Abstract

The *Densest-Subgraph* problem has various applications in the fields of data mining, telecommunications and social networks, biology, and financial sciences. This problem, in its simplest form, aims to find a subgraph of a graph with the highest possible density. The concept of graph density can be defined based on multiple relationships depending on the application, but in its simplest case, the density of a subset S of a graph is defined as $\frac{|E[S]|}{|S|}$. This case of the problem has received significant attention in recent years, and several exact and approximation algorithms have been proposed for it.

In this report, we first provide a detailed description of the most well-known algorithms for this case of the problem as well as some other variations. We then practically analyze these algorithms from different perspectives using a comprehensive dataset consisting of 77 random and non-random test cases (which are famous graphs in the real world). The results obtained by us confirm previous findings from a theoretical standpoint, including the approximation ratio of the given approximation algorithms and their convergence. Additionally, we present new observations that can be further investigated in future research.

1 Introduction

The problem of finding the densest subgraph is one of the fundamental problems with various applications in the real world, including telecommunications and data mining. In this problem, the input is a simple graph $G = (V, E)$ and the goal is to find a subset $S \subseteq V$ such that $d(S)$ is maximized. $d(S)$, which represents the induced subgraph density on the vertices S, can have different definitions depending on the literature. The simplest and most common definition for $d(S)$ is the average degree of vertices in the induced subgraph on S, or equivalently $\frac{|E[S]|}{|S|}$. However, besides this standard definition, there are other definitions such as directed densest subgraph [18] or densest subgraph in terms of cluster count [19]. A generalization of the simplest case is to set $d(S) = \frac{f(S)}{|S|}$, where $f(S)$ is a supermodular function with some properties that are discussed in detail in the report. We know that $|E[S]|$ is a supermodular function and therefore, the presented case is a special case of this general problem. An interesting point about the simple case of the problem and its generalization for supermodular functions is that these problems belong to complexity class $\mathcal{P}$. However, since these algorithms are used on large-scale graphs, finding approximate algorithms with faster execution time has become particularly important in recent years.

1.1 Previous Works

For the simple case of this problem, the first algorithm was presented by Picard and Queyranne in 1982 based on flow computations [20]. After 2 years, Goldberg improved their algorithm in terms of time complexity [13], which is currently the fastest exact algorithm proposed for this problem. For 15 years, no serious study was conducted on this problem until Charikar obtained two important results in 2000 [7]:

- Introducing a linear programming formulation that solves the problem exactly and presenting an algorithm based on this formulation.
- Presenting a 2-approximation algorithm called "Greedy Peeling" that has much faster execution time compared to previous exact algorithms.

In 2020, Boob et al. proposed an iterative version of the "Greedy Peeling" algorithm called "Greedy++" or "Iterative Greedy Peeling" that converges to the optimal solution of the problem [4]. However, they were unable to theoretically prove their results and only made conjectures about the correctness of their algorithm through emprical results. In 2022, Chekuri et al. proved Boob et al.'s conjecture. They showed that not only for the simple version of the problem but also for its generalized version on supermodular functions, executing this algorithm converges towards the optimal solution.

During this period, different individuals worked on different versions of this problem obtained by adding various constraints to it. In section 4, we have examined these versions more precisely and

tried to provide a comprehensive summary of the latest results obtained in these problems. It is worth noting that some of these problems, unlike their simple and generalized versions, are not in the complexity class $\mathcal{P}$ and their study is of special interest.

1.2 Our Works

Our most important work and the core of the project is implementing the algorithms and studying this problem from various perspectives such as execution time, approximation ratio, and convergence of algorithms. In section 5, we have discussed and examined these aspects in detail. However, before that, in order to have a complete and comprehensive understanding of the literature on this problem and to understand what works have been done on this problem and how we can have an impact on it, we tried to comprehensively present different algorithms for this problem and prove the previously obtained results accurately. We have also proven some omitted proofs that were removed for brevity in other articles and provided simpler explanations with the help of class materials.

1.2.1 Implementation of Algorithms and Computational Study

Initially, we implemented Goldberg's algorithm, Charikar LP algorithm, Greedy Peeling algorithm, and Iterative Greedy Peeling algorithm. In implementing these algorithms, we tried to optimize them in terms of execution time and algorithm readability so a significant amount of time was spent thinking about how to implement these algorithms using optimized data structures. You can see the codes (along with results and tests) on our project's GitHub page.

Furthermore, to analyze and study the algorithms practically by testing different questions that arise regarding the algorithms, we designed a comprehensive dataset. In designing test cases, we selected some tests related to real-life topics where these algorithms are applicable. For example, we used graphs from maps of some cities like California, Texas, and, Pennsylvania; graphs from some social networks like Facebook and Twitter; as well as other famous graphs. To select these graphs, we utilized various sources of datasets including Stanford's dataset.

Additionally, to add random tests, we have implemented several different generators to create sparse and dense graphs, as well as a generator for creating special graphs that behave badly with the Greedy Peeling algorithm. As a result, we obtained a dataset consisting of 77 different test cases of graphs.

Furthermore, various topics including the analysis of algorithm outputs, approximation ratios, runtime analysis of algorithms, examining the approximation ratio of the Greedy Peeling algorithm based on the graph's density and, the convergence of the Iterative Greedy Peeling algorithm have been tested and studied. The results obtained have been presented in the form of charts and tables.

In section 5, you can see in more detail the work we have done in terms of implementation and computational study, along with the results we have obtained and some issues that can be addressed in the future.

2 Previous Problem Algorithms

Before we get to the main section of the report, which is the analysis section 5, it is better to become familiar with the previous algorithms introduced for the DSP problem and understand the reasons behind them. This will help us better understand the classic ideas that have been proposed for this problem and gain more intuition about the new algorithms for this problem.

As mentioned earlier, the densest subgraph problem is in the complexity class $\mathcal{P}$, and there are polynomial-time algorithms for it. However, since the densest subgraph problem is practically very useful and exact algorithms for it have long execution times, extensive research has also been done on approximation algorithms for this problem. Therefore, in this section, we divide our study into two parts and investigate both exact and approximation algorithms for this problem.

2.1 Exact Algorithms

The first exact algorithm proposed for this problem was by Picard and Queyranne in [20], which solved this problem using a series of successive maximum flow computations. After a few years, Goldberg in [13] improved the execution time of this algorithm and introduced the Goldberg algorithm, which is the most well-known exact algorithm for this problem, and its description is provided in this section.

Later, Charikar in [7] was able to formulate the densest subgraph problem precisely as a linear program, which has great importance in other algorithms proposed for the densest subgraph problem or its generalized versions, including the Iterative Greedy Peeling algorithm, which is one of the main topics of our report.

2.1.1 Goldberg Algorithm

First, let's provide some intuition about what the maximum flow has to do with our problem. Suppose, instead of looking for the densest subgraph, we want to see if there is a subgraph with density greater than or equal to λ:

$$\exists S \subseteq V \,:\, \frac{|E[S]|}{|S|} \geq \lambda \iff \exists S \subseteq V \,:\, \lambda|S| - |E[S]| \leq 0 \iff \min_{S \subseteq V}(\lambda|S| - |E[S]|) \leq 0$$

If we can do this, we can hope to find the optimal answer with a binary search-like operation. With a little care, it can be seen that the above relations bear a great resemblance to designing a separation oracle for the linear program of the minimum spanning tree problem based on the Subtour Elimination method and, in fact, that's what it is. The Goldberg algorithm initially maintains an upper and lower bound on the maximum density and constructs a new graph G' from G. Then it finds the maximum flow on it to answer the above question and finally, with binary search, finds the optimal density.

Let's set $\lambda_0 = 0$ and $\lambda_1 = \frac{m}{2}$ initially, where m is the number of edges in the graph, because if a subgraph wants to have edges, it must have at least two vertices. Now set $\beta = \frac{\lambda_0 + \lambda_1}{2}$. We construct a graph $G'(\beta)$ as follows: add two vertices s and t to G. Then, for each $v \in V$, we add an edge with weight $\frac{\deg_G(v)}{2}$, from s to v and for each $uv = e \in E$, we add an edge with weight $\frac{1}{2}$ from u to v and from v to u. Finally, for each $v \in V$, we add an edge with weight β from v to t.

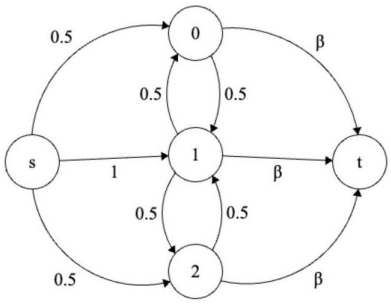

Figure 1: An example of graph G' when $G = P_3$

Now we examine the weight of the s,t-cuts S.

Lemma 1. *For any s,t-cut S where $s \in S$, let $A = S - \{s\}$, we have:*

$$c(\delta(S)) = \beta|A| + m - |E[A]|$$

Proof. The proof of this fact is straightforward, and it suffices to write down all three sets of edges that appear in the cut S.

$$c(\delta(S)) = \sum_{v \in V - A} \frac{deg_G(v)}{2} + \sum_{e \in \delta(A, V-A)} \frac{1}{2} + \sum_{v \in A} \beta$$

Now, in the term $\sum_{v \in V-A} deg_G(v)$, the edges of $\delta(A, V-A)$ appear once, and the edges of $E[V-A]$ appear twice. Therefore, we can continue the calculations as follows.

$$c(\delta(S)) = |E[V-A]| + \frac{E[A, V-A]}{2} + \frac{E[A, V-A]}{2} + \beta|A| = \beta|A| + m - |E[A]|$$

□

Now, according to the lemma above, if we find a minimum s,t-cut in the newly constructed graph and obtain $A = S - \{s\}$, we have minimized $\beta|A| - |E[A]|$. Now, according to our previous calculations, if this value is negative, it means the density of the densest subgraph is greater than β, and we can replace λ_0 with β. Otherwise, we replace λ_1 with β.

Lemma 2. *If $S_1, S_2 \subseteq V$ are two subgraphs with different densities, then $|d(S_1) - d(S_2)| \geq \frac{1}{n(n-1)}$.*

Proof.

$$|d(S_1) - d(S_2)| = \left| \frac{|E[S_1]|}{|S_1|} - \frac{|E[S_2]|}{|S_2|} \right| = \left| \frac{|E[S_1]||S_2| - |E[S_2]||S_1|}{|S_1||S_2|} \right|$$

Now, let's consider two cases:
Case 1: $|S_1| = |S_2| = k$
In this case, we can simplify the expression to $|\frac{|E[S_1]| - |E[S_2]|}{k}|$. Since the numerator is positive, and the denominator is less than n, we have:

$$|\frac{|E[S_1]| - |E[S_2]|}{k}| \geq \frac{1}{n} \geq \frac{1}{n(n-1)}$$

Case 2: $|S_1| \neq |S_2|$
In this case, we have an upper bound of $n(n-1)$ for the denominator, and the numerator is positive. So, we can definitely say that:

$$|\frac{|E[S_1]||S_2| - |E[S_2]||S_1|}{|S_1||S_2|}| \geq \frac{1}{n(n-1)}$$

In both cases, the lemma is satisfied. □

Now, based on the two lemmas above, it is sufficient to continue the binary search until the gap between the lower and upper bounds we have maintained becomes less than $\frac{1}{n(n-1)}$. At that moment, we know that by running the maximum flow algorithm once more with constant λ_0, the minimum cut we obtain is, in fact, the densest subgraph.

Algorithm 1 Pseudo-code for the Goldberg Algorithm

Input: A simple graph $G = (V, E)$.
Output: $S \subseteq V$ such that $G[S]$ has the highest density.
(1) $\lambda_0 \leftarrow 0$, $\lambda_1 \leftarrow \frac{m}{2}$
(2) While $\lambda_1 - \lambda_0 \geq \frac{1}{n(n-1)}$:
(3) $\quad \beta \leftarrow \frac{\lambda_1 + \lambda_0}{2}$
(4) $\quad$ Find the minimum s,t-cut S in the graph $G'(\beta)$.
(5) $\quad A \leftarrow S - \{s\}$
(6) $\quad$ If A is empty:
(7) $\quad\quad \lambda_1 \leftarrow \beta$
(8) $\quad$ Else:
(9) $\quad\quad \lambda_0 \leftarrow \beta$
(10) Find the minimum s,t-cut S in the graph $G'(\lambda_0)$.
(11) Return $S - \{s\}$.

The only remaining issue is the runtime of this algorithm.

Theorem 1. *The algorithm 1 finishes in $\mathcal{O}(T_{flow} \log n)$ time. (For example, using the Edmonds-Karp algorithm, the runtime of the Goldberg algorithm becomes $\mathcal{O}(m^2 n \log n)$).*

Proof. Initially, we have a gap of $\frac{m}{2}$ between the two bounds, and in each iteration of the loop, this gap is halved. Therefore, in $\mathcal{O}(\log(mn(n-1))) = \mathcal{O}(\log(n^4)) = \mathcal{O}(\log n)$ iterations of the loop, the algorithm terminates. The runtime of each loop iteration is T_{flow}, so it's clear that the overall runtime of the algorithm is $\mathcal{O}(T_{flow} \log n)$. □

2.1.2 Charikar's Linear Program

Now we introduce the algorithm based on Charikar's linear program. He showed in [7] that the optimal solution to this linear program is always equal to the density of the densest subgraph.

The initial idea for designing a linear program for this problem is to introduce a variable y_v for each vertex, which represents whether the vertex is included in the optimal subgraph. Similarly, we introduce a variable x_e for each edge. The challenge here is to design the objective function. Essentially, having $|S|$ in the denominator prevents us from directly achieving this idea.

Charikar tackled this problem by introducing a form of load balancing constraints. He aimed to set up the constraints in such a way that when an edge is included in the subgraph, x_e becomes $\frac{1}{|S|}$. Clearly, if we can achieve this, our objective function becomes a very simple one: $\sum_{e \in E} x_e$. The linear program presented by Charikar is as follows:

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_{e \in E} x_e \\ \text{subject to} \quad & x_{u,v} \leq y_u \; \forall uv \in E \\ & x_{u,v} \leq y_v \; \forall uv \in E \\ & \textstyle\sum_{v \in V} y_v = 1 \\ & x, y \geq 0 \end{aligned} \tag{1}$$

Now, we examine the properties of this linear program.

Lemma 3. *For any subgraph S with density d, there exists a corresponding solution in the linear program 1 such that its objective function value is d.*

Proof. It is clear that by setting $y_v = \frac{1}{|S|}$ for vertices $v \in S$ and $x_e = \frac{1}{|S|}$ for edges $e \in E$, we satisfy the requirements of the theorem. □

Now, we present the algorithm. The idea behind this algorithm bears a resemblance to the rounding techniques we saw in class for the prize-collecting Steiner tree problem.

Algorithm 2 Pseudo-code for the Charikar Algorithm

Input: A simple graph $G = (V, E)$.
Output: $S \subseteq V$ such that $G[S]$ has the highest density.
(1) Solve the linear program 1 and find the optimal solution x^*, y^*.
(2) For each $0 \leq r \leq y^*_{\max}$, set $S_r = \{v : y_v \geq r\}$.
(3) Return the S_r with the highest density.

Clearly, in the second line of the algorithm, it is sufficient to find S_r for finitely many possible values (the same values as y_v), and for the sake of easier analysis, we wrote the algorithm this way.

Now, for the analysis of the algorithm, we will use a strategy similar to the strategies we used in class for problems like multi-cut or prize-collecting Steiner tree.

Theorem 2. *The algorithm described in 2 is an exact algorithm for the densest subgraph problem.*

Proof. Suppose that for every r, we have $\frac{|E[S_r]|}{|S_r|} < OPT$. Then we can say $|E[S_r]| < |S_r| \cdot OPT$. Now, let's calculate both sides of the following inequality:

$$\int_0^{y^*_{\max}} |E[S_r]|\, dr < OPT \int_0^{y^*_{\max}} |S_r|\, dr$$

Now, let's compute both sides of the inequality:

$$\int_0^{y^*_{\max}} |E[S_r]|\, dr = \sum_{e \in E} \int_0^{y^*_{\max}} \mathbb{1}(e \in S_r)\, dr = \sum_{e \in E} \min(y^*_u, y^*_v) = \sum_{e \in E} x_e = OPT$$

Likewise, for the other side of the inequality:

$$\int_0^{y^*_{\max}} |S_r|\, dr = \sum_{v \in V} \int_0^{y^*_{\max}} \mathbb{1}(v \in S_r)\, dr = \sum_{v \in V} y_v = 1$$

Now, by combining these equations, we arrive at the contradiction that $OPT < OPT$. Therefore, there must be an r for which S_r has a density greater than or equal to OPT. Since, according to Lemma 3, $OPT \geq d(S_r)$, we conclude that $d(S_r) = OPT$, which proves our claim. □

2.2 Approximation Algorithms

As mentioned earlier, despite having a polynomial-time algorithm for this problem, extensive research has also been conducted on its approximation algorithms. The first approximation algorithm for this problem was introduced by Kortsarz and Peleg in [17]. They showed that the maximum w-core in a graph is a 2-approximation for our problem.

Definition 1. *A maximally connected subgraph with minimum degree at least w is called a w-core.*

You might wonder about the connection between this w-core concept and the densest subgraph problem. In fact, with a little observation, it's clear that the density of a subgraph is nothing but the average degree of its vertices, which is why we can search for subgraphs with high degrees to find subgraphs with high density. Hence, there is a correlation between these two problems.

Continuing, Charikar, in [7], inspired by the same idea, designed a greedy algorithm with a linear running time for this problem, which is the most well-known approximation algorithm for this problem. We have analyzed this algorithm on various tests in section 5 and demonstrated that its approximation ratio is significantly better than 2 in practice.

Following this algorithm, Boob et al. in [4] proposed a new algorithm based on successive executions of the greedy algorithm and conjectured that this algorithm converges to the optimal solution. Chekuri, Quanrud, and Torres in [8] proved this conjecture, which we discussed in 3.

2.2.1 Greedy Peeling Algorithm

Suppose you want to greedily increase the density of a subgraph. If you remove a vertex from this subgraph, you decrease the numerator of the density fraction by the degree of that vertex and the denominator by one. Therefore, the best choice for removing a vertex from the subgraph greedily is a vertex with the minimum degree. This general intuition is the basis of the algorithm we want to introduce here.

Algorithm 3 Pseudo-code for the Greedy Peeling Algorithm

Input: A simple graph $G = (V, E)$.
Output: $S \subseteq V$ such that $G[S]$ has the highest density.
(1) $S_n \leftarrow V$
(2) For i from n down to 1, do the following:
(3) Find a vertex with minimum degree in $G[S_i]$ and name it v.
(4) $S_{i-1} \leftarrow S_i - \{v\}$
(5) $j \leftarrow \text{argmax}_i\ density(S_i)$
(6) Return S_j.

This algorithm also has a connection with the definition of w-core. In fact, we have the following theorem.

Theorem 3. *The Greedy Peeling algorithm finds all w-cores of the graph.*

Since we present a proof for a slightly different version of this algorithm in 3, we omit the correctness proof and the approximation ratio here.

Based on the intuitions we provided for this algorithm, it appears that the algorithm performs poorly on cases where the densest subgraph has vertices with very low degrees. In reality, this is the case, and here we present some examples inspired by [14] to demonstrate that the approximation ratio of 2 for this algorithm is exact.

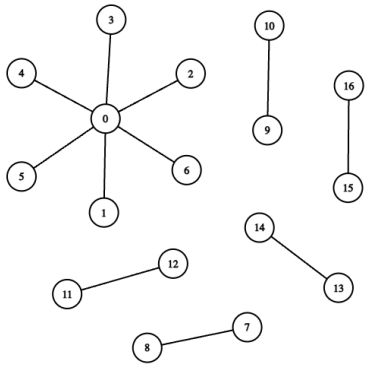

Figure 2: An example of a worst-case scenario for the algorithm when $p = 6$ and $t = 5$.

Theorem 4. *The approximation ratio of* 2 *for the Greedy Peeling algorithm is exact.*

Proof. We construct an example as follows. Initially, place $2t$ different vertices, and create a perfect matching between them. In fact, create t completely isolated edges. Now, add $p+1$ additional vertices and connect one of these vertices to the other p vertices (a star with $p + 1$ vertices). Now, it can be easily seen that the densest subgraph of this graph is the same star with a density of $\frac{p}{p+1}$. Now, assume that the algorithm always removes a vertex from the star in case of multiple choices. In this case, the algorithm removes one vertex from the star in each step until the entire star is removed, leaving only t isolated edges. Then, it removes vertices among them. It can be seen that in this case, the algorithm reduces the density with each vertex removed, so the maximum density that the algorithm sees and outputs is the same as the initial density of the graph. Now, our graph has $2t + p + 1$ vertices and $t + p$ edges, so the output of the algorithm is $\frac{t+p}{2t+p+1}$.

$$Algorithms\ Answer : \frac{t+p}{2t+p+1}\ ,\ Optimal\ Answer : \frac{p}{p+1}\ \rightarrow\ Aprroximate\ Ratio = \frac{\frac{t+p}{2t+p+1}}{\frac{p}{p+1}}$$

Now, if p goes to infinity and $t >> p$, the above expression approaches $\frac{1}{2}$. □

In the case of the execution time of this algorithm and its implementation details, we discussed it in detail in section 5.

3 The `Iterative Greedy Peeling` Algorithm

In this section, we intend to introduce the `Iterative Greedy Peeling` algorithm, also known as `Greedy++`. The initial idea of this algorithm was proposed by Boob et al. in [4], but they couldn't say much about the performance of the algorithm and its approximation ratio. Recently, Chekuri,

Quanrud, and Torres in [8] managed to prove that this algorithm converges to the optimal solution for the densest subgraph problem. The initial idea of this algorithm is inspired by the Multiplicative Weight Update method, and it has a natural intuition. Since the Greedy Peeling method works well and usually performs much better than its theoretical approximation ratio, the idea is to run the Greedy Peeling algorithm multiple times and, after each run, calculate a measure of the value of each vertex in the graph. In the subsequent runs of the Greedy Peeling algorithm, instead of removing vertices with the lowest degree, we remove vertices with the lowest value according to our measure. In this section, we first discuss the Greedy Peeling algorithm and its properties. Then we move on to the Greedy++ algorithm and try to explain why this algorithm works well and can provide a $(1-\epsilon)$ approximation for any $\epsilon > 0$.

3.1 The Greedy Peeling Algorithm for the supermodular Densest Subgraph Problem

As the name of this section suggests, we are going to focus on a more general version of the densest subgraph problem called the supermodular densest subgraph problem. Instead of finding a subgraph $S \subseteq V$ that maximizes $\frac{|E[S]|}{|S|}$, we aim to find a subgraph that maximizes $\frac{f(S)}{|S|}$, where f is a non-negative supermodular function of our choice. We assume that $f(\emptyset) = 0$, which is not necessarily $|E[S]|$ (we saw in class that the function $|E[S]|$ is supermodular). To explain further, whenever we refer to the function f, we assume it satisfies these conditions.
To explain the details, we first introduce several definitions.

Definition 2. *For a function $f : 2^V \to \mathbb{R}_{\geq 0}$, a subset $S \subseteq V$, and $v \in V - S$, we define:*

$$f(v|S) = f(S) - f(S - \{v\})$$

Definition 3. *An equivalent definition to the one we had in class for a supermodular function f is that for every $A \subset B \subset V$ and $x \in V - B$, we have:*

$$f(x|A) \leq f(x|B)$$

Definition 4. *For any function f, we define the parameter c_f as follows:*

$$c_f = \max_{S \subseteq V} \frac{\sum_{v \in s} f(v|S-v)}{f(S)}$$

Theorem 5. *In the case where $f(S) = |E[S]|$, the value of c_f is 2.*

Proof. It can be easily seen that if $f(S) = |E[S]|$, then $f(v|S-v)$ becomes the degree of vertex v in $G[S]$. Since the sum of degrees of vertices in a graph is twice the number of edges, in this case, $c_f = 2$. □

Lemma 4. *For any function f and subset $S \subseteq V$, we have:*

$$\sum_{v \in s} f(v|S-v) \geq f(S)$$

Proof.

$$\sum_{v \in s} f(v|S-v) = \sum_{v \in s} (f(S) - f(S-v)) = |S|f(S) - \sum_{v \in s} f(S-v)$$

Now, we use the property $f(A \cup B) + f(A \cap B) \geq f(A) + f(B)$ for supermodular functions. Suppose $S = \{v_1, \dots v_{|S|}\}$.

$$f(S-v_1) + \cdots + f(S-v_{|S|}) \leq f(S) + f(S - \{v_1, v_2\} + f(S-v_3) + \cdots + f(S-v_{|S|})$$

$$\leq \cdots \leq (|S|-1)f(S) + f(\emptyset) = (|S|-1)f(S)$$

By combining the above calculations, we arrive at:

$$\sum_{v \in s} f(v|S-v) \geq |S|f(S) - (|S|-1)f(S) = f(S)$$

Thus, the lemma is proven. □

Now, we can continue the analysis of the algorithm. You might wonder how we remove nodes in the Greedy Peeling algorithm now that f is an arbitrary function. It no longer makes sense to remove nodes based on their degrees. Therefore, in the case of the supermodular densest subgraph problem, the Greedy Peeling algorithm removes a node v from the current set S_i of nodes at each step, where $f(v|S_i)$ is minimized, and $S_{i-1} = S_i - v$.

Lemma 5. *If S^* is an optimal solution to the problem, and $d^* = \frac{f(S^*)}{|S^*|}$, then for any $v \in S^*$, we have $f(v|S^*) \geq d^*$.*

Proof. If S^* is a singleton, the claim is obvious because $f(v|S^*) = f(S^*) - f(\emptyset) = f(S^*) = d^*$. Now, suppose S^* has more than one element. By contradiction, if the theorem does not hold, there exists $v \in S^*$ such that $f(v|S^*) < d^*$. Consider the density of $S^* - v$:

$$\frac{f(S^*-v)}{|S^*-v|} > \frac{f(S^*)-d^*}{|S^*|-1} = d^*$$

However, this contradicts the optimality of S^*. Thus, the lemma is proven. □

Theorem 6. *The Greedy Peeling algorithm, for finding the densest supermodular subgraph, is a $\frac{1}{c_f}$-approximation algorithm.*

Proof. Let S^* be the optimal solution to the problem, and let $d^* = \frac{f(S^*)}{|S^*|}$. Now, suppose S_i is a minimal set in the execution of the Greedy Peeling algorithm such that $S^* \subseteq S_i$. In this case, in step i of the algorithm, a node $v_j \in S^*$ must have been removed from S_i, meaning $v_j = \text{argmin}_{v_k \in S_i} f(v_k|S_i)$. Now, let's calculate the density of S_i:

$$\frac{f(S_i)}{|S_i|} = \frac{f(S_i)}{|S_i|} = \frac{\sum_{v \in s} f(v|S_i - v)}{|S_i|} \times \frac{f(S_i)}{\sum_{v \in s} f(v|S_i - v)}$$

Using the fact that $v_j = \text{argmin}_{v_k \in S_i} f(v_k|S_i)$ and the definition of the parameter c_f, we have:

$$\frac{f(S_i)}{|S_i|} \geq \frac{1}{c_f} \times \frac{|S_i| f(v_k|S_i)}{|S_i|} = \frac{f(v_k|S_i)}{c_f}$$

Since $S^* \subset S_i$, by Definition 3, we have $f(v_k|S_i) \geq f(v_k|S^*)$. Furthermore, from Lemma 5, we know that $f(v_k|S^*) \geq d^*$. Therefore, we can continue the calculations as follows:

$$\frac{f(S_i)}{|S_i|} \geq \frac{f(v_k|S_i)}{c_f} \geq \frac{f(v_k|S^*)}{c_f} \geq \frac{d^*}{c_f}$$

Thus, S_i itself is a $\frac{1}{c_f}$-approximation, and since the algorithm outputs the best among all S_k, the algorithm is also $\frac{1}{c_f}$-approximate. □

One result of Theorem 6 is that the Greedy Peeling algorithm is a 2-approximation algorithm for the standard Densest Subgraph problem.
Now that we have stated Theorem 6, the only remaining challenge is to find the exact value or an upper bound for c_f. This depends on the specific function f and its properties, such as the argument we made for the function $f(S) = |E[S]|$.

3.2 Iterative Greedy Peeling Algorithm

In accordance with the discussions above, this algorithm should utilize the previous runs of the Greedy Peeling algorithm and assign weights to nodes based on their importance in solving the problem.

A natural choice for assigning weights to nodes is to maintain a value l_v for each node, initially set to zero. When a node v is removed from a set S_i in the algorithm, the value of l_v is updated as follows: $l_v = l_v + \deg_{G[S_i]}(v)$. This way, the Iterative Greedy Peeling algorithm uses the degree of a node in previous executions of the Greedy Peeling algorithm as a guide to its impact on finding the densest subgraph. The pseudocode for the algorithm is as follows:

Algorithm 4 Pseudocode for Iterative Greedy Peeling Algorithm

Input: A simple graph $G = (V, E)$ and a positive integer T.
Output: $S \subseteq V$ such that $G[S]$ has the highest density.
(1) For each i from 1 to T:
(2) $\quad S_{i,n} \leftarrow V$
(3) $\quad$ For j from n to 1:
(4) $\quad\quad$ Find the node v for which $l_v + \deg_{G[S_{i,j}]}(v)$ is minimized.
(5) $\quad\quad S_{i,j-1} \leftarrow S_{i,j} - \{v\}$
(6) $\quad\quad l_v \leftarrow l_v + \deg_{G[S_{i,j}]}(v)$
(7) $i, j \leftarrow \text{argmax}_{i,j}\text{density}(S_{i,j})$
(8) Return $S_{i,j}$ as the output.

This algorithm was first introduced in [4], but an approximation factor was not provided until recently when Chekuri, Quanrud, and Torres proved the following theorem in [8]:

Theorem 7. *For a graph G with maximum degree Δ and maximum density d^*, the Iterative Greedy Peeling algorithm (Algorithm 4) provides a $(1-\epsilon)$-approximation for the Densest Subgraph problem if $T \geq \mathcal{O}(\frac{\Delta \ln(n)}{d^* \epsilon})$.*

Their result is actually stronger than the theorem above and is given for the Densest supermodular Subgraph.

4 Different Versions of the Densest Subgraph Problem

Different versions of the Densest Subgraph problem can generally be categorized into two groups:

1. Problems with constraints on the size of the subgraph.
2. Problems with constraints on the connectivity of the subgraph.

In this section, we introduce these two categories of problems and discuss their computational complexity along with some known algorithms.

4.1 Problems with Constraints on Subgraph Size

Among these problems, we can mention the Densest k-Subgraph (**DKSG**), Densest At Least k-Subgraph (**DALKSG**), and Densest At Most k-Subgraph (**DAMKSG**) problems. These problems seek a subgraph of a specific size k (exact, minimum, or maximum) with the maximum density. We will investigate these three problems.

4.1.1 Densest k-Subgraph Problem

As mentioned earlier, the Densest k-Subgraph problem aims to find the densest subgraph of exactly k vertices, which is defined as $\max_{S \subseteq V, |S|=k} \frac{|E[S]|}{|S|}$. It is evident that the denominator of this fraction is always equal to k and does not affect the optimization result. Therefore, our goal is to find a subgraph of k vertices with the maximum number of edges. It can be shown that $Clique \leq_P DKSG$, and since the Clique problem is NP-hard, the Densest k-Subgraph problem is also NP-hard.

Theorem 8. *Clique* $\leq_P$ *DKSG*

Proof. It is evident that if there exists a clique of size k in the graph, then the densest subgraph of size k in the graph will be this clique, and its density will be $\frac{k-1}{2}$. So, for each $1 < r \leq n$, we can find the densest subgraph of size r. Now, the largest r for which we found an r-node subgraph with density $\frac{r-1}{2}$ will be the size of the largest clique in the graph, and in this way, the reduction is completed. □

Several polynomial-time approximation algorithms have been proposed for this problem. For example, the $\mathcal{O}(n^{\frac{1}{3}})$-approximation algorithm by Feige, Kortsarz, and Peleg [11] combines different methods to output the best subgraph among multiple candidates. One of these methods selects a subgraph randomly, which performs well for large values of k, while another method constructs a subgraph greedily.

Additionally, research has been conducted on the parameterized complexity of this problem.

Theorem 9. *The Densest k-Subgraph problem is W[1]-hard, even in regular graphs.*

This theorem, provided by Cai L in [6], establishes the hardness of the Densest k-Subgraph problem in the context of parameterized complexity.
Also, this problem has been investigated in cases where additional constraints are placed on the value of k or the number of edges in the graph.

Theorem 10. *In the case of* $k = \frac{n}{2}$*, the trivial algorithm achieves a* $\frac{1}{4}$*-approximation.*

Proof. If we uniformly select a subset S of $\frac{n}{2}$ vertices, the probability of selecting each vertex is $\frac{1}{2}$. Therefore, we have:

$$\mathbb{P}[v \in S] = \frac{1}{2} \rightarrow \mathbb{P}[uv = e \in S] = \mathbb{P}[u \in S]P[v \in S] = \frac{1}{4} \rightarrow \mathbb{E}[|E[S]|] = \frac{m}{4}$$

Now, using random sampling techniques similar to those seen in class, we can find a subset S where the number of edges $|E[S]|$ is greater than $\frac{m}{4}$. Since the maximum possible number of edges a subgraph can have is m, this provides a $\frac{1}{4}$-approximation for our problem. □

However, this is not the best possible result, and Ye and Zhang in [22], using a combination of semi-definite programming and rounding algorithms, achieved a 0.586-approximation algorithm for this problem in this specific case, improving the previous approximation algorithms. Additionally, in this paper, they investigate the relationship between the problem of DKSG and the problem of Min-Bisection, which we also studied in class.

4.1.2 Densest At Least k Subgraph Problem

This problem is computationally hard, falling into the class of NP-hard problems. However, proving this fact is not as straightforward as the other problems mentioned in this section. Khuller and Saha demonstrated in [16] by reducing the DKSG problem to this problem.

Theorem 11. *The Densest At Least k Subgraph (DALKS) problem is NP-hard, and DKSG* $\leq_p$ *DALKS.*

Proof. Let's assume we are given an n-vertex graph G, a positive integer k, and an integer d, and we want to determine if there exists a k-vertex subgraph in G with a density greater than d.
Now, add a complete graph with n^2 vertices to graph G to obtain graph G'. Essentially, G' is the union of G and a K_{n^2}. Now, find the densest subgraph of at least $n^2 + k$ vertices in this graph G', and denote it as S. We claim that this answer has some favorable properties.
Property 1: All vertices of K_{n^2} are included in S.

Proof. Suppose not. The set of vertices from this complete graph that are not in S is denoted as T. Let's assume that $|S| = r$, and we denote the density of S as $d(S)$.
Now, let's calculate how many edges are included in the subgraph when we add T to S. Each vertex in T is connected to n^2 vertices, but we have counted the edges within T twice, so the number of edges

added is $|T|n^2 - \frac{|T|(|T|-1)}{2} = |E_T|$.
Now, we demonstrate the following inequality.

$$|E_T| = |T|n^2 - \frac{|T|(|T|-1)}{2} = |T|(n^2 - \frac{|T|-1}{2}) \geq |T|(\frac{n^2-1}{2}) \geq |T|d(S)$$

The reason for the last inequality is that when T is non-empty, both parts $S - K_{n^2}$ and $K_{n^2} - T$ in S have densities less than $\frac{n^2-1}{2}$.
So, for the density of the new solution, we have:

$$d(S \cup T) = \frac{d(S)r + |E_T|}{r + |T|} > \frac{d(S)(r + |T|)}{r + |T|} = d(S)$$

And the first property is proven. □

Property 2: Exactly k vertices from G are included in S, i.e., $|S \cap G| = k$.

Proof. The reason for this fact is that $d(S)$ is decreasing concerning $|S \cap G$, and it prefers to take the minimum possible number of vertices from G. However, since it must take at least $n^2 + k$ vertices, it must take exactly k vertices from G.
Suppose that if exactly l vertices are taken from G, the optimal solution is S_l. Now, the reason for the decreasing property is as follows:

$$d(S_l) \leq \frac{\binom{n^2}{2} + \binom{l}{2}}{n^2 + l} \leq \frac{\binom{n^2}{2} + 1}{n^2 + l - 1} \leq d(S_{l-1})$$

And this property is also proven. □

Now, by combining the two properties above, it becomes clear that the k vertices from G included in S precisely form the densest k-vertex subgraph of G, and the density of this subgraph is equal to:

$$\frac{d(S)(n^2 + k) - \binom{n^2}{2}}{k}$$

So, if $d(S) > \frac{dk + \binom{n^2}{2}}{n^2 + k}$, then the densest k-vertex subgraph of G has a density greater than d. □

In terms of algorithms for this problem, Andersen and Chellapilla in [1] showed that the Greedy Peeling algorithm, which we introduced for the DSP problem, provides a 3-approximation in linear time. The proof for this approximation factor is similar to the proof of the 2-approximation for the Greedy Peeling algorithm and relies on the properties related to the graph's w-core, which we introduced in the definition 1. We also saw that a significant part of the idea for solving the densest subgraph problem is derived from the concept of Core Decomposition.

However, this is not the best possible approximation for this problem. Khuller and Saha in [16] designed a 2-approximation algorithm for this problem. Initially, for all possible values of $l \geq k$, they solve the following linear program, which is inspired by Charikar's linear program [7] that precisely models the DSP problem:

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in E} x_e \\
\text{subject to} \quad & x_{u,v} \leq y_u \ \forall uv \in E \\
& x_{u,v} \leq y_v \ \forall uv \in E \\
& \sum_{v \in V} y_v = 1 \\
& y_v \leq \tfrac{1}{l} \ \forall v \in V \\
& x, y \geq 0
\end{aligned}$$

Since we don't know the optimal solution's size, they attempt to guess the number of vertices in the optimal solution by solving all these linear programs. They name the optimal solution of the linear program corresponding to l as $d^*(l)$. Then, for each optimal solution $d^*(l)$, they obtain a subgraph with at least k vertices and a density of at least $\frac{d^*(l)}{2}$. Finally, among all obtained subgraphs, they output the one with the highest density.

It is evident that if the optimal subgraph of the problem has l^* vertices, by setting $y_i = \frac{1}{l^*}$ and $x_{i,j} = \frac{1}{l^*}$ for all vertices and edges in the optimal subgraph in the linear program corresponding to l^*, the objective function's value of this linear program becomes equal to the density of the optimal solution. Therefore, they achieve a 2-approximation since if the optimal subgraph's density is OPT, then:

$$\forall l \geq k \ : \ d(G_A) \geq d(G_l)\,,\, d(G_{l^*}) \geq \frac{d^*(l^*)}{2} \geq \frac{OPT}{2} \rightarrow d(G_A) \geq \frac{OPT}{2}$$

It may raise the question of why this algorithm does not work for the DKSG problem. The point is that the subgraph G_l they find in this algorithm is not necessarily l-vertex, and it merely has more than k vertices. Therefore, it is not possible to necessarily reach a solution for the DKSG problem by solving the above linear program with $l = k$.

4.1.3 The Problem of Densest At Most k Subgraph

The problem of Densest At Most k Subgraph (DAMKS) is, like the previous problems, in the class NP-hard. With a little attention, we can see that the reduction given in Theorem 8 also works here, because if a graph has a cluster of size k, then that cluster becomes the densest subgraph of size at most k in that graph, and its density is $\frac{k-1}{2}$. So, the reduction we provided in that theorem applies here as well.

Theorem 12. *The problem of DAMKS is an NP-hard problem, and Clique $\leq_p$ DAMKS.*

Regarding the approximation hardness of this problem, there are results showing that this problem is almost as hard to approximate as the DKSG problem, and their differences are in the range of a constant factor. More precisely, we have the following theorem from Khuller and Saha [16]:

Theorem 13. *If we have an α-approximation algorithm for the DAMKS problem, then we will have a 4α-approximation algorithm for the DKSG problem.*

4.2 Problems with Connectivity Constraints on Subgraphs

In a category of problems, we seek the densest subgraph that satisfies a series of connectivity constraints. This version of the densest subgraph problem is essential because one of the problems with the densest subgraph is that even though the found subgraph has high density, it may have a Single Point of Failure, which is a negative event in telecommunications and computer networks.

Depending on the type of connectivity we consider, we can divide these problems into two categories:

1. The problem of finding the densest subgraph that is k-vertex-connected.
2. The problem of finding the densest subgraph that is k-edge-connected.

These two problems were first introduced in [3], where a 4-approximation algorithm was presented. In fact, their proposed algorithm is a $(\frac{4}{\alpha}, \frac{1}{\alpha})$-approximation algorithm, where $2 \geq \alpha \geq 1$. This means that, as we aim for better connectivity, the final solution has a worse approximation factor and vice versa.

5 Analysis

5.1 Setup and Implementation

The implementation of all algorithms, test generators, and the results of algorithm executions along with plots are all available on GitHub at this link: GitHub Repository. All the algorithms mentioned

below are implemented in Python and executed on a computer equipped with an Intel Core i9-11900H @ 2.5GHz processor and 16GB of RAM. The reported times are in milliseconds.

In the following sections, we report the results obtained from the execution of the following algorithms:

- Goldberg's algorithm in Section 1.
- Charikar's algorithm in Section 2.
- Greedy Peeling algorithm in Section 3.
- Iterative Greedy Peeling algorithm in Section 4.

For the implementation of the max-flow and Charikar algorithms, which require solving LP, libraries like max-flow and cvxpy have been used. Additionally, in the Greedy Peeling algorithm, to linearize the execution time of the program, a bidirectional linked list has been used. This way, the algorithm can be implemented in $\mathcal{O}(n+m)$ time.

Furthermore, in the Iterative Greedy Peeling algorithm, the heapq library has been used for adding and removing the minimum element, making the algorithm $\mathcal{O}(T(n+m)\log(n))$ in terms of complexity.

5.2 Generators and Testbed

The tests used for analyzing the algorithms and their results were obtained through two methods:

- Some of the tests are derived from well-known datasets and real-world graphs, which have non-random structures. Many of these tests consist of large-sized graphs. To collect these graphs, multiple sources, including the Stanford dataset, have been used. It's worth noting that since this problem is highly relevant in various applications, studying and analyzing these tests is of great importance. Additionally, some of these tests model the graphs of roads in certain cities or social network graphs in real-world applications.

  A total of 27 test cases of this category have been included in the test suite.

- Several test cases have been created through various generators that we implemented. These tests were generated by the following methods:

  1. 20 test cases consist of sparse random graphs. These graphs were created by randomly adding m unique edges to an n-node graph. To generate these m edges, a one-to-one mapping from a permutation of $\{0, 1, \ldots, \binom{n}{2}\}$ to all possible edges was used.
  2. 20 test cases consist of dense random graphs. These graphs were generated by first applying a random permutation to all edges and then selecting the first m edges as the graph's edges. Similar to the random sparse graphs, the permutation of edges used the same mapping.
  3. 10 test cases were generated for graphs with an approximation factor of 2 (worst-case approximation factor) with various values of t and p. The structure of these graphs can be seen in Section 2.2.1.

In total, 77 different test cases have been used for analyzing the algorithms and their performance.

5.3 Analysis

5.3.1 General Results and Outputs

Results are available in full on GitHub, and here, for analysis, results and outputs for a selected set of tests are shown in Table 5.3.1. To obtain the results, the `goldberg` algorithm was executed for a maximum of 40 seconds. If no answer was received from the algorithm, the algorithm execution was terminated, and the value `NaN` was recorded as the output for that test case. For the `charikar` algorithm, if n and m were both less than or equal to 5000, the algorithm was executed; otherwise, `NaN` was recorded in the output.

testname	|V|	|E|	goldberg	charikar	greedy peeling	greedy++	greedy peeling AR	greedy++ AR
adjnoun	112	425	4.7917	4.7917	4.7778	4.7917	0.9971	1.0
blogcatalog	10321	33983	98.2791	NaN	98.2791	98.2791	1.0	1.0
com-amazon.ungraph	334863	925872	4.8041	NaN	3.7576	4.5909	0.7822	0.9556
com-youtube	1134890	2987624	NaN	NaN	45.5731	45.5865	0.9997	NaN
ego-facebook	4039	88234	77.3465	NaN	77.3465	77.3465	1.0	1.0
ego-twitter	81306	1342296	NaN	NaN	68.4142	69.6185	0.9827	NaN
web-Google	875713	4322051	NaN	NaN	27.1792	27.7872	0.9781	NaN
roadNet-CA	1965206	2766607	NaN	NaN	1.7138	1.7611	0.9732	NaN
roadNet-TX	1379917	1921660	NaN	NaN	1.8462	1.9322	0.9555	NaN
roadNet-PA	1088092	1541898	NaN	NaN	1.6878	1.7059	0.9894	NaN
sparse12	54213	6312	0.9	NaN	0.7	0.8889	0.7778	0.9877
sparse13	9934	4123	0.9910	NaN	0.9048	0.9851	0.9130	0.9940
sparse2	2037	432	0.9167	0.9167	0.7	0.9167	0.7636	1.0
dense10	981	381726	389.1193	NaN	389.1193	389.1193	1.0	1.0
dense15	873	153215	175.5040	NaN	175.5040	175.5040	1.0	1.0
dense13	10234	298984	NaN	NaN	29.2148	29.2148	1.0	NaN
worst-case2	481	250	0.9523	0.9524	0.5198	0.9524	0.5457	1.0
worst-case6	19279	9757	0.9958	NaN	0.5061	0.9958	0.5082	1.0
worst-case1	27	18	0.9091	0.9091	0.6667	0.9091	0.7333	1.0

Table 1: outputs and overall results(AR stands for approximation ratio.)

Additionally, the `greedy++` algorithm is executed on each test with $T = \min(\lfloor \frac{10^7}{m} \rfloor, 30)$.

To examine the approximation ratios, if an answer from the `goldberg` algorithm was available, it was compared with that answer. Otherwise, it was compared with the answer from the `Iterative Greedy Peeling` algorithm, and the approximation ratio for `Iterative Greedy Peeling` was set to `NaN`. Finally, note that the approximation ratios have been calculated in a way that they are always less than or equal to 1.

The first observation that can naturally be made is that the output values of the `goldberg` and `charikar` algorithms are always equal. This is because both algorithms are exact algorithms, and their output is the optimal solution. Additionally, it can be observed that the approximation ratio in both algorithms is always greater than or equal to 0.5, as expected. However, in practice, it can be seen that the approximation ratio is very close to 1 on denser and more random graphs. This fact was mentioned in previous work on these algorithms, and we will further investigate it in the following sections.

Regarding the `Iterative Greedy Peeling` algorithm, it can be seen that even with a small value of T (less than or equal to 30), the approximation ratio is always very close to 1. Although it has been theoretically proven that the algorithm converges to the optimal solution, the upper bound on the number of iterations provided in the proof has a noticeable difference from practical results. This raises the question of whether the provided upper bound on the number of iterations can be improved or not. Further details about the convergence of this algorithm will be discussed in the following sections.

Finally, note that in the `worst-case` test cases, even with small values of t and p, the approximation ratio for `greedy-peeling` differs significantly from other graphs and can easily be made close to 0.5. This observation implies that the presented approximation ratio for `greedy-peeling` is tight.

5.3.2 Execution Time

Results are available in full on GitHub, and here, for analysis, results and outputs for a selected set of tests are shown in Table 5.3.2.

The `charikar` algorithm, due to solving the LP, has a very poor execution time and has only been run on smaller tests. In these tests, it can be observed that the execution time of `charikar`

testname	|V|	|E|	goldberg	charikar	greedy peeling	greedy++
adjnoun	112	425	472.53ms	440.82ms	1.01ms	10.52ms
blogcatalog	10321	33983	37835.16ms	NaN	341.53	26517.91ms
com-amazon.ungraph	334863	925872	39977.00ms	NaN	2988.44ms	38694.52ms
com-youtube	1134890	2987624	NaN	NaN	7118.82ms	42766.46ms
ego-facebook	4039	88234	2809.32ms	NaN	92.89ms	4383.23ms
ego-twitter	81306	1342296	NaN	NaN	1989.47ms	30624.26ms
web-Google	875713	4322051	NaN	NaN	10709.81ms	49318.26ms
roadNet-CA	1965206	2766607	NaN	NaN	7118.94ms	26429.34ms
roadNet-TX	1379917	1921660	NaN	NaN	5521.85ms	50446.94ms
roadNet-PA	1088092	1541898	NaN	NaN	4333.37ms	31495.55ms
sparse12	54213	6312	1594.27ms	NaN	149.51ms	3619.99ms
sparse13	9934	4123	912.01ms	NaN	15.64ms	426.23ms
sparse2	2037	432	547.14ms	4504.56ms	2.00ms	100.49ms
dense10	981	381726	23708.92ms	NaN	272.26ms	26198.75ms
dense15	873	153215	6701.25ms	NaN	104.51ms	8794.20ms
dense13	10234	298984	NaN	NaN	311.09ms	25965.30ms
worst-case2	481	250	502.81ms	526.64ms	<0.01ms	15.54ms
worst-case6	19279	9757	1227.61ms	NaN	37.67ms	1193.81ms
worst-case1	27	18	507.59ms	24.55ms	<0.01ms	1.01ms

Table 2: runtime results (values are in terms of miliseconds.)

differs significantly from other algorithms, especially the approximation algorithms. For example, in the `sparse2` test, the execution time of `charikar` is approximately 4.5 seconds, while the execution time of the `goldberg` algorithm is about half a second, and the execution time of greedy peeling and iterative greedy peeling is approximately 2 and 100 milliseconds, respectively. This demonstrates that other algorithms have a clear advantage over the `charikar` algorithm in terms of execution time.

As expected, the `greedy-peeling` algorithm has a significantly lower execution time compared to `goldberg` and `Iterative Greedy Peeling` due to its linearity and the use of efficient data structures such as doubly-linked lists. Even on very dense graphs with a high number of vertices, it performs optimally in terms of execution time. However, its issue lies in its approximation ratio, as previously mentioned.

Regarding the `goldberg` and `Iterative Greedy Peeling` algorithms, note that the execution time of `goldberg` is $\mathcal{O}(T_{flow}\log(n))$, where T_{flow} is the execution time of the maximum flow algorithm in Python, which, according to [5], is $\mathcal{O}(mn^2|C|\log(n))$, where $|C|$ is the cost of the minimum cut. However, as mentioned in [5], the `goldberg` algorithm is much faster in practice. Furthermore, the execution time of the `Iterative Greedy Peeling` algorithm is $\mathcal{O}(T(n+m)\log(n))$, and as a result, it often performs faster. In some tests like `ego-facebook`, it can be seen that the `goldberg` algorithm is faster, mainly due to the small size of the graph and the high value of T in the `Iterative Greedy Peeling` algorithm. However, in many tests where the execution of `goldberg` is terminated due to slowness, the `Iterative Greedy Peeling` algorithm has been stopped within a maximum of 50 seconds.

5.3.3 Statistical Analysis of the Greedy-Peeling Approximation Ratio

For a statistical analysis of the approximation ratio of this algorithm, we first divide the test cases into three categories:

1. Test cases related to `worst-case` scenarios.
2. Test cases for which the `goldberg` algorithm did not provide an answer.
3. Test cases that do not fall into the above two categories.

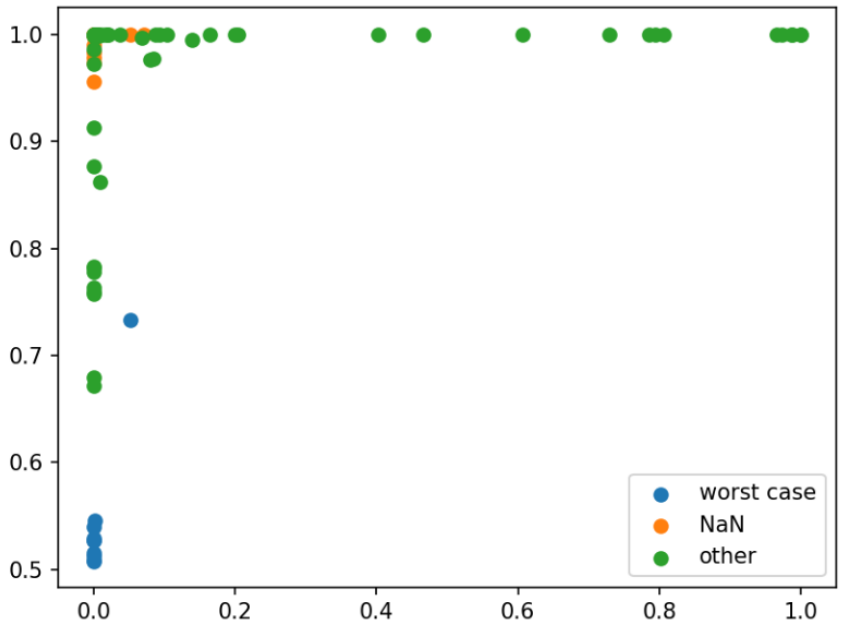

Figure 3: Approximation ratio based on graph density.

The results obtained are displayed in Figure 5.3.3. Note that the x-axis in the graph represents the graph density, calculated by the formula $\frac{|E|}{\binom{|V|}{2}}$.

As observed, the approximation ratio is consistently greater than or equal to 0.5. Also, in `worst-case` graphs, the approximation ratio is often very close to 0.5, indicating that in this category of graphs, the approximation ratio is tight.

On the other hand, in test cases where the `goldberg` algorithm did not provide an answer, the approximation ratio approaches 1 significantly. This is because these graphs are usually dense, and, as explained later, the approximation ratio tends to approach 1 in such cases. Furthermore, in these graphs, the solution has been compared to `Iterative Greedy Peeling`, which, due to the large size of the graph, results in a small value of T relative to the graph size, making the solutions of the two algorithms close to each other.

In other test cases, it can be seen that with increasing graph density, the approximation ratio approaches 1, while in sparser graphs, the approximation ratio exhibits more unpredictable behavior. This is because, as previously mentioned, the `greedy-peeling` algorithm effectively finds the w-cores of the graph, and this approach works poorly when the optimal subgraph contains low-degree nodes. This is why the algorithm's behavior is worse on sparser graphs.

5.3.4 Convergence of the Iterative Greedy Peeling Algorithm

As previously mentioned, the Iterative Greedy Peeling algorithm converges to the optimal solution as T increases. In practice, it can be observed that the algorithm converges to the optimal solution much faster than what is theoretically proven.

To investigate this matter, we examined the behavior of the algorithm on test cases such as "roadnet-TX," "com-amazon.ungraph," "sparse17," and "worst10" for various values of T. The results of this study are presented in Figure 4. Please note that in the "roadnet-TX" test case, where the Goldberg algorithm did not provide an answer, we compared the solution to the output of the Iterative Greedy Peeling algorithm for T = 50.

As expected, the Iterative Greedy Peeling algorithm converges to the solution very quickly on all of these test cases. This demonstrates that while the greedy-peeling algorithm may have an approximation ratio close to 0.5 on its own, running a limited number of iterations of this algorithm

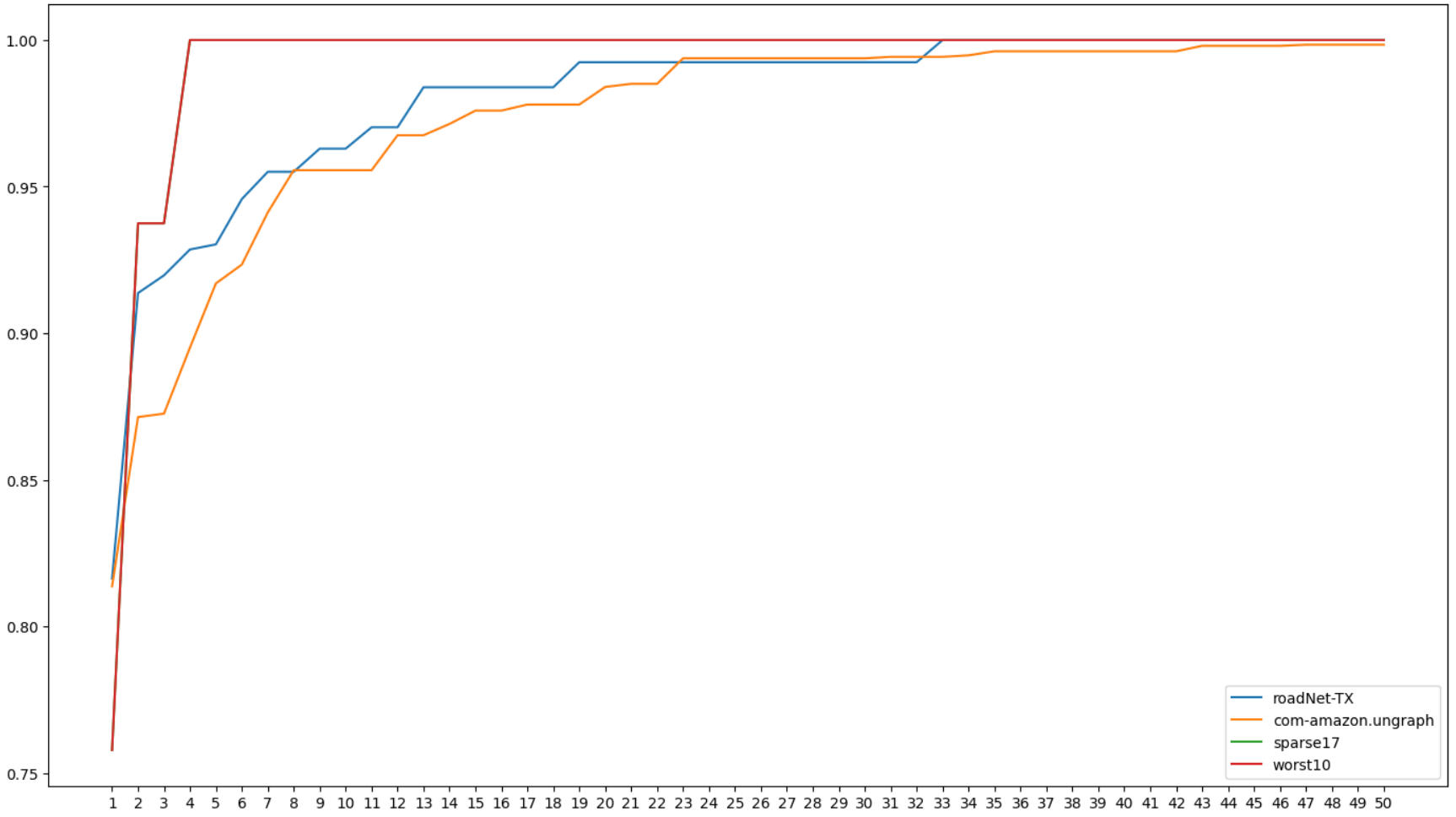

Figure 4: Approximation ratio of the Iterative Greedy Peeling algorithm on selected test cases for different values of T.

at high speed can quickly get close to the optimal solution.

6 Conclusion and Open Questions

As previously mentioned, the densest subgraph problem is an important problem in graph theory and combinatorial optimization. Due to its practical applications in various fields such as data extraction from graphs, clustering, as well as the analysis of DNA structures and virtual networks, it has received significant attention in recent years. Multiple versions of this problem exist, including directed densest subgraph, densest subgraph in hypergraphs, and densest subgraph in metric spaces. While we couldn't cover these versions in this article, interested readers can refer to [18], [15], and [10] for more details.

Moreover, for those interested in studying this problem in different computational models such as streaming, parallel, or distributed models, we recommend exploring [2], [21], [12], and [9].

In this report, we aimed to thoroughly examine important exact and approximation algorithms for this problem, explaining their correctness in simple terms. Furthermore, we endeavored to analyze the performance of these algorithms on various real-world and random instances, discussing the different results achieved in practice.

6.1 Open Questions

Despite the extensive research on this problem in recent years, there are still many open questions and challenges related to this problem. In this section, we introduce some of these open questions that have come to our minds or have been seen in reputable sources.

While it is true that the Iterative Greedy Peeling algorithm converges to the optimal solution, and it often performs much better than a 2-approximation algorithm, there is still room for improvement. It is possible to develop linear-time algorithms with better approximation ratios than 2, or even slightly modify the Greedy Peeling algorithm to handle worst-case scenarios where its approximation ratio approaches 2. We feel that there is potential for improving the current state of approximation algorithms for this problem.

Additionally, the proof of the Iterative Greedy Peeling algorithm's convergence was recently established in 2022 in [8]. There is still much room for further study to more precisely understand the full power, results, and limitations of this approach.

In various versions of this problem discussed in Section 4, there are many open questions, and the approximation ratios of algorithms given for these problems often have a significant gap with the known bounds on the approximation ratios for these problems.

References

[1] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In *Workshop on Algorithms and Models for the Web-Graph*, 2009.

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

[3] Francesco Bonchi, David Garc'ia-Soriano, Atsushi Miyauchi, and Charalampos E. Tsourakakis. Finding densest k-connected subgraphs. *Discret. Appl. Math.*, 305:34–47, 2020.

[4] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*, WWW '20, page 573–583, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[6] Leizhen Cai. Parameterized complexity of cardinality constrained optimization problems. *Comput. J.*, 51:102–121, 2008.

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

[8] 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. SIAM, 2022.

[9] A. Das Sarma, A. Lall, D. Nanongkai, and A. Trehan. *Dense subgraphs on dynamic networks*, volume 7611 LNCS of *Lecture Notes in Computer Science*, pages 151–165. Springer, January 2012. 26th International Symposium, DISC 2012. ; Conference date: 16-10-2013 Through 18-11-2013.

[10] Hossein Esfandiari and Michael Mitzenmacher. Metric sublinear algorithms via linear sampling. *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 11–22, 2018.

[11] Uriel Feige, Guy Kortsarz, and David Peleg. The dense k -subgraph problem. *Algorithmica*, 29:410–421, 2001.

[12] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrovic. Improved parallel algorithms for density-based network clustering. In *International Conference on Machine Learning*, 2019.

[13] Andrew V. Goldberg. Finding a maximum density subgraph. 1984.

[14] Naga Venkata Chaitanya Gudapati, Enrico Malaguti, and Michele Monaci. In search of dense subgraphs: How good is greedy peeling?, 11 2019.

[15] D.J.-H. Huang and Andrew B. Kahng. When clusters meet partitions: new density-based methods for circuit decomposition. *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 60–64, 1995.

[16] 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.

[17] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. In *J. Algorithms*, 1992.

[18] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. On directed densest subgraph discovery. *ACM Trans. Database Syst.*, 46(4), nov 2021.

[19] Giannis Nikolentzos, Polykarpos Meladianos, Yannis Stavrakas, and Michalis Vazirgiannis. K-clique-graphs for dense subgraph discovery. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 617–633. Springer, 2017.

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

[21] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *CoRR*, abs/2002.10047, 2020.

[22] Yinyu Ye and Jiawei Zhang. Approximation of dense-n/2-subgraph and the complement of min-bisection. *Journal of Global Optimization*, 25(1):55–73, 2003.