

Flowless: Extracting Densest Subgraphs Without Flow Computations

Digvijay Boob
Georgia Tech
digvijaybb40@gatech.edu

Saurabh Sawlani
Georgia Tech
sawlani@gatech.edu

Yu Gao
Georgia Tech
ygao380@gatech.edu

Charalampos E. Tsourakakis
Boston University
ctsourak@bu.edu

Richard Peng
Georgia Tech
rpeng@cc.gatech.edu

Di Wang
Google
wadi@google.com

Junxing Wang
CMU
junxingw@cs.cmu.edu

ABSTRACT

The problem of finding dense components of a graph is a major primitive in graph mining and data analysis. The *densest subgraph problem* (DSP) that asks to find a subgraph with maximum average degree forms a basic primitive in dense subgraph discovery with applications ranging from community detection to unsupervised discovery of biological network modules [16]. The DSP is exactly solvable in polynomial time using maximum flows [14, 17, 22]. Due to the high computational cost of maximum flows, Charikar’s greedy approximation algorithm is usually preferred in practice due to its linear time and linear space complexity [3, 8]. It constitutes a key algorithmic idea in scalable solutions for large-scale dynamic graphs [5, 7]. However, its output density can be a factor 2 off the optimal solution.

In this paper we design GREEDY++, an iterative peeling algorithm that improves upon the performance of Charikar’s greedy algorithm significantly. Our iterative greedy algorithm is able to output near-optimal and optimal solutions fast by adding a few more passes to Charikar’s greedy algorithm. Furthermore GREEDY++ is more robust against the structural heterogeneities (e.g., skewed degree distributions) in real-world datasets. An additional property of our algorithm is that it is able to assess *quickly*, without computing maximum flows, whether Charikar’s approximation quality on a given graph instance is closer to the worst case theoretical guarantee of $1/2$ or to optimality. We also demonstrate that our method has significant efficiency advantage over the maximum flow based exact optimal algorithm. For example, our algorithm achieves $\sim 145\times$ speedup on average across a variety of real-world graphs while finding subgraphs of density that are at least 90% as dense as the densest subgraph.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**; • **Theory of computation** → **Graph algorithms analysis**.

KEYWORDS

dense subgraph discovery, algorithm design, graph mining, applications

ACM Reference Format:

Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos E. Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting Densest Subgraphs Without Flow Computations. In *Proceedings of The Web Conference 2020 (WWW ’20)*, April 20–24, 2020, Taipei, Taiwan. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3366423.3380140>

1 INTRODUCTION

Finding dense components in graphs is a major topic in graph mining with diverse applications including DNA motif detection, unsupervised detection of interesting stories from micro-blogging streams in real time, indexing graphs for efficient distance query computation, and anomaly detection in financial networks, and social networks [16]. The *densest subgraph problem* (DSP) is one of the major formulations for dense subgraph discovery, where, given an undirected weighted graph $G(V, E, w)$ we want to find a set of nodes $S \subseteq V$ that maximizes the *degree density* $w(S)/|S|$, where $w(S)$ is the sum of the weights of the edges in the graph induced by S . When the weights are non-negative, the problem is solvable in polynomial time using maximum flows [17]. Since maximum flow computations are expensive despite the theoretical progress achieved over the recent years, Charikar’s greedy peeling algorithm is frequently used in practice [8]. This algorithm iteratively peels the lowest degree node from the graph, thus producing a sequence of subsets of nodes, of which it outputs the densest one. This simple, linear time and linear space algorithm provides a $1/2$ -approximation for the DSP. However, when the edge weights are allowed to be negative, the DSP becomes NP-hard [36].

Our work was originally motivated by a natural question: How can we quickly assess whether the output of Charikar’s algorithm on a given graph instance is closer to optimality or to the worst case

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.

WWW ’20, April 20–24, 2020, Taipei, Taiwan

© 2020 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.

ACM ISBN 978-1-4503-7023-3/20/04.

<https://doi.org/10.1145/3366423.3380140>

1/2-approximation guarantee? However, we ended up answering the following intriguing question that we state as the next problem:

Problem 1.1. Can we design an algorithm that performs (i) as well as Charikar’s greedy algorithm in terms of efficiency, and (ii) as well as the maximum flow-based exact algorithm in terms of output quality?

Contributions. The contributions of this paper are summarized as follows:

- We design a novel algorithm GREEDY++ for the densest subgraph problem, a major dense subgraph discovery primitive that “lies at the heart of large-scale data mining” [5]. GREEDY++ combines the best of two different worlds, the accuracy of the exact maximum flow based algorithm [14, 17], and the efficiency of Charikar’s greedy peeling algorithm [8].
- It is worth outlining that Charikar’s greedy algorithm typically performs better on real-world graphs than the worse case 1/2-approximation; on a variety of datasets we have tried, the worst case approximation was 0.8. Nonetheless, the only way to verify how close the output is to optimality relies on computing the exact solution using maximum flow. Our proposed method GREEDY++ can be used to assess the accuracy of Charikar’s algorithm in practice. Specifically, we find empirically that for all graph instances where GREEDY++ after a couple of iterations does not significantly improve the output density, the output of Charikar’s algorithm is near-optimal.
- We implement our proposed algorithm in C++ and apply it on a variety of real-world datasets. We verify the practical value of GREEDY++. Our empirical results indicate that GREEDY++ is a valuable addition to the toolbox of dense subgraph discovery; on real-world graphs, GREEDY++ is both fast in practice, and converges to a solution with an arbitrarily small approximation factor.

Notation. Let $G(V, E)$ be an undirected graph, where $|V| = n$, $|E| = m$. For a given subset of nodes $S \subset V$, $e[S]$ denotes the number of edges induced by S . When the graph is weighted, i.e., there exists a weight function $w : E \mapsto \mathbb{R}^+$, and $w(S)$ denotes the sum of the weights of the edges induced by S . We use $N(u)$ to define the set of neighbors of u , and $\deg(u) = |N(u)|$. We use $\deg_S(u)$ to denote u ’s degree in S , i.e., the number of neighbors of u within the set of nodes S . We use $\deg_{\max}$ to denote the maximum degree in G . Finally, the *degree density* $\rho(S)$ of a vertex set $S \subseteq V$ is defined as $\frac{e[S]}{|S|}$, or $\frac{w(S)}{|S|}$ when the graph is weighted.

2 RELATED WORK

Dense subgraph discovery. Detecting dense components is a major problem in graph mining. It is not surprising that many different notions of a dense subgraph are used in practice. The prototypical dense subgraph is a clique. However, the maximum clique problem is not only NP-hard, but also strongly inapproximable, see [19]. The notion of optimal quasi-cliques has been developed to detect subgraphs that are not necessarily fully interconnected but very dense [35]. However, finding optimal quasi-cliques is also NP-hard [21, 34]. Another popular and scalable approach to finding dense components is based on k -cores [12]. Recently, k -cores have also been used to detect anomalies in large-scale networks [15, 29].

The interested reader may refer to the recent survey by Gionis and Tsourakakis on the more broad topic of dense subgraph discovery [16]. In the following, we only provide a brief overview of work related to the densest subgraph problem.

Densest subgraph problem (DSP). The goal of the *densest subgraph problem* (DSP) is to find the set of nodes S which maximizes the degree density $\rho(S)$. The densest subgraph can be identified in polynomial time by solving a maximum flow problem [14, 17, 22]. Charikar [8] proved that the greedy algorithm proposed by Asashiro et al.¹ [3] produces a 1/2-approximation of the densest subgraph in linear time. To obtain fast algorithms with better approximation factors, McGregor et. al. [25], and Mitzenmacher et. al. [26] uniformly sparsified the input graph, and computed the densest subgraph in the resulting sparse graph. The first near-linear time algorithm for the DSP, given by Bahmani et. al. [4], relies on approximately solving the LP dual of the DSP. It is worth mentioning that Kannan and Vinay [20] gave a spectral $O(\log n)$ approximation algorithm for a related notion of density.

Charikar’s greedy peeling algorithm. Since our algorithm GREEDY++ is an improvement over Charikar’s greedy algorithm, we discuss the latter algorithm in greater detail. The algorithm removes in each iteration, the node with the smallest degree. This process creates a nested sequence of sets of nodes $V = S_n \supset S_{n-1} \supset S_{n-2} \supset \dots \supset S_1 \supset \emptyset$. The algorithm outputs the graph $G[S_j]$ that maximizes the degree density among $j = 1, \dots, n$. The pseudocode is shown in Algorithm 1.

Algorithm 1 GREEDY

Input: Undirected graph G

Output: A dense subgraph of G : G_{densest} .

```

1:  $G_{\text{densest}} \leftarrow G$ 
2:  $H \leftarrow G$ ;
3: while  $H \neq \emptyset$  do
4:   Find the vertex  $u \in H$  with minimum  $\deg_H(u)$ ;
5:   Remove  $u$  and all its adjacent edges  $uv$  from  $H$ ;
6:   if  $\rho(H) > \rho(G_{\text{densest}})$  then
7:      $G_{\text{densest}} \leftarrow H$ 
8:   end if
9: end while
10: Return  $G_{\text{densest}}$ .
```

Fast numerical approximation algorithms for DSP. Bahmani et. al. [4] approached the DSP via its dual problem, which in turn they reduced to $O(\log n)$ instances of solving a positive linear program. To solve these LPs, they employed the multiplicative weights update framework [2, 27] to achieve an ϵ -approximation in $O(\log n / \epsilon^2)$ iterations, where each iteration requires $O(m)$ work.

Notable extensions of the DSP. The DSP has been studied in weighted graphs, as well as directed graphs. When the edge weights are non-negative, both the maximum flow algorithm and Charikar’s greedy algorithm maintain their theoretical guarantees. In the presence of negative weights, the DSP in general becomes NP-hard [36]. For directed graphs Charikar [8] provided a linear programming

¹Despite the fact that the greedy algorithm was originally proposed in [3], it is widely known as Charikar’s greedy algorithm.

approach which requires the computation of n^2 linear programs and a $1/2$ -approximation algorithm which runs in $O(n^3 + n^2m)$ time. Khuller and Saha have provided more efficient implementations of the exact and approximation algorithms for the undirected and directed versions of the DSP [22]. Furthermore, Tsourakakis et al. recently extended the DSP to the k -clique, and the (p, q) -biclique densest subgraph problems [26, 33]. These extensions can be used for finding large near-cliques in general graphs and bipartite graphs respectively. The DSP has also been studied in the dynamic setting [7, 10, 28], the streaming setting [5, 7, 11, 25], and in the MapReduce computational model [5]. Bahmani, Goel, and Munagala use the multiplicative weights update framework [2, 27] to design an improved MapReduce algorithm [4]. We discuss this method in greater detail in Section 3. Tatti and Gionis [32] introduced a novel graph decomposition known as *locally-dense*, that imposes certain insightful constraints on the k -core decomposition. Further, efficient algorithms to find locally-dense subgraphs were developed by Danisch et al. [9].

We notice that in the DSP there are no restrictions on the size of the output. When restrictions on the size of S are imposed the problem becomes NP-hard. The densest- k -subgraph problem asks for find the subgraph S with maximum degree density among all possible sets S such that $|S| = k$. The state-of-the art algorithm is due to Bhaskara et al. [6], and provides a $O(n^{1/4+\epsilon})$ approximation in $O(n^{1/\epsilon})$ time. A long standing question is closing the gap between this upper bound and the lower bound. Other versions where $|S| \geq k$, $|S| \leq k$ have also been considered in the literature see [1].

3 PROPOSED METHOD

3.1 The GREEDY++ algorithm

As we discussed earlier, Charikar's peeling algorithm greedily removes the node of smallest degree from the graph, and returns the densest subgraph among the sequence of n subgraphs created by this procedure. While ties may exist, and are broken arbitrarily, for the moment it is useful to think as if Charikar's greedy algorithm produces a single permutation of the nodes, that naturally defines a nested sequence of subgraphs.

Algorithm description. Our proposed algorithm GREEDY++ iteratively runs Charikar's peeling algorithm, while keeping some information about the past runs. This information is crucial, as it results in different permutations, that naturally yield higher quality outputs. The pseudocode for GREEDY++ is shown in Algorithm 2. It takes as input the graph G , and a parameter T of the number of passes to be performed, and runs an iterative, weighted peeling procedure. In each round the load of each node is a function of its induced degree and the load from the previous rounds. It is worth outlining that the algorithm is easy to implement, as it is essentially T instances of Charikar's algorithm. What is less obvious perhaps, is why this algorithm makes sense, and works well. We answer this question in detail in Section 3.2.

Example. We provide a graph instance that clearly illustrates why GREEDY++ is a significant improvement over the classical greedy algorithm. We discuss the first two rounds of GREEDY++. Consider the following graph $G = B \cup \left(\bigcup_{i=1}^k H_i\right)$ where $B = K_{d,D}$ and $H_i = K_{d+2}$. Namely G is a disjoint union of a complete $d \times D$ bipartite

Algorithm 2 GREEDY++

Input: Undirected graph G , iteration count T

Output: An approximately densest subgraph of G : G_{densest} .

```

1:  $G_{\text{densest}} \leftarrow G$ 
2: Initialize the vertex load vector  $\ell^{(0)} \leftarrow 0 \in \mathbb{Z}^n$ ;
3: for  $i : 1 \rightarrow T$  do
4:    $H \leftarrow G$ ;
5:   while  $H \neq \emptyset$  do
6:     Find the vertex  $u \in H$  with minimum  $\ell_u^{(i-1)} + \deg_H(u)$ ;
7:      $\ell_u^{(i)} \leftarrow \ell_u^{(i-1)} + \deg_H(u)$ ;
8:     Remove  $u$  and all its adjacent edges  $uv$  from  $H$ ;
9:     if  $\rho(H) > \rho(G_{\text{densest}})$  then
10:       $G_{\text{densest}} \leftarrow H$ 
11:   end if
12: end while
13: end for
14: Return  $G_{\text{densest}}$ .
```

graph B , and of k $(d+2)$ -cliques $H_1, \dots, H_k$. Consider the case where $d \ll D, k \rightarrow +\infty$. G is pictured in Figure 1(a). The density of G is

$$\frac{2dD + (d+1)(d+2)k}{2d + 2D + 2k(d+2)} \rightarrow \frac{d+1}{2}.$$

Notice that this is precisely the density of any $(d+2)$ -clique. However, the density of B is $\frac{dD}{d+D} \approx d$, which is in fact the optimal solution. Charikar's algorithm outputs G itself, since it starts eliminating nodes of degree d from B , and by doing this, it never sees a subgraph with higher density. This example illustrates that the $\frac{1}{2}$ approximation is tight. Consider now a run of GREEDY++.

In its first iteration, it simply emulates Charikar's algorithm. The $D-d$ vertices of B which were eliminated first - each have load d . At this stage, our input is the disjoint union of k cliques and a $d \times d$ bipartite graph. Of the remaining $2d$ vertices in B , one vertex is charged with load d , two vertices each with loads $(d-1)$, $(d-2)$, $\dots$, 1 , and one vertex with load 0 . On the other hand, vertices in H_i are charged with loads $d+1, d, \dots, 0$. Figure 1(b) shows the cumulative degrees of vertices in G after one iteration of GREEDY++.

Without any loss of generality let us assume the vertex from B that got charged 0 originally had degree d . This vertex in the second iteration will get deleted first, and the vertex whose sum of load and degree is $d+1$ will get deleted second. But after these two, all the cliques get peeled away by the algorithm. This leaves us with a $d \times D - 2$ bipartite graph as the output after the second iteration, whose density is almost optimal.

Theoretical guarantees. Before we prove our theoretical properties for our proposed algorithm GREEDY++, it is worth outlining that experiments indicate that the performance of GREEDY++ is significantly better than the worst-case analysis we perform. Furthermore, we conjecture that our guarantees are not tight from a theoretical perspective; an interesting open question is to extend our analysis in Section 3.2 for GREEDY++ to prove that it provides asymptotically an optimal solution for the DSP. We conjecture that our algorithm is a $(1 + \frac{1}{\sqrt{T}})$ -approximation algorithm for the DSP. Our first lemma states that GREEDY++ is a 2-approximation algorithm for the DSP.

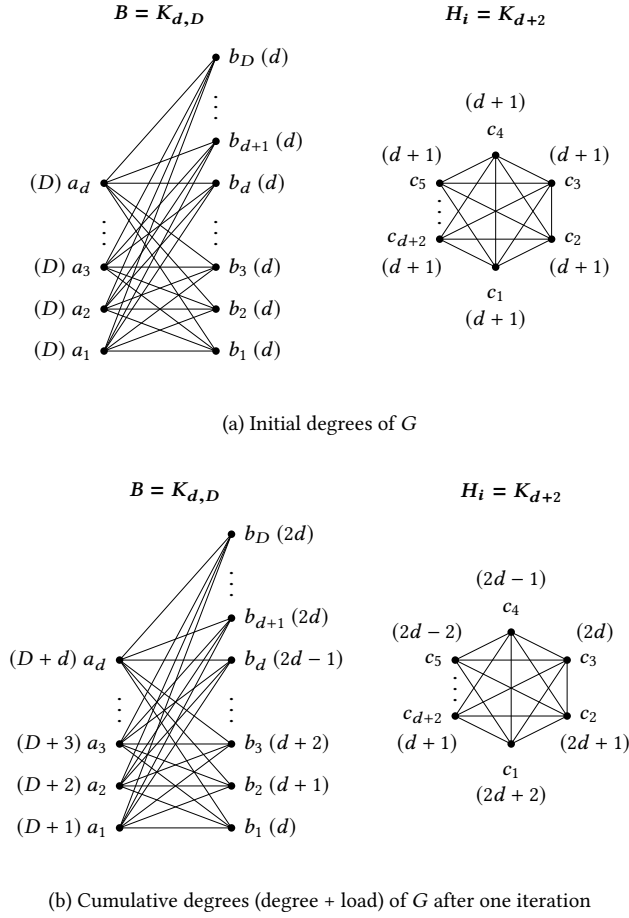


Figure 1: Illustration of two iterations of GREEDY++ on G . The output after one iteration is G itself (density $\approx (d+1)/2$), whereas the output after the second iteration is $B \setminus \{b_1, b_2\}$ (density $\approx d$).

LEMMA 3.1. Let G_{densest} the output of GREEDY++. Then, $\rho(G_{\text{densest}}) \geq \rho_G^*/2$, where ρ_G^* denotes the optimum value of the problem.

PROOF. Notice that the first iteration is identical to Charikar's 2-approximation algorithm, and G_{densest} is at least as dense as the output of the first iteration. $\square$

The next lemma provides bounds the quality of the dual solution, i.e., at each iteration the average load (average over the algorithm's iterations) assigned to any vertex is at most $2\rho_G^*$.

LEMMA 3.2. The following invariant holds for GREEDY++: for any vertex v and iteration i , $\ell_v^{(i)} \leq 2i \cdot \rho_G^*$.

PROOF. Consider the point at which vertex v is chosen in iteration i . Denote the graph at that instant to be $G_v^{(i)} = \langle V_v^{(i)}, E_v^{(i)} \rangle$.

First, let $i = 1$. The proof for this base case goes through identically as in [8].

$$\begin{aligned} \ell_v^{(1)} &= \deg_{G_v^{(1)}}(v) \leq \frac{1}{|V_v^{(1)}|} \sum_{u \in V_v^{(1)}} \deg_{G_v^{(1)}}(u) \\ &= \frac{2|E_v^{(1)}|}{|V_v^{(1)}|} \\ &= 2 \cdot \rho_{G_v^{(1)}} \leq 2 \cdot \rho_G^*. \end{aligned}$$

Now, assume that the statement is true for some iteration index $i-1$. For any vertex u at that point, the cumulative degree is $\ell_u^{(i-1)} + \deg_{G_v^{(i)}}(u)$. Since v has the minimum cumulative degree at that point,

$$\begin{aligned} \ell_v^{(i)} &= \ell_v^{(i-1)} + \deg_{G_v^{(i)}}(v) \leq \frac{1}{|V_v^{(i)}|} \sum_{u \in V_v^{(i)}} (\ell_u^{(i-1)} + \deg_{G_v^{(i)}}(u)) \\ &\leq 2(i-1)\rho_G^* + \frac{1}{|V_v^{(i)}|} \sum_{u \in V_v^{(i)}} \deg_{G_v^{(i)}}(u) \\ &\leq 2i \cdot \rho_G^*. \quad \square \end{aligned}$$

Running time. Finally, we bound the runtime of the algorithm as follows. The next lemma states that our algorithm can be implemented to run in $O((n+m) \cdot \min(\log n, T))$.

LEMMA 3.3. Each iteration of the above algorithm runs in time $O((n+m) \cdot \min(\log n, T))$.

PROOF. The deletion operation, along with assigning edges to a vertex and updating degrees takes $O(m)$ time since every edge is assigned once. Finding the minimum degree vertex can be implemented in two ways:

- (1) Since degrees in our algorithm can go from 0 to $2Tm$, we can create lists for each separate integer degree value. Now we need to scan each list from $\deg = 1$ to $\deg = 2Tm$. However, after deleting a vertex of degree d , we only need to scan from $d-1$ onwards. So the total time taken is $O(2Tm+n) = O(mT)$.
- (2) We can maintain a priority queue, which needs a total of $O(m)$ update operations, each taking $O(\log n)$ time. $\square$

Note that in the case of weighted graphs, we cannot maintain lists for each possible degree, and hence, it is necessary to use a priority queue.

3.2 Why does GREEDY++ work well?

Explaining the intuition behind GREEDY++ requires an understanding of the load balancing interpretation of Charikar's LP for the DSP [8], and the multiplicative weights update (MWU) framework by Plotkin, Shmoys and Tardos [27] used for packing/covering LPs. In the context of the DSP, the MWU framework was first used by Bahmani, Goel, and Munagala [4]. We include a self-contained exposition of the required concepts from [4, 8] in this section, that has a natural flow and concludes with our algorithmic contributions. Intuitively, the additional passes that GREEDY++ performs, improve the load balancing.

Charikar's LP and the load balancing interpretation. The following is a well-known LP formulation of the densest subgraph

problem, introduced in [8], which we denote by $\text{PRIMAL}(G)$. Binary variables x_u and y_e indicate the inclusion of a vertex u /edge e in an optimal densest subgraph. Relaxing the variables to real numbers and suitably scaling down the x_u values, we get the following LP, whose optimal objective value is known to be ρ_G^* .

$$\begin{aligned} & \text{maximize} && \sum_{e \in E} y_e \\ & \text{subject to} && y_e \leq x_u, & \forall e = uv \in E \\ & && y_e \leq x_v, & \forall e = uv \in E \\ & && \sum_{v \in V} x_v \leq 1, \\ & && y_e \geq 0, & \forall e \in E \\ & && x_v \geq 0, & \forall v \in V \end{aligned}$$

We then construct the dual LP for the above problem. Let $f_e(u)$ be the dual variable associated with the first $2m$ constraints of the form $y_e \leq x_u$, and let D be associated with the last constraint. We get the following LP, which we denote by $\text{DUAL}(G)$, and whose optimum is also ρ_G^* .

$$\begin{aligned} & \text{minimize} && D \\ & \text{subject to} && f_e(u) + f_e(v) \geq 1, & \forall e = uv \in E \\ & && \ell_v \stackrel{\text{def}}{=} \sum_{e \ni v} f_e(v) \leq D, & \forall v \in V \\ & && f_e(u) \geq 0, & \forall e = uv \in E \\ & && f_e(v) \geq 0, & \forall e = uv \in E \end{aligned}$$

This LP can be visualized as follows. Each edge $e = uv$ has a load of 1, which it wants to send to its end points: $f_e(u)$ and $f_e(v)$ such that the total load of any vertex v , ℓ_v , is at most D . The objective is to find the minimum D for which such a load assignment is feasible.

For a fixed D , the above dual problem can be framed as a flow problem on a bipartite graph as follows: Let the left side L represent V and the right side R represent E . Add a super-source s and edges from s to all vertices in L with capacity D . Add edges from $v \in V$ to $e \in E$ if e is incident on v in G . All vertices in R have demands of 1 unit. Although Goldberg's initial reduction [17] involved a different flow network, this graph can also be used to use maximum flow and use that to find the exact optimum to our problem. From strong duality, we know that the optimal objective values of both linear programs are equal, i.e., exactly ρ_G^* . Let ρ_G be the objective of any feasible solution to $\text{PRIMAL}(G)$. Similarly, let $\hat{\rho}_G$ be the objective of any feasible solution to $\text{DUAL}(G)$. Then, by optimality of ρ_G^* and weak duality, we obtain the optimality result $\rho_G \leq \rho_G^* \leq \hat{\rho}_G$.

Bahmani et al. [4] use the following covering LP formulation: decide the feasibility of constraints $f_e(u) + f_e(v) \geq 1$ for each edge $e = uv \in E$ subject to the polyhedral constraints:

$$\begin{aligned} & \sum_{e \ni v} f_e(v) \leq D, & \forall v \in V \\ & f_e(u) \geq 0, & \forall e = uv \in E \\ & f_e(v) \geq 0, & \forall e = uv \in E \end{aligned}$$

The width of this linear program is the maximum value of $f_e(u) + f_e(v)$ provided that $f_e(u), f_e(v)$ satisfy the constraints of the program. Bahmani et al. in order to provably bound the width of the above LP, they introduce another set of simple constraints

as follows:

$$\begin{aligned} & \sum_{e \ni v} f_e(v) \leq D, & \forall v \in V \\ & q \geq f_e(u) \geq 0, & \forall e = uv \in E \\ & q \geq f_e(v) \geq 0, & \forall e = uv \in E \end{aligned}$$

where $q \geq 1$ is a small constant. So, for a particular value of D , they verify the approximate feasibility of the covering problem using the MWU framework. However, this necessitates running a binary search over all possible values of D and finding the lowest value of D for which the LP is feasible. Since the precision for D can be as low as ϵ , this binary search is inefficient in practice. Furthermore, due to the added ℓ_∞ constraint to bound the width, extracting the primal solution (i.e. an approximately densest subgraph) from the dual is no longer straightforward, and the additional rounding step to overcome this incurs additional loss in the approximation factor.

In order to overcome these practical issues, we propose an alternate MWU formulation which sacrifices the width bounds but escapes the binary search phase over D . Eliminating the artificial width bound makes it straightforward to extract a primal solution. Moreover, our experiments on real world graphs suggest that width is not a bottleneck for the running time of the MWU algorithm. Even more importantly, our alternate formulation naturally yields GREEDY++ as we explain in the following.

Our MWU formulation. We can denote the LP $\text{DUAL}(G)$ succinctly as follows:

$$\begin{aligned} & \text{minimize} && D \\ & \text{subject to} && \mathbf{B}\mathbf{f} \leq D\mathbf{1} \\ & && \mathbf{f} \in \mathcal{P} \end{aligned}$$

where $\mathbf{f}$ is the vector representation of the all $f_e(v)$ variables, $\mathbf{B} \in \mathbb{R}^{n \times 2m}$ is the matrix denoting the left hand side of all constraints of the form $\sum_{e \ni v} f_e(v) \leq D$. $\mathbf{1}$ denotes the vector of 1's and $\mathcal{P}$ is a polyhedral constraint set defined as follows:

$$\begin{aligned} & f_e(u) + f_e(v) \geq 1 & \forall e = uv \in E \\ & f_e(u) \geq 0 & \forall e \in E, \forall v \in e. \end{aligned}$$

Note that for any $\mathbf{f} \in \mathcal{P}$, we have that the minimum D satisfying $\mathbf{B}\mathbf{f} \leq D\mathbf{1}$ is equal to $\|\mathbf{B}\mathbf{f}\|_\infty$. This follows due to the non-negativity of $\mathbf{B}\mathbf{f}$ for any $\mathbf{f} \in \mathcal{P}$. Now a simple observation shows that for any non-negative vector $\mathbf{y}$, we can write

$$\|\mathbf{y}\|_\infty = \max_{\mathbf{x} \in \Delta_n^+} \mathbf{x}^T \mathbf{y}$$

where $\Delta_n^+ := \{\mathbf{x} \geq \mathbf{0} : \mathbf{1}^T \mathbf{x} \leq 1\}$. Hence, we can now write $\text{DUAL}(G)$ as:

$$\begin{aligned} \min_{\mathbf{f} \in \mathcal{P}} \|\mathbf{B}\mathbf{f}\|_\infty &= \min_{\mathbf{f} \in \mathcal{P}} \max_{\mathbf{x} \in \Delta_n^+} \mathbf{x}^T \mathbf{B}\mathbf{f} \\ &= \max_{\mathbf{x} \in \Delta_n^+} \min_{\mathbf{f} \in \mathcal{P}} \mathbf{x}^T \mathbf{B}\mathbf{f}. \end{aligned} \quad (1)$$

Here the last equality follows due to strong duality of the convex optimization.

The "inner" minimization part of (1) can be performed easily. In particular, we need an oracle which, given a vector $\mathbf{x}$, solves

$$C(\mathbf{x}) = \min_{\mathbf{f} \in \mathcal{P}} \sum_{e=uv} x_u f_e(u) + x_v f_e(v).$$

LEMMA 3.4. *Given a vector $\mathbf{x}$, $C(\mathbf{x})$ can be computed in $O(m)$ time.*

PROOF. For each edge $e = uv$, simply check which of x_u and x_v is smaller. WLOG, assume it is x_u . Then, set $f_e(u) = 1$ and $f_e(v) = 0$. $\square$

We denote the optimal $\mathbf{f}$ for a given $\mathbf{x}$ as $\mathbf{f}(\mathbf{x})$. Now, using the above oracle, we can apply the MWU algorithm to the “outer” problem of (1), i.e., $\max_{\mathbf{x} \in \Delta_n^+} C(\mathbf{x})$. Additionally, to apply the MWU framework, we need to estimate the width of this linear program. The width for (1) can be bounded by largest degree, $d_{\max}$ of the graph G . Indeed, we see in Lemma 3.4 that $\mathbf{f}(\mathbf{x})$ is a 0/1 vector. In that case, $\|\mathbf{B}\mathbf{f}(\mathbf{x})\|_\infty \leq d_{\max}$.

We conclude our analysis of this alternative dual formulation of the DSP with the following theorem.

THEOREM 3.5. *Our alternative dual formulation admits a MWU algorithm that outputs an $\mathbf{f} \in \mathcal{P}$ such that $\|\mathbf{B}\mathbf{f}\|_\infty \leq (1 + \epsilon)\rho_G^*$.*

For the sake of completeness, we detail the MWU algorithm and the proof of Theorem 3.5 in Appendix A.

Let us now view Charikar’s peeling algorithm in the context of this dual problem. In a sense, the greedy peeling algorithm resembles one “inner” iteration of the MWU algorithm, where whenever a vertex is removed, its edges assign their load to it. Keeping this in mind, we designed GREEDY++ to add “outer” iterations to the peeling algorithm, thus improving the approximation factor arbitrarily with increase in iteration count. By weighting vertices using their load from previous iterations, GREEDY++ implicitly performs a form of load balancing on the graph, thus arriving at a better dual solution.

4 EXPERIMENTS

4.1 Experimental setup

The experiments were performed on a single machine, with an Intel(R) Core(TM) i7-2600 CPU at 3.40GHz (4 cores), 8MB cache size, and 8GB of main memory. We find densest subgraphs on the samples using binary search and maximum flow computations. The flow computations were done using C++ implementations of the push-relabel algorithm [18], HiPR². We have implemented our algorithm GREEDY++ and Charikar’s greedy algorithm in C++. Our implementations are efficient and our code is available publicly³.

We use a variety of datasets obtained from the Stanford’s SNAP database [24], ASU’s Social Computing Data Repository [37], BioGRID [31] and from the Koblenz Network Collection [23], that are shown in table Table 1. A majority of the datasets are from SNAP, and hence we mark only the rest with their sources. Multiple edges, self-loops are removed, and directionality is ignored for directed graphs. The first cluster of datasets are unweighted graphs. The largest unweighted graph is the *web-trackers* graph with roughly 141M edges, while the smallest unweighted graph has roughly 25K edges. For weighted graphs, we use a set of Twitter graphs that were crawled during the first week of February 2018 [30]. Finally, we use a set of signed networks (*slashdot*, *epinions*). We remind the reader that while the DSP is NP-hard on signed graphs, Charikar’s

²HiPR is available at <http://www.avglab.com/andrew/soft/hipr.tar>

³Our code for GREEDY++ and the exact algorithm is available at the anonymous link <https://www.dropbox.com/s/jzouo9fjoytyqg3/code-greedy%2B%2B.zip?dl=0>

Name	n	m
web-trackers [23]	40 421 974	140 613 762
orkut [23]	3 072 441	117 184 899
livejournal-affiliations [23]	10 690 276	112 307 385
wiki-topcats	1 791 489	25 447 873
cit-Patents	3 774 768	16 518 948
actor-collaborations [23]	382 219	15 038 083
ego-gplus	107 614	12 238 285
dblp-author	5 425 963	8 649 016
web-BerkStan	685 230	6 649 470
flickr [37]	80 513	5 899 882
wiki-Talk	2 394 385	4 659 565
web-Google	875 713	4 322 051
com-youtube	1 134 890	2 987 624
roadNet-CA	1 965 206	2 766 607
web-Stanford	281 903	1 992 636
roadNet-TX	1 379 917	1 921 660
roadNet-PA	1 088 092	1 54 898
Ego-twitter	81 306	1 342 296
com-dblp	317 080	1 049 866
com-Amazon	334 863	925 872
soc-slashdot0902	82 168	504 230
soc-slashdot0811	77 360	469 180
soc-Epinions	75 879	405 740
blogcatalog [37]	10,312	333 983
email-Enron	36 692	183 831
ego-facebook	4 039	88 234
ppi [31]	3 890	37 845
twitter-retweet [30]	316 662	1 122 070
twitter-favorite [30]	226 516	1 210 041
twitter-mention [30]	571 157	1 895 094
twitter-reply [30]	196 697	296 194
soc-sign-slashdot081106	77 350	468 554
soc-sign-slashdot090216	81 867	497 672
soc-sign-slashdot090221	82 140	500 481
soc-sign-epinions	131 828	711 210

Table 1: Datasets used in our experiments.

algorithm does provide certain theoretical guarantees, see Theorem 2 in [36].

4.2 Experimental results

Before we delve in detail into our experimental findings, we summarize our key findings here:

- Our algorithm GREEDY++ when given enough number of iterations *always* finds the optimal value, and the densest subgraph. This agrees with our conjecture that running T iterations of GREEDY++ gives a $1 + O(1/\sqrt{T})$ approximation to the DSP.
- Experimentally, Charikar’s greedy algorithm always achieves at least 80% accuracy, and occasionally finds the optimum.
- For graphs on which the performance of Charikar’s greedy algorithm is optimal, the first couple of iterations of GREEDY++

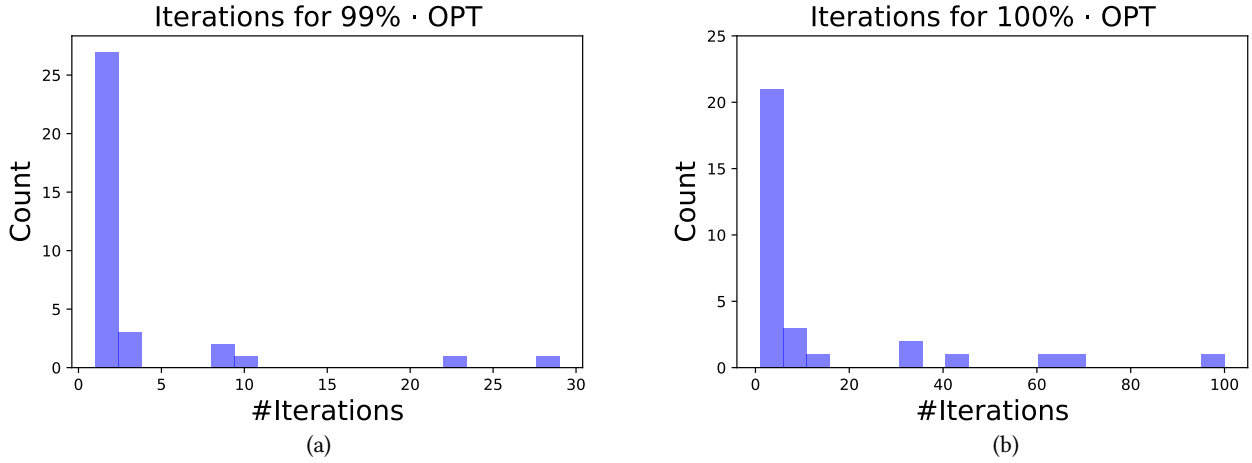


Figure 2: Number of iterations for GREEDY++. Histograms of number of iterations to reach (a) 99% of the optimum degree density, (b) the optimum degree density.

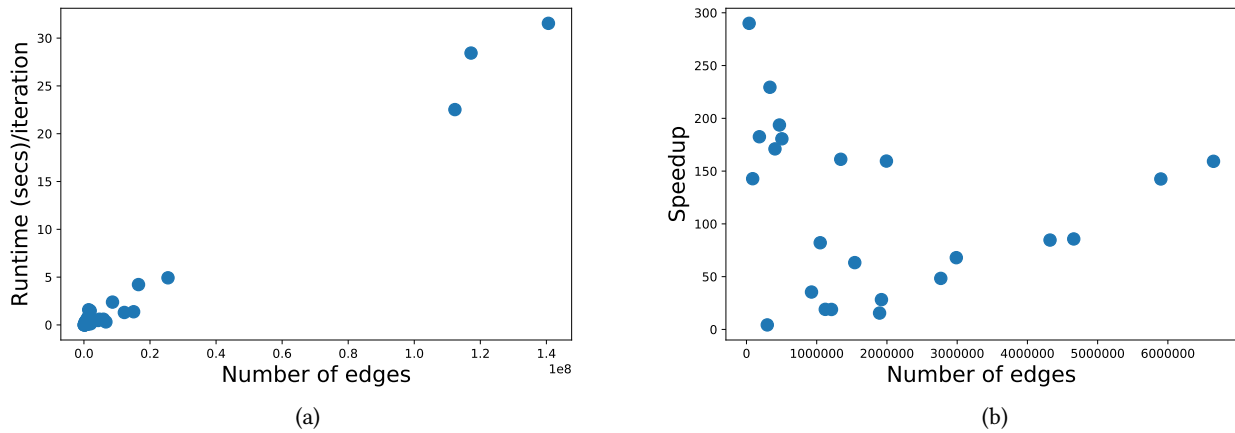


Figure 3: Scalability. (a) Running time in seconds of each iteration of GREEDY++ versus the number of edges. (b) Speedup achieved by GREEDY++ over the *exact max-flow based algorithm*, plotted vs. number of edges in the graph. Specifically, the *y*-axis is the ratio of the run time of the exact max flow algorithm divided by the run time of GREEDY++ that finds 90% of the optimal solution.

suffice to deduce convergence safely, and thus act in practice as a certificate of optimality. This is the first method to the best of our knowledge that can be used to infer quickly the actual approximation of Charikar’s algorithm on a given graph instance.

- When Charikar’s algorithm does not yield an optimal solution, then GREEDY++ within few iterations is able to increase the accuracy to 99% of the optimum density, and by adding a few more iterations is able to find the optimal density and extract an optimal output.
- When we are able to run the exact algorithm (for graphs with more than 8M edges, the maximum flow code crashes) on our machine, the average speedup that our algorithm provides to reach *the optimum* is 144.6× on average, with a standard deviation equal to 57.4. The smallest speedup observed was

67.9×, and the largest speedup 290×. Additionally, we remark that the exact algorithm is only able to find solutions up to an accuracy of 10^{-3} on most graphs.

- The speedup typically increases as the size of the graph increases. In fact, the maximum flow exact algorithm cannot complete on the largest graphs we use.
- The maximum number of iterations needed to reach 90% of the optimum is at most 3, i.e., by running two more passes compared to Charikar’s algorithm, we are able to boost the accuracy by 10%.
- The same remarks hold for both weighted and unweighted graphs.

Number of iterations. We first measure how many iterations we need to reach 99% of the optimum, or even the optimum. Figures 2(a), (b) answer these questions respectively. We observe the

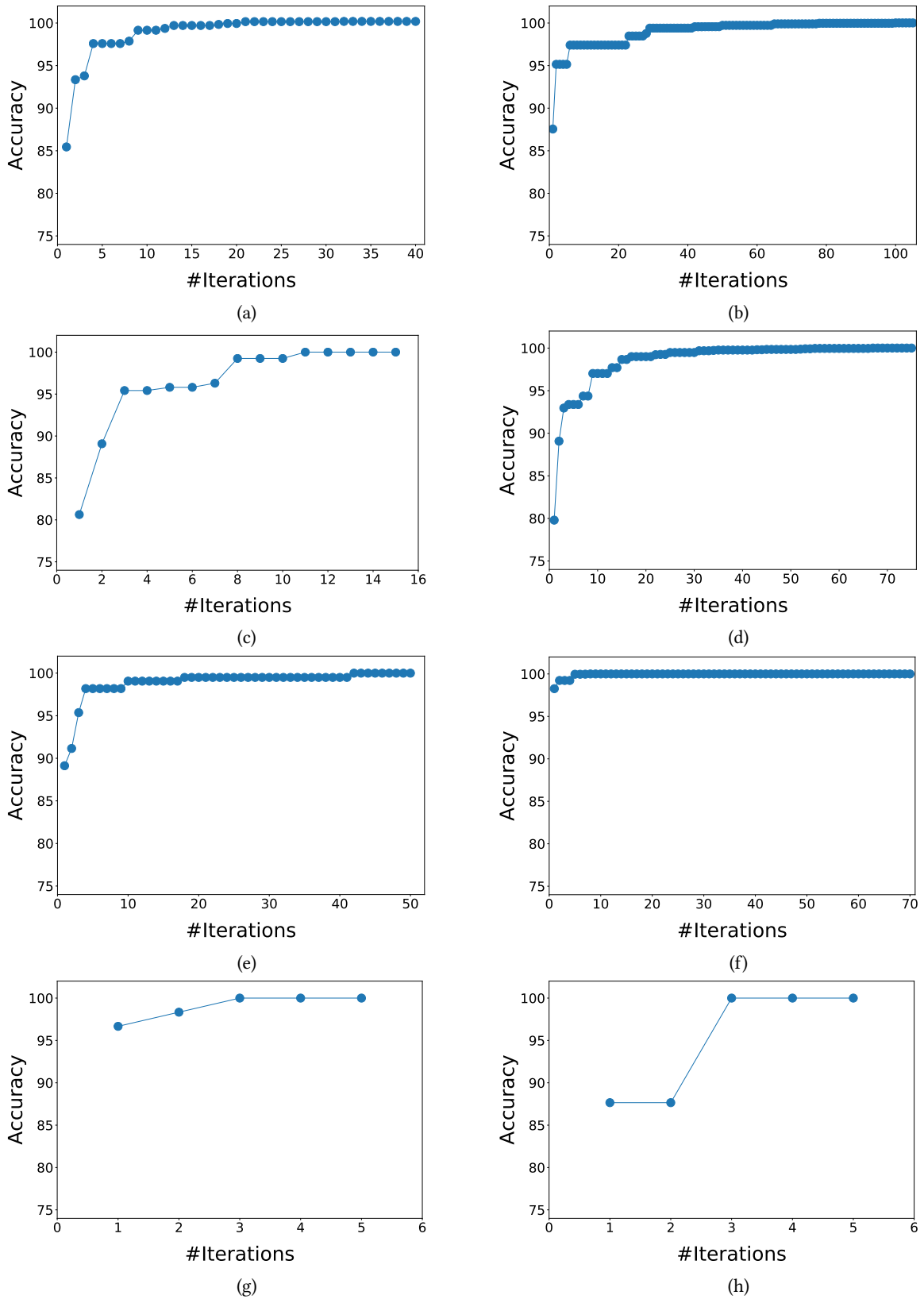


Figure 4: Convergence to optimum as a function of the number of iterations of GREEDY++. (a) roadNet-CA, (b) roadNet-PA, (c) roadNet-TX, (d) com-Amazon, (e) dblp-author, (f) ego-twitter, (g) twitter-favorite, (h) twitter-reply. Here, the *accuracy* is given by $\frac{\rho(H_i)}{\rho_G^*}$, where H_i is the output of GREEDY++ after i iterations.

impressive performance of Charikar’s greedy algorithm; for the majority of the graph instances we observe that it finds a near-optimal densest subgraph. Nonetheless, even for those graph instances –as we have emphasized earlier– our algorithm GREEDY++ acts as a certificate of optimality. Namely, we observe that the objective remains the same after a couple of iterations if and only if the algorithm has reached the optimum. For the rest of the graphs where Charikar’s greedy algorithm outputs an approximation greater than 80% but less than 99%, we observe the following: for five datasets it takes at most 3 iterations, for one graph it takes nine iterations, and then there exist three graphs for which GREEDY++ requires 10, 22, and 29 iterations respectively. If we insist on finding the optimum densest subgraph, we observe that the maximum number of iterations can go up to 100. On average, GREEDY++ requires 12.69 iterations to reach the optimum densest subgraph.

Scalability. Our experiments verify the intuitive facts that (i) each iteration of the greedy algorithm runs fast, and (ii) the exact algorithm that uses maximum flows is comparatively slow. We constrain ourselves on the set of data for which we were able to run the exact algorithm. Figure 3(a) shows the time that each iteration of the GREEDY++ takes on average (runtimes are well concentrated around the average) over the iterations performed to reach the optimal densest subgraph. Figure 3(b) shows the speedup achieved by our algorithm when we condition on obtaining *at least* 90% (notice that frequently the actual accuracy is greater than 95%) of the optimal solution versus the exact max-flow based algorithm. Specifically, we plot the ratio of the running times of the exact algorithm by the time of GREEDY++ versus the number of edges. Notice that for small graphs, the speedups are very large, then they drop, and they exhibit an increasing trend as the graph size grows. For the largest graphs in our collection, the exact algorithm is infeasible to run on our machine.

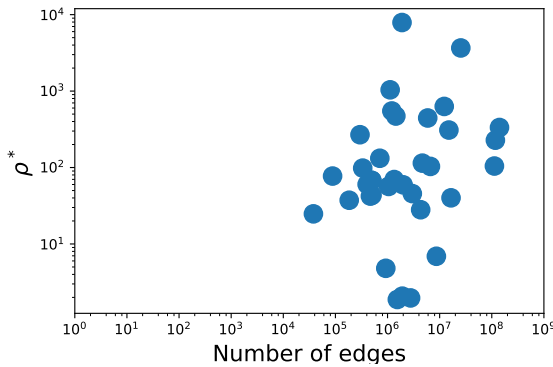


Figure 5: Log-log plot of optimal degree density ρ^* versus the number of edges in the graph.

Convergence. Figure 4 illustrates the convergence of GREEDY++ for various datasets. Specifically, each figure plots the accuracy of GREEDY++ after T iterations versus T . The accuracy is measured as the ratio of the degree density achieved by GREEDY++ by the optimal degree density. Figures 4(a),(b),(c),(d),(e),(f),(g),(h) correspond to the

convergence behavior of roadNet-CA, roadNet-PA, roadNet-TX, com-Amazon, dblp-author, ego-twitter, twitter-favorite, twitter-reply respectively. These plots illustrate various interesting properties of GREEDY++ in practice. Observe Figure 4(e). Notice how GREEDY++ keeps outputting the same subgraph for few consecutive iterations, but then suddenly around the 10th iteration it “jumps” and finds an even denser subgraph. Recall that on average over our collection of datasets for which we can run the exact algorithm (i.e., datasets with less than 8M edges), GREEDY++ requires roughly 12 iterations to reach the optimum densest subgraph. For this reason we suggest running GREEDY++ for that many iterations in practice. Furthermore, we typically observe an improvement over the first pass, with the exception of the weighted graph twitter-reply, where the “jump” happens at the end of the third iteration.

Anomaly detection. It is worth outlining that GREEDY++ provides a way to compute the densest subgraph in graphs where the maximum flow approach does not scale. For example, for graphs with more than 8 million edges, the exact method does not run on our machine. By running GREEDY++ for enough iterations we can compute a near-optimal or the optimal solution. This allows us to compute a proxy of ρ^* for the largest graphs, like orkut and trackers. We examined to what extent there exists a pattern between the size of the graph and the optimal density. In contrast to the power law relationship between the k -cores and the graph size claimed in [29], we do not observe a similar power law when we plot ρ^* (the exact optimal value or the proxy value found by GREEDY++ after 100 iterations for the largest graphs) versus the number of edges in the graph. This is shown in Figure 5. Part of the reason why we do not observe such a law are anomalies in graphs. For instance, we observe that small graphs may contain extremely dense subgraphs, thus resulting in significant outliers.

5 CONCLUSION

In this paper we provide a powerful algorithm for the *densest subgraph problem*, a popular and important objective for discovering dense components in graphs. The main practical value of our GREEDY++ algorithm is two-fold. First, by running few more iterations of Charikar’s greedy algorithm we obtain (near-)optimal results that can be obtained using only maximum flows. Second, GREEDY++ can be used to answer for first time the question “Is the approximation of Charikar’s algorithm on this graph instance closer to $1/2$ or to 1 ?” without computing the optimal density using maximum flows. Empirically, we have verified that GREEDY++ combines the best of “two worlds” on real data, i.e., the efficiency of the greedy peeling algorithm, and the accuracy of the exact maximum flow algorithm. We believe that GREEDY++ is a valuable addition to the algorithmic toolbox for dense subgraph discovery that combines the best of two worlds, i.e., the accuracy of maximum flows, and the time and space efficiency of Charikar’s greedy algorithm.

An intriguing open question which remains from our work is a theoretical convergence proof for GREEDY++. In addition to the empirical evidence towards convergence, we believe that our algorithm mimics a linear programming solution given by the multiplicative-weight-update framework [2, 27]. Hence, we state this open problem as the following conjecture:

Conjecture 5.1. GREEDY++ is a $1 + O(1/\sqrt{T})$ approximation algorithm for the DSP, where T is the number of iterations it performs.

REFERENCES

- [1] 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.
- [2] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing* 8, 1 (2012), 121–164.
- [3] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 2000. Greedily finding a dense subgraph. *Journal of Algorithms* 34, 2 (2000), 203–221.
- [4] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient primal-dual graph algorithms for mapreduce. In *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 59–78.
- [5] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment* 5, 5 (2012), 454–465.
- [6] Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. 2010. Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k -subgraph. In *Proceedings of the forty-second ACM symposium on Theory of computing*. ACM, 201–210.
- [7] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 173–182.
- [8] 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.
- [9] 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, WWW 2017, Perth, Australia, April 3-7, 2017*. 233–242. DOI: <http://dx.doi.org/10.1145/3038912.3052619>
- [10] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 300–310.
- [11] Hossein Esfandiari, MohammadTaghi Hajiaghayi, and David P. Woodruff. 2015. Applications of uniform sampling: Densest subgraph and beyond. *arXiv preprint arXiv:1506.04505* (2015).
- [12] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and Streaming Algorithms for K -Core Decomposition. *arXiv preprint arXiv:1808.02546* (2018).
- [13] Yoav Freund and Robert E. Schapire. 1996. Game Theory, On-line Prediction and Boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory (COLT '96)*. ACM, New York, NY, USA, 325–332. DOI: <http://dx.doi.org/10.1145/238061.238163>
- [14] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* 18, 1 (1989), 30–55.
- [15] Christos Giatsidis, Fragkiskos Malliaros, Dimitrios Thilikos, and Michalis Vazirgiannis. 2014. Corecluster: A degeneracy based graph clustering framework. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.
- [16] Aristides Gionis and Charalampos E. Tsourakakis. 2015. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2313–2314.
- [17] Andrew V. Goldberg. 1984. *Finding a maximum density subgraph*. University of California Berkeley, CA.
- [18] Andrew V. Goldberg and Robert E. Tarjan. 1988. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)* 35, 4 (1988), 921–940.
- [19] J. Hästad. 1999. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica* 182, 1 (1999).
- [20] Ravi Kannan and V. Vinay. 1999. *Analyzing the structure of large graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn.
- [21] Yasushi Kawase and Atsushi Miyauchi. 2018. The densest subgraph problem with a convex/concave size function. *Algorithmica* 80, 12 (2018), 3461–3480.
- [22] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *International Colloquium on Automata, Languages, and Programming*. Springer, 597–608.
- [23] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, New York, NY, USA, 1343–1350. DOI: <http://dx.doi.org/10.1145/2487788.2488173>
- [24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. (June 2014). <http://snap.stanford.edu/data>
- [25] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. 2015. Densest subgraph in dynamic graph streams. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 472–482.
- [26] 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 *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 815–824.
- [27] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research* 20, 2 (1995), 257–301.
- [28] Saurabh Sawlani and Junxing Wang. 2019. Near-Optimal Fully Dynamic Densest Subgraph. *arXiv preprint arXiv:1907.03037* (2019).
- [29] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. Corescope: graph mining using k -core analysis: patterns, anomalies and algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 469–478.
- [30] Konstantinos Sotiropoulos, John W. Byers, Polyvios Pratikakis, and Charalampos E. Tsourakakis. 2019. TwitterMancer: Predicting Interactions on Twitter Accurately. *arXiv preprint arXiv:1904.11119* (2019).
- [31] Chris Stark, Bobby-Joe Breitkreutz, Teresa Regul, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. 2006. BioGRID: A general repository for interaction datasets. *Nucleic acids research* 34 (01 2006), D535–9. DOI: <http://dx.doi.org/10.1093/nar/gkj109>
- [32] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly Graph Decomposition. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. 1089–1099. DOI: <http://dx.doi.org/10.1145/2736277.2741119>
- [33] Charalampos Tsourakakis. 2015. The k -clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 1122–1132.
- [34] Charalampos Tsourakakis. 2015. Streaming graph partitioning in the planted partition model. In *Proceedings of the 2015 ACM on Conference on Online Social Networks*. ACM, 27–35.
- [35] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsirlis. 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*. ACM, 104–112.
- [36] Charalampos E. Tsourakakis, Tianyi Chen, Naonori Kakimura, and Jakub Pachocki. 2019. Novel Dense Subgraph Discovery Primitives: Risk Aversion and Exclusion Queries. *arXiv preprint arXiv:1904.08178* (2019).
- [37] R. Zafarani and H. Liu. 2009. Social Computing Data Repository at ASU. (2009). <http://socialcomputing.asu.edu>

A MULTIPLICATIVE WEIGHTS UPDATE ALGORITHM

In this section, we give an algorithm to solve the zero-sum game $\max_{\mathbf{x} \in \Delta_n} \min_{\mathbf{f} \in \mathcal{P}} \mathbf{x}^T \mathbf{B} \mathbf{f}$, which corresponds to solving the dual of the densest subgraph problem, as described in Section 3.2. Given that we have an oracle access to $\min_{\mathbf{f} \in \mathcal{P}} \mathbf{x}^T \mathbf{B} \mathbf{f}$, we can use the multiplicative weights update framework to get an ϵ -approximation of the game [13].

The pseudocode for the MWU algorithm is shown in Algorithm 3.

To prove the convergence of Algorithm 3, we use the following theorem from [2]. We modify it slightly to accommodate for the fact that the width of the DSP, $\|\mathbf{B} \mathbf{f}(x)\|_\infty$, can be at most $\deg_{\max}$. In other words, the oracle can assign at most $\deg_{\max}$ edges to any particular vertex.

LEMMA A.1 (THEOREM 3.1 FROM [2]). *Given an error parameter ϵ , there is an algorithm which solves the zero-sum game up to an additive factor of ϵ using $O(W \log n/\epsilon^2)$ calls to ORACLE, with an additional processing time of $O(n)$ per call, where W is the width of the problem.*

Using the fact that our ORACLE runs in $O(m)$ time (from Lemma 3.4), and using $W = \deg_{\max}$, we get the following corollary.

Algorithm 3 Multiplicative Weight Update Algorithm**Input:** Matrix $\mathbf{B}$, approximation factor ε .**Output:** An approximate solution to the zero-sum game.

- 1: Initialize the weight vector as $w_i^{(1)} \leftarrow 1$ for all $i \in [n]$
- 2: Initialize $\eta \leftarrow \varepsilon/2 \deg_{\max}$
- 3: **for** $t : 1 \rightarrow T$ **do**
- 4: $x_i^{(t)} \leftarrow w_i^{(t)} / \|\mathbf{w}^{(t)}\|_1$ for all $i \in [n]$.
- 5: Find $\mathbf{f}(\mathbf{x}^{(t)})$ using Oracle($\mathbf{x}^{(t)}$).
- 6: Set $C(\mathbf{x}^{(t)}) \leftarrow (\mathbf{x}^{(t)})^T \mathbf{B} \mathbf{f}(\mathbf{x}^{(t)})$
- 7: Let $\mathbf{b}_i^T \mathbf{f}(\mathbf{x}^{(t)})$ be the i -th element in $\mathbf{B} \mathbf{x}^{(t)}$.
- 8: Update the weights as

$$w_i^{(t+1)} \leftarrow w_i^{(t)} (1 + \eta \mathbf{b}_i^T \mathbf{f}(\mathbf{x}^{(t)})).$$
- 9: **end for**
- 10: Return $\frac{1}{T} \sum_{t \in [T]} C(\mathbf{x}^{(t)})$ as the solution.

COROLLARY A.2. *The Multiplicative Weight Update algorithm (Algorithm 3) outputs a $(1 + \varepsilon)$ approximate solution to the densest subgraph problem in time $O(m \deg_{\max} \log n / \varepsilon^2)$.*