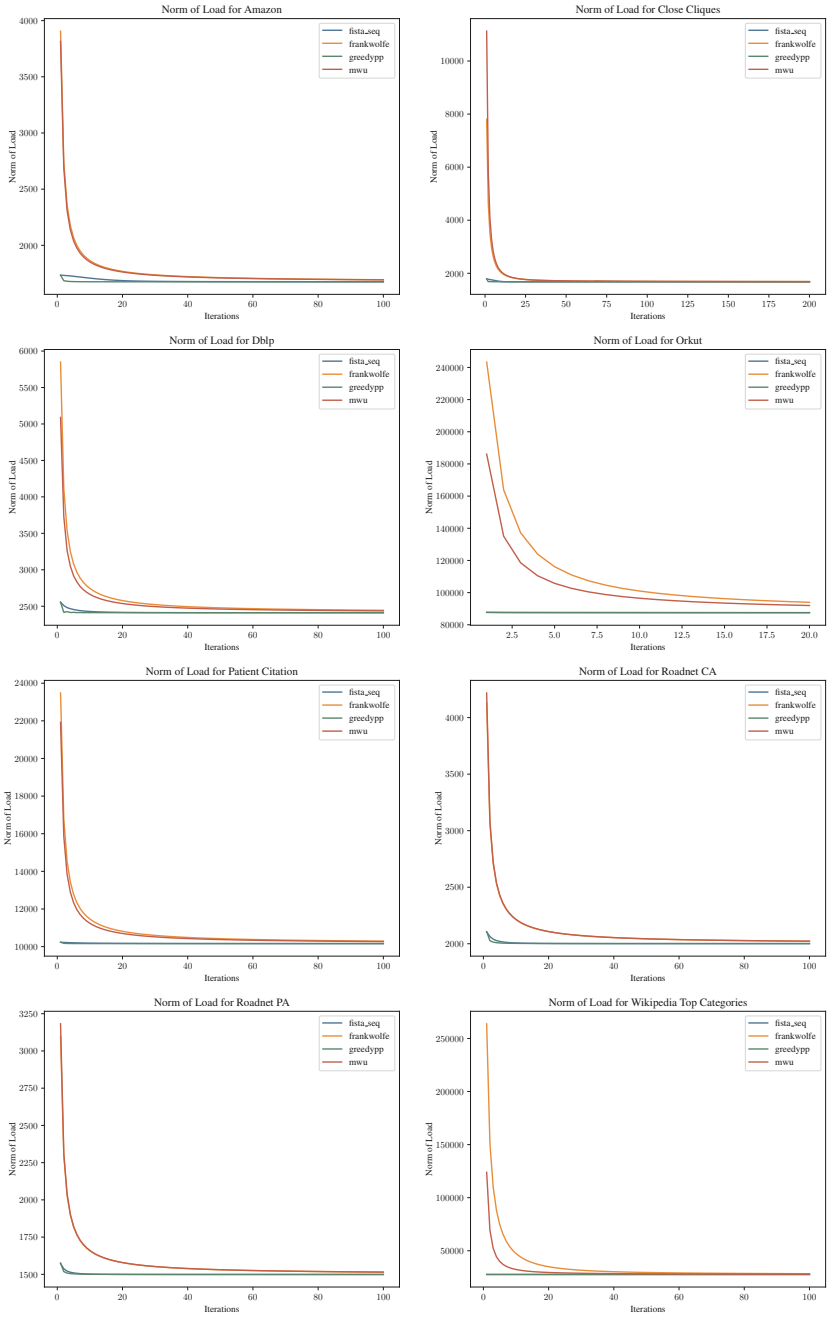


Figure 6.10: $L_2$ norm of load vector. See Figure 6.11 for a zoom-in on the last 20 iterations.

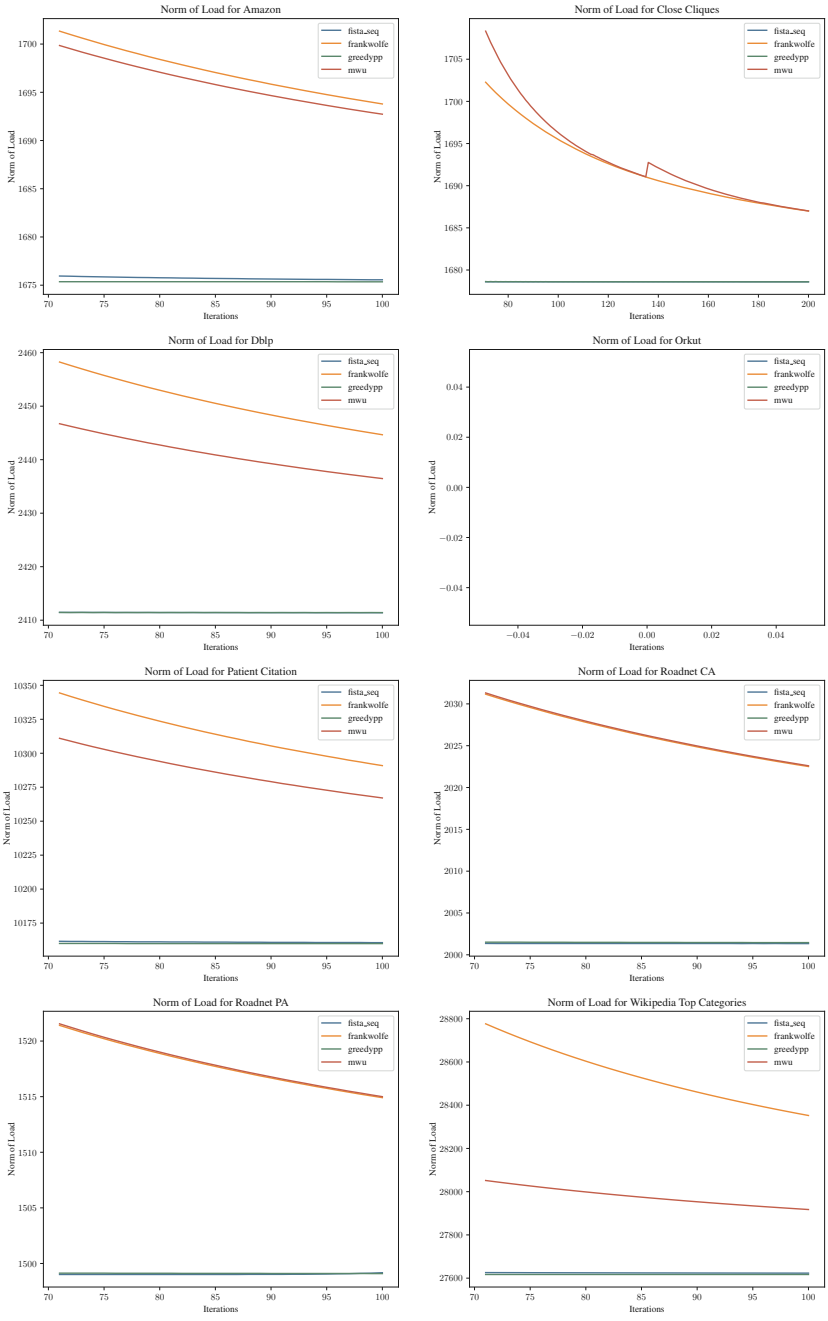


Figure 6.11: Same as Figure 6.10 but zoomed in from Iteration 70 and after.

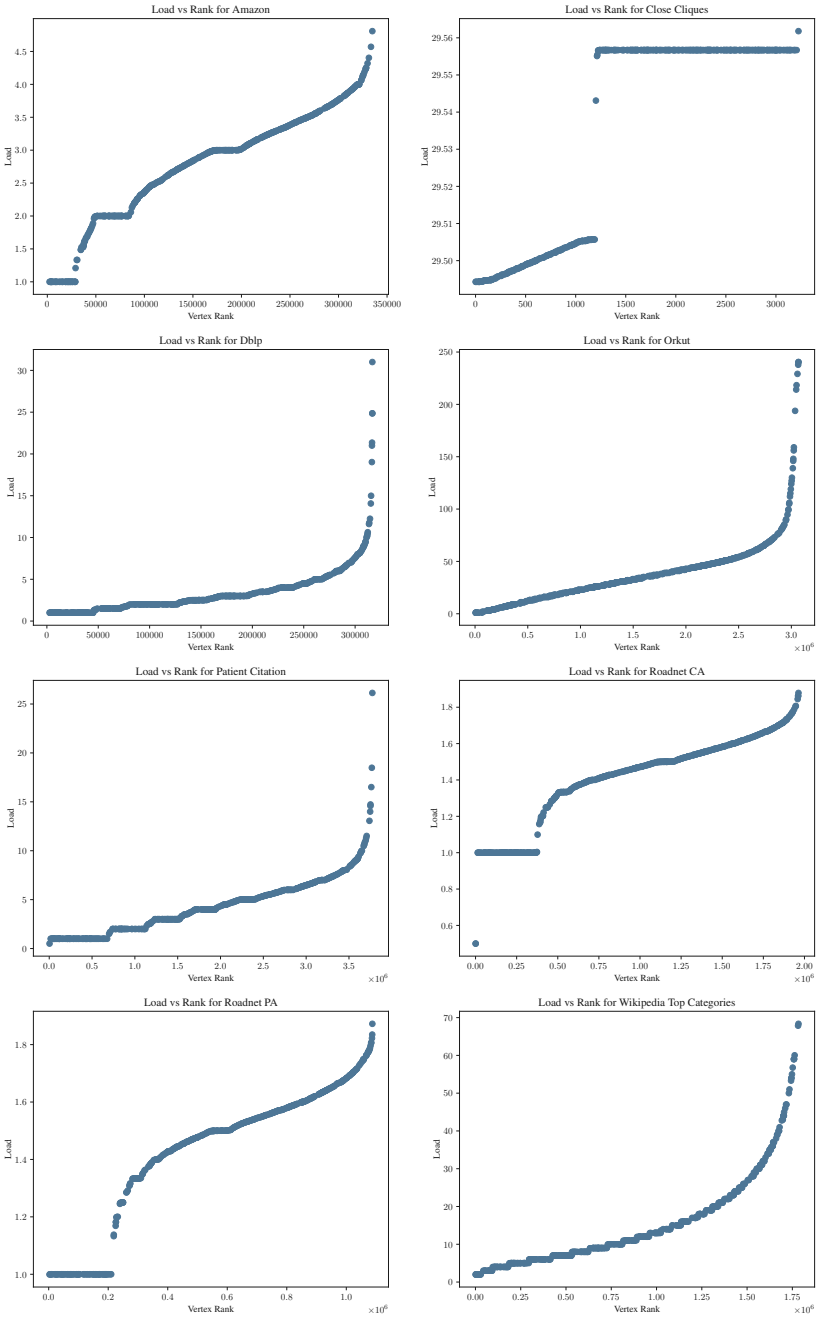


Figure 6.12: Scatter plot of sorted load vector. This can be used to approximate the densest at least $k$ subgraph.

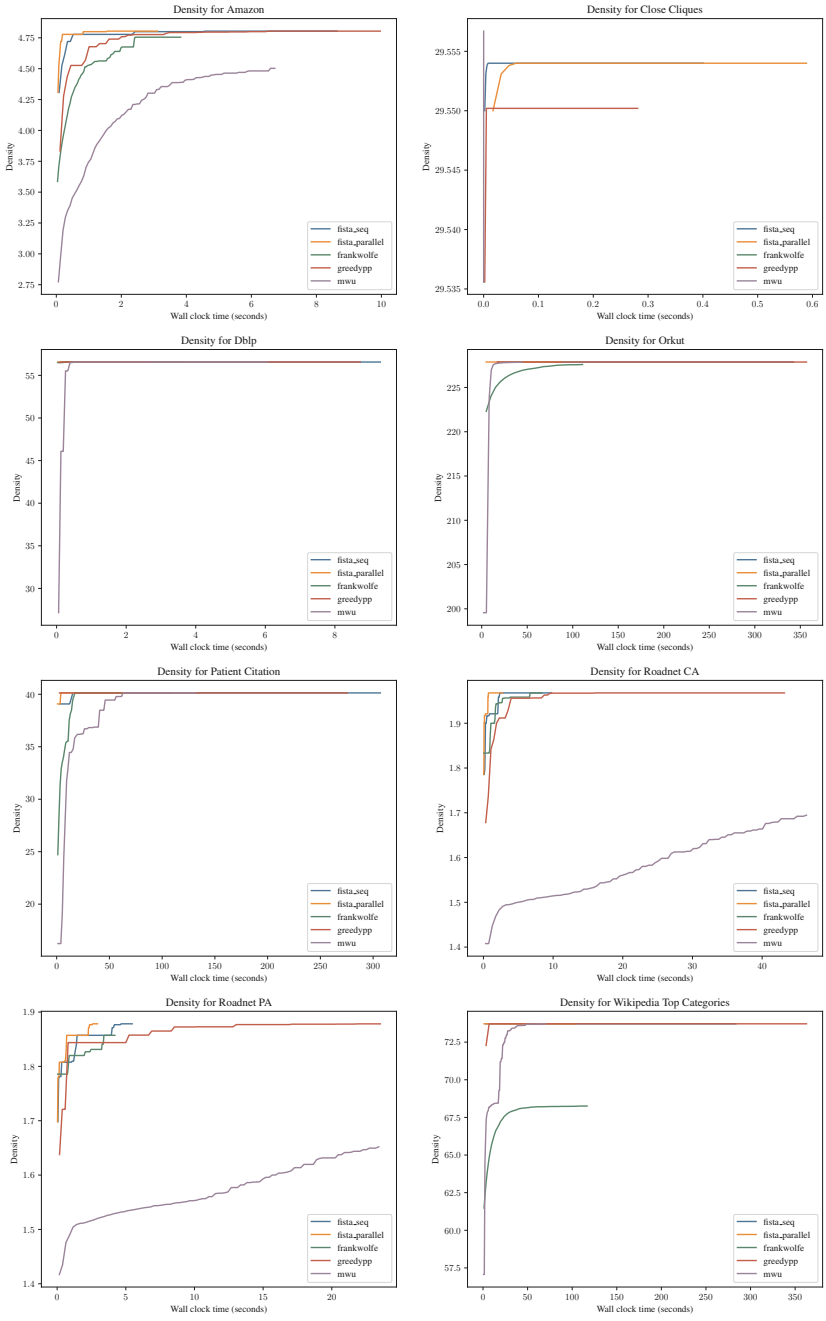


Figure 6.13: Wall clock time vs Maximum Density. See Figure 6.14 for a zoom in on first few seconds of each dataset.

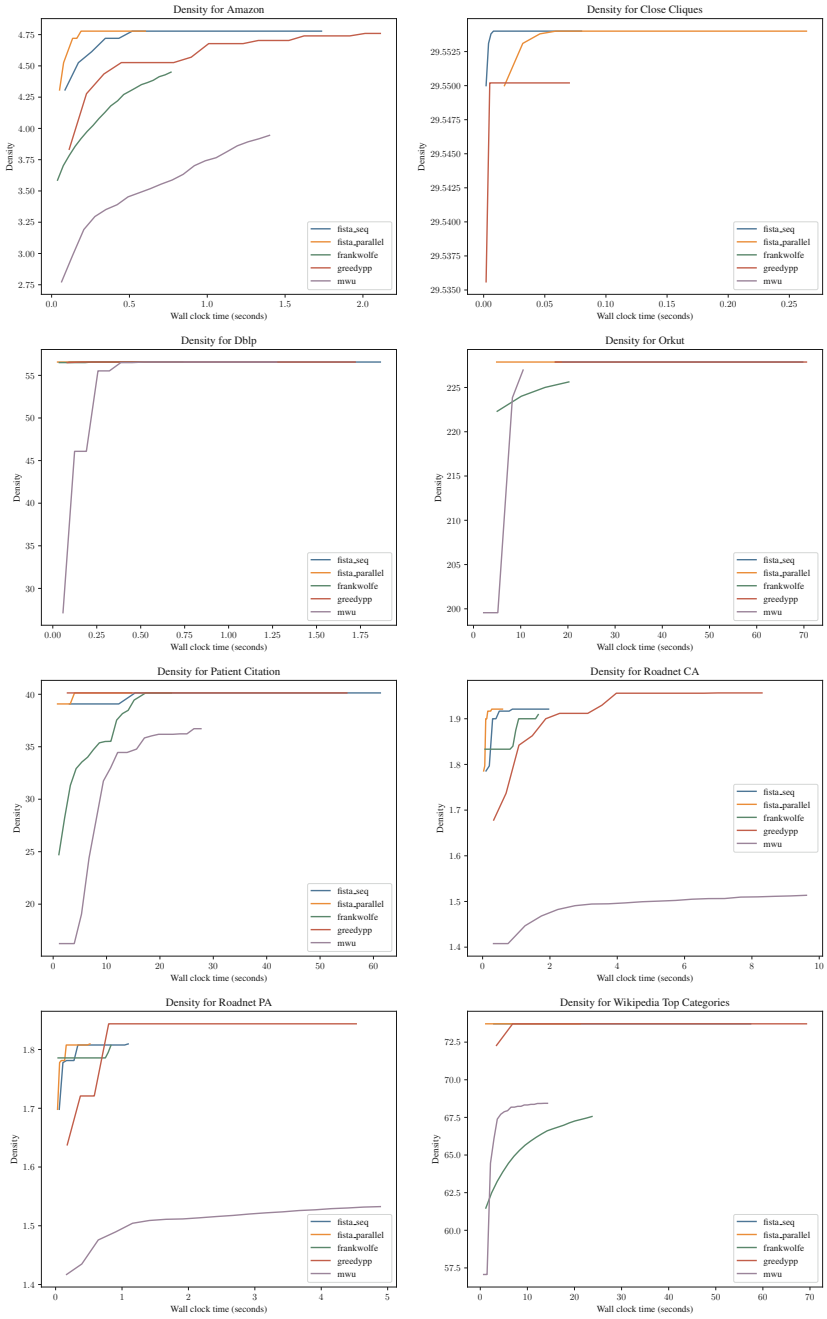


Figure 6.14: Wall clock time vs Maximum Density zoomed in on first few seconds for each dataset.