Received: 8 March 2021 | Accepted: 9 March 2021 | Published on: 20 March 2021

DOI: 10.1002/net.22034

SPECIAL ISSUE ARTICLE

In search of dense subgraphs: How good is greedy peeling?

Naga V. C. Gudapati | Enrico Malaguti | Michele Monaci

DEI, University of Bologna, Bologna, Italy

Correspondence
Enrico Malaguti, DEI, University of Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy.
Email: enrico.malaguti@unibo.it

Abstract

The problem of finding the densest subgraph in a given graph has several real-world applications, particularly in areas like social network analysis, protein, and gene networks. Depending on the application, finding dense subgraphs can be used to determine regions of high importance, similar characteristics, or enhanced interaction. The densest subgraph extraction problem is fundamentally a non-linear optimization problem. Nevertheless, it can be solved in polynomial time by an exact algorithm based on iteratively solving a series of max-flow subproblems. Despite its polynomial-time complexity, the computing time required by exact algorithms on very large graphs could be prohibitive. Thus, to approach graphs with millions of vertices and edges, one has to resort to heuristic algorithms. We provide an efficient implementation of a greedy heuristic from the literature that is extremely fast and has some nice theoretical properties. We also introduce a new heuristic algorithm that is built on top of the greedy and the exact methods. An extensive computational study is presented to evaluate the performance of various algorithms on a benchmark composed of 86 instances taken from the literature and real world. This analysis shows that the proposed heuristic algorithm is very effective on a large number of test instances, often providing either the optimal solution or a near-optimal solution within short computing times.

KEYWORDS

approximation, computational experiments, dense subgraphs, exact algorithms, heuristic algorithms, network optimization, worst-case analysis

1 | INTRODUCTION

A graph is a mathematical structure containing vertices and edges that is often used to represent different real-life scenarios. Besides very traditional applications in transportation, mapping, and logistics, graphs may also be used to describe many social, biological, financial, and technological systems. In these cases, vertices represent individuals, cells, proteins, components, and edges represent some kind of interaction between the vertices. As a result, *graph theory* is one of the most extensively researched areas in computer science.

Graph networks that arise in real-life applications have edges which are either weighted or unweighted. While unweighted edges simply represent some connection between two vertices, weighted edges can be used to indicate the importance of a connection in the graph, or the time required for traveling on a given edge, or the probability of an edge to occur in the network. The edges could be further directed or undirected: the former model one-way relationships, like the “follow” network in Twitter, while the latter are used for two-way connections, for instance, Facebook friendships.

Identification of dense areas is a very interesting problem in social network analysis. Intuitively, dense components in a graph can be considered to be subsets of highly-connected vertices that correspond to regions where there is more interaction among the vertices. For instance, consider a network describing the interactions between various Internet Service Providers, exchange points, customers, and other related parties: identifying dense subgraphs in this network allows us to detect critical points of failure, which could further help in planning for contingencies to mitigate unplanned service outages. Similarly, for social networks, dense subgraphs identify areas of common interests and communities. Many other examples where finding dense subgraphs is a key problem are detailed in [16] and in [9].

This article deals with the search for dense subgraphs in large graphs. This article is organized as follows. Section 2 discusses the definition of the problem and gives an overview of both the historical and more recent approaches to the *densest subgraph extraction (DSE)* problem. Section 3 reviews the existing literature, presenting the main exact approaches for computing an optimal solution of the DSE problem, and an existing heuristic algorithm known as *greedy peeling*. Section 4 introduces a new algorithm called the *hybrid* algorithm that is built on top of greedy peeling and an exact algorithm. All algorithms are computationally tested in Section 5 on a large set of graph instances taken from the literature including both unweighted and weighted graphs. Finally, Section 6 gives a summary and draws some conclusions.

This article has three main contributions. From a practical viewpoint, we introduce a simple heuristic algorithm that is built on top of the greedy heuristic and any exact method. Our proposed algorithm is typically very fast, produces solutions that improve over the greedy solution, and gives us near-optimal solutions.

From a theoretical point of view, we present a simple graph instance where the greedy peeling algorithm approaches its worst-case performance. To the best of our knowledge, there is only one other example in the literature showing a similar behavior of the greedy peeling algorithm (see [11]) but the example we present is simpler than the existing one. Besides, the example provided in [11] refers to a disconnected graph, and could be efficiently tackled by considering each connected component, one at a time. On the contrary, our example is a connected graph, for which we show that the greedy peeling algorithm achieves the theoretical worst-case performance.

Finally, from a computational perspective, we present a thorough experimental analysis, that is by far the most extensive in the literature for this class of problems. While most of the previous works in the literature have dealt with small or medium-sized instances, in this article, we make a considerable step forward concerning the instance size by considering graphs with tens of millions of vertices and hundreds of millions of edges. Our computational study shows that the practical performance of the *greedy peeling* is much better than its theoretical guarantee, and that a further improvement can be achieved with limited computational effort.

2 | DEFINITION OF THE PROBLEM

In this section, we give a formal definition of the problem. Let $G = (V, E)$ be an unweighted, undirected graph with vertex set V and edge set E. Throughout the text, we will assume that G is a simple graph, that is, there are no multiple edges connecting the same pair of vertices. The *density* of G, sometimes referred to as *average degree*, is defined as

$$f(G) = \frac{|E|}{|V|}, \tag{1}$$

and corresponds to the ratio between the number of edges and the number of vertices in the graph.

For a given subset of vertices $S \subseteq V$, we define $E(S)$ as the induced set of edges, that is, $E(S) = \{(u, v) \in E : u \in S, v \in S\}$, and $G(S) = (S, E(S))$ as the subgraph *induced* by S. When no confusion arises, we will write that set S has a density

$$f(S) = f(G(S)) = \frac{|E(S)|}{|S|} \tag{2}$$

Given an unweighted graph $G = (V, E)$, the DSE problem requires one to determine a subset $S \subseteq V$ of vertices that induces a subgraph of maximum density. Although it can be easily proved that there always exists an optimal solution to the DSE problem inducing a connected subgraph, we do not make any assumption on the input graph.

As already mentioned, in many applications, each edge $(u, v) \in E$ has a positive *weight* w_{uv}, which could, for instance, be used to represent the importance of a relationship between two vertices in the network. Weighted graphs can also be used to model a unique scenario where the actual edge set is unknown and each potential edge has an associated non-negative probability. In this probabilistic setting, one is interested in finding a subgraph that has a large probability to be the one with maximum density. This leads to a natural extension of the density definition in (1) to the edge-weighted graphs as

$$f^w(G) = \frac{\sum_{(u,v)\in E} w_{uv}}{|V|}. \tag{3}$$

Similarly, we can define the weighted density for a given subset $S \subseteq V$ of vertices.

The aforementioned density definitions are valid for undirected graphs only. For directed graphs, different definitions are typically used and we refer the interested reader to [6] and [15].

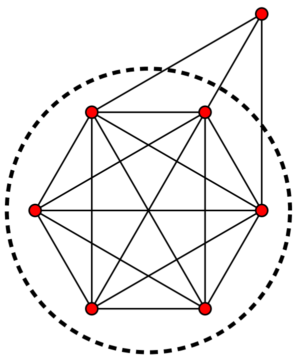

FIGURE 1 A small example in which a clique is not the densest subgraph [Color figure can be viewed at wileyonlinelibrary.com]

The DSE problem has been studied since the early 1980s. Though this problem is fundamentally an unconstrained non-linear optimization problem, it can still be solved efficiently. Indeed, a flow-based algorithm to get an optimal solution of the problem for unweighted graphs was introduced in [17] and it requires at most $|V|$ max-flow (min-cut) operations on a network of $|V|+2$ vertices, that is, it runs in polynomial time. Later, an alternative flow-based algorithm with better computational complexity was introduced in [12]. This algorithm determines the densest subgraph in only $\mathcal{O}(\log(|V|))$ max-flow operations and can easily be extended to weighted graphs. Finally, a parametric max-flow algorithm which can solve the DSE with a single max-flow computation was given in [10]. This parametric max-flow algorithm improves upon the complexity of the previous method described in [12] by a factor of $\log(|V|)$.

Though solvable in polynomial time, computing densest subgraphs using flow-based algorithms could be very time consuming for very large graphs. Thus, when real-world applications with millions of vertices and edges are considered, one has to resort to heuristics. One of the most important heuristic algorithms for the DSE problem is the greedy peeling introduced in [3]. Besides being very fast in practice, this algorithm has nice theoretical properties. It has been proved in [6] that this algorithm has a worst-case 2-approximation, that is, the density of the subgraph found by greedy peeling is at least half of the density of the optimal subgraph. The algorithm can be implemented to have time complexity of $\mathcal{O}(|E| + |V|)$ in case of unweighted graphs and $\mathcal{O}(|E| + |V|\log(|V|))$ in case of weighted graphs. Finally, we mention a variant of the greedy peeling algorithm, introduced in [4], that can be implemented in a distributed way and for which the input is not stored, in order to reduce the memory requirement. This algorithm makes $\mathcal{O}(\log(|V)|)$ passes over the input graph and uses $\mathcal{O}(|V|)$ main memory, and has a worst-case approximation equal to $(2 + 2\epsilon)$ for any $\epsilon > 0$.

In some applications, additional constraints are imposed to limit (either from below or from above) the size of set S; in this case, the resulting problem becomes an $\mathcal{NP}$-hard problem. An extensive discussion on finding dense subgraphs with size bounds can be found in [2].

Many alternative definitions of density have been proposed in the literature. Indeed, the average-degree definition may produce subgraphs that have a large number of vertices, and are not extensively connected. For instance, a clique, which is intuitively a dense subgraph, might not be the densest subgraph according to the average degree, as another larger and loosely connected subgraph could produce a bigger ratio according to (1). Figure 1 shows an example in which the whole graph corresponds to the densest subgraph, with a density of $\frac{18}{7} = 2.57$, although a clique exists (defined by the vertices in the dashed circle) that has a density equal to $\frac{15}{6} = 2.5$. Additional considerations about the downsides of using definition (1) as a metric to find the dense subgraphs are given in many papers from the literature. A different density metric, called *quasi-clique*, was introduced in [19]; according to this definition, the density of graph $G=(V,E)$ is given by $f(G) = |E(S)| - \alpha \binom{|S|}{2}$, where α is a tuning parameter. The authors in [19] claim that quasi-clique metric is better than average-degree, as it was shown that quasi-clique produces subgraphs that are tightly connected and smaller. In the same vein as [19], authors in [20] proposed another density metric called *discounted average degree* as $f(S) = \frac{|E(S)|}{|S|^{\beta}}$, where β is a parameter that can be chosen to affect the size of the desired subgraph. They also give four desirable properties of a density metric and show that their discounted average degree metric performs well on satisfying those four properties. Other than these two definitions, also depending on the type of graph, there have been many other proposed definitions of density, including edge ratio, triangle density, and triangle ratio, and others (see [1, 5, 7], and [18]).

Despite these alternatives, there is no clear consensus on using any of them as standard, and average degree remains the most common and accepted.

```
***Goldberg's algorithm:*** input = $G(V,E)$
initialize: $\ell := 0,\ u := |E|, S^E = \emptyset$;
while $u - \ell \geq \frac{1}{|V|(|V|-1)}$ do
    $g := \frac{u+\ell}{2}$;
    define the augmented network $A(g)$ associated with $g$;
    find the minimum cut $(S,T)$ in $A(g)$;
    if $S = \{s\}$ then $u := g$
    else
        $l := g$;
        $S^E := S \setminus \{s\}$;
end while
return $S^E$
```

FIGURE 2 Goldberg's algorithm

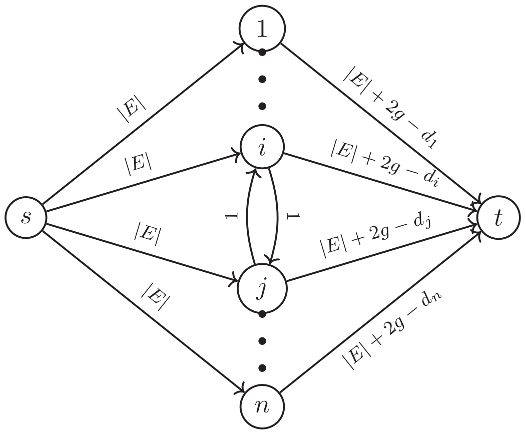

FIGURE 3 Augmented network $A(g)$, courtesy of [12]

3 | ALGORITHMS

In this section, we discuss solution approaches for the DSE problem that have been proposed in the literature. The next subsection describes an exact algorithm and a mathematical formulation of the DSE, while Section 3.2 presents a greedy heuristic and analyzes its theoretical performance.

3.1 | Exact algorithms

The first exact algorithm we consider is *Goldberg's* algorithm which has been introduced in [12] and is a relatively fast exact algorithm to compute a densest subgraph in a given graph G. For the sake of completeness, we report the algorithm's pseudocode in Figure 2. The algorithm iteratively guesses the solution value, solves a max-flow problem on an augmented network, and updates the value of the guess.

Figure 3 shows an illustration of an augmented network for a given guess g. The vertex set in the network is $V \cup \{s,t\}$, that is, there are $|V|+2$ vertices. Each edge in G is replaced by two reverse arcs with unit capacity. In addition, there is an arc from vertex s to each vertex $v \in V$ with capacity $|E|$, and an arc from each vertex $v \in V$ to vertex t with capacity $(|E|+2g - d_v)$, where d_v is the degree of vertex v with respect to G.

At each iteration, the algorithm defines the augmented network $A(g)$ associated with the current guess g and computes a max $s-t$ flow (minimum cut) on this network. Depending on whether the minimum cut isolates vertex s, or instead separates the vertices in V in two nonempty subsets, the current g value reveals itself either a lower or an upper bound on the optimal density. The algorithm updates these bounds accordingly, until the difference between lower and upper bound is below some threshold.

It was proved in [12] that, as the optimal g value can only take a finite set of values in the interval $[0, |E(S)|]$, the binary search converges to the optimal value and the number of iterations is bounded by $\mathcal{O}(\log(|V|))$. There are many efficient algorithms for

```
Greedy Peeling: input = $G(V, E)$
initialize: $n := |V|$, $S_n := V$;
for $i = n$ to 1 do
    let $u$ be the smallest degree vertex in $G(S_i)$;
    $S_{i-1} := S_i \setminus \{u\}$;
endfor
$S^H \in \arg\max_{i=1,\ldots,n} f(S_i)$;
return $S^H$
```

FIGURE 4 Greedy peeling algorithm for the unweighted case

solving max-flow problem (see, e.g., [14]). Using the Push-Relabel algorithm (see [13]), the max-flow problem can be solved in $\mathcal{O}(|V|^3)$ time, producing an overall $\mathcal{O}(\log(|V|)\ |V|^3)$ time complexity for Goldberg's algorithm.

A completely different exact solution method has been proposed in [6]. This approach describes the DSE problem by means of a *linear programming* (LP) model, that can be solved using any general-purpose LP solver. The LP model can easily be extended to the weighted case with minor modifications. The model has $|V|+|E|$ variables and two constraints per edge, that is, its size is polynomial in the size of the input graph. Despite this, the constraint matrix of the formulation can be massive and the memory requirements to solve the model can be prohibitive for large graph instances. Typically this produces computational performances that are worse than those of the flow-based Goldberg algorithm discussed above. However, the LP model provides a good foundation for finding dense subgraphs in directed graphs and its related proofs as discussed in [6].

3.2 | Greedy peeling algorithm

For very large graphs, the application of the exact algorithms described in the previous section may require large memory and long computational times. This is where heuristic approaches can be used for getting reasonably good solutions quickly. The heuristic algorithm described in this section produces subgraphs whose density is usually close to an optimal one.

As the objective of DSE is to find a subgraph with best average degree, the algorithm consists of starting with the initial graph and removing, one at a time, a vertex with the smallest degree in the current graph. The resulting algorithm, called greedy peeling, is described in Figure 4 and can be naively implemented to run in $\mathcal{O}(|V|^2)$ time. To prove the time complexity, it is enough to observe that there are n iterations; each iteration requires $\mathcal{O}(|V|)$ time to find the vertex u with minimum degree with respect to the current subgraph (breaking ties arbitrarily), and another $\mathcal{O}(|V|)$ time to update the subgraph once u has been removed. A more efficient implementation can be obtained using a "degree-lists" data structure, in which a list is defined for each possible value of the degree of a vertex. All vertices with same degree are placed in the same list and lists are ordered by increasing degree. Using this data structure, the determination of the next vertex u to be removed can be done in constant time, taking an arbitrary vertex in the first non-empty list. Since removing vertex u decreases the degree of its neighbors by one unit, updating the graph (essentially data-lists) can be done by moving each neighbor of u from its current list to the previous one (i.e., to the list with degree one less than current degree). Since the number of vertex movements among the lists is equal to the number of edges of G, the time complexity of the algorithm is $\mathcal{O}(|E|+|V|)$. The results in this article (see Section 5) correspond to this implementation of the algorithm.

3.2.1 | Extension to the weighted case

The greedy peeling algorithm can easily be extended to the weighted case by selecting, at each iteration, vertex u as the one having the minimum *weighted-degree*, that is, the weighted sum of all the incident edges with respect to the current subgraph. However, the linear time complexity of the algorithm is not preserved because the degree-lists data structure cannot be used for graphs with general weights. Using Fibonacci heaps to determine, at each iteration, the minimum weighted-degree vertex, the algorithm runs in $\mathcal{O}(|E| + |V| \log(|V|))$, see [6]. The degree-lists implementation could also be used to determine weighted dense subgraphs, similar to the unweighted case, if weights are either integer numbers or are all scaled to integers. Assume the weights are integers and let u be the vertex that has currently been selected for removal. The weighted-degree of each neighbor of u, say vertex v, is decreased by an amount w_{uv}, instead of one as in the unweighted case. However, using degree lists may yield to a considerable worsening in the performance of the algorithm as the number of lists to be considered is bounded by the maximum weight degree of all vertices, that is, it is pseudo-polynomial in the size of the input (and is strongly dependent on the number of significant digits in the weight values, if they are not integers).

To avoid this issue, we use binary heaps to implement the greedy peeling algorithm for the weighted case. A binary heap data structure is a complete binary tree which satisfies heap ordering. In particular, we use the min-heap property, which requires that the value of each node in the tree is greater than or equal to the value of its parent node. Initially, we compute the weighted-degree of each vertex and insert all the vertices in a heap data structure satisfying the min-heap property. At each iteration, determining

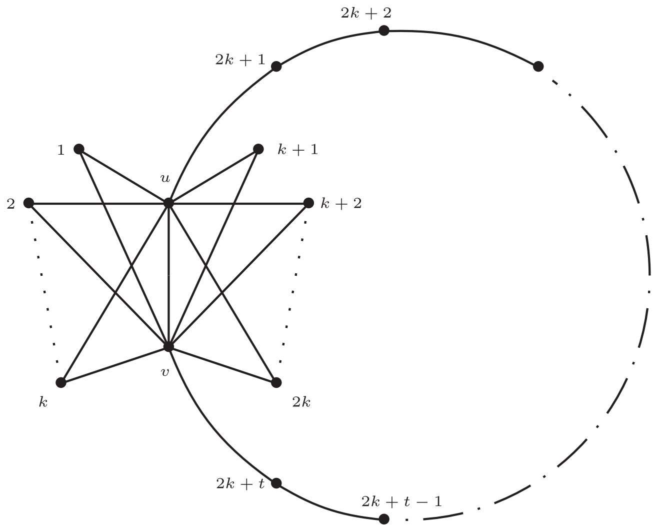

FIGURE 5 Bad connected instance for the greedy peeling algorithm. The graph has $2k + t + 2p$ vertices and $4k + t + 2$ edges

the next vertex to be removed can be done in constant time, as the minimum value is associated with the root node of the tree. Once a vertex has been removed, updating the weighted degree of its neighbors and rearranging those vertices in the heap can be done in $\mathcal{O}(\log(|V|))$ time. In the following, we will report results for this implementation, which works for both rational and integer weights, and is very fast in practice even for large graphs.

3.2.2 | Worst-case analysis

The theoretical performance of the greedy peeling algorithm was analyzed in [6] (and in [3] for a constrained version of the DSE problem), where the worst-case performance ratio of the algorithm was proved to be equal to 2. To the best of our knowledge, the only example for which the approximation is asymptotically tight has been given in [11].

In the following, we present a simpler instance, where the worst-case performance ratio is approached. In addition, while the example reported in [11] is based on a disconnected graph, the following instance refers to a connected graph. To the best of our knowledge, this is the first example showing that the worst-case performance ratio of the algorithm may be hit for connected graphs as well. This result provides a relevant piece of information about the performance of the greedy peeling algorithm; indeed, it shows that, given a disconnected graph, the worst-case approximation provided by the algorithm cannot be improved by sequentially considering all the connected components, one at a time.

The instance shown in Figure 5 is a graph G which has two vertices u and v connected by an edge; both vertices u and v are also connected to additional $2k$ vertices indexed by $\{1, \ldots, 2k\}$ by $4k$ edges. Vertices u and v are also connected by a path consisting of another set of t vertices and $t + 1$ edges. Thus, graph G has $2 + 2k + t$ vertices and $1 + 4k + t + 1 = 4k + t + 2$ edges.

At the first iteration the greedy peeling algorithm considers the full graph, which has a density equal to $f(G) = \frac{4k+t+2}{2k+t+2}$. In the first $2k$ iterations, all vertices but u and v have degree 2. Breaking ties by lowest index, the algorithm removes, in turn, vertices $1, 2, \ldots, 2k$. Each vertex removal induces the elimination of two edges from the remaining subgraph; it is easy to see that the resulting density cannot be larger than $f(G)$. When vertices $1, 2, \ldots, 2k$ have been removed, the remaining subgraph is a cycle spanning $t + 2$ vertices. Regardless the order in which the vertices are removed, the algorithm encounters subgraphs having a density smaller than the initial one. Thus, the greedy peeling algorithm returns a heuristic solution with value $f^G = \frac{4k+t+2}{2k+t+2}$.

An optimal solution is defined by vertex set $\{1, 2, \ldots, 2k\} \cup \{u, v\}$. The induced subgraph has $2k + 2$ vertices and $4k + 1$ edges, hence the optimal solution value is $f^* = \frac{4k+1}{2k+2}$. Thus, the ratio between the optimal and the approximate solution values is given by

$$\frac{f^*}{f^G} = \frac{\frac{4k+1}{2k+2}}{\frac{4k+t+2}{2k+t+2}} = \frac{(4k+1)(2k+t+2)}{(2k+2)(4k+t+2)} \tag{4}$$

Taking $t = k^2$ we have that $\frac{f^*}{f^G}$ is arbitrarily close to 2 for sufficiently large values of k.

Finally, observe that a simple adaptation to the weighted case of the worst-case analysis given in [6] shows that, also in this case, the greedy peeling algorithm returns a solution value which is at least half of the optimal density. As graph G is a weighted instance with all weights equal to 1, this results shows that the worst-case approximation ratio is tight in the weighted case as well.

```
***Hybrid***: input $= G(V, E)$
// *Greedy Peeling*
$S^1 :=$ ***Greedy Peeling***$\big(G(V, E)\big)$;
// *Expansion* phase
$S^2 := \{v \in V : (u, v) \in E$ for some $v \in S^1\}$;
$E^2 := \{(u, v) \in E : u \in S^2, v \in S^2\}$;
// *Exact* phase
$S^H :=$ ***Exact***$\big(G(S^2, E^2)\big)$;
**return** $S^H$
```

FIGURE 6 Hybrid algorithm

```
***Expansion***: input $= S^1, V, E$
$S^2 := \emptyset$, $E^2 := \emptyset$;
//consider each vertex $u$ in the input solution
**for each** $u \in S^1$ **do**
    $S^2 := S^2 \cup \{u\}$;
    // add all neighbors of $u$
    **for each** $v \in V : (u, v) \in E$ **do**
        **if** $v \in S^1$ **then**
            **if** $v \in S^2$ **then** $E^2 := E^2 \cup \{(u, v)\}$;
        **else**
            $S^2 := S^2 \cup \{v\}$, $E^2 := E^2 \cup \{(u, v)\}$;
            // add edges between vertices that both are in $S^2 \setminus S^1$
            **for each** $k \in S^2 \setminus S^1 : (v, k) \in E$ **do** $E^2 := E^2 \cup \{(v, k)\}$;
        **endif**
    **endfor**
**endfor**
**return** $G(S^2, E^2)$
```

FIGURE 7 Expansion phase

4 | HYBRID ALGORITHM

In this section, we present a hybrid algorithm that combines the greedy peeling algorithm and an exact algorithm to improve the greedy solution value. The algorithm is given in Figure 6 and consists of three phases, namely, greedy peeling, expansion phase, and exact phase. The first phase corresponds to the execution of the greedy peeling algorithm discussed in Section 3.2 and is intended to quickly produce an initial solution. Using this initial greedy solution, the expansion phase obtains a "core" subgraph, which is likely to contain either all or most of the vertices in an optimal solution. Finally, the exact phase solves the DSE problem on the core using an exact algorithm, for instance, the flow-based Goldberg algorithm or the LP approach described in Section 3.1.

The expansion phase takes as input a subset of vertices S^1, possibly identified by the greedy peeling algorithm, expands the vertex set by adding all those vertices that are neighbors of one vertex in S^1, and defines the induced edge set E^2. An implementation of this phase is described in Figure 7. Set S^2 includes all the vertices that are currently included in the expanded graph. Before the expansion phase, $S^2 = \emptyset$. In the expansion phase, we considers all vertices in S^1, one at a time. For each $u \in S^1$, we consider all its neighbors; if the current neighbor v is in $S^1 \cap S^2$, we add the edge (u, v) to E^2. If $v \notin S^2$, we add vertex v to S^2 and edge (u, v) to E^2, and scan all neighbors of v; for each neighbor k that is currently in set S^2 we also add an edge (v, k) to E^2.

Figure 8 gives an example of the expansion phase. The original graph has 12 vertices and $S^1 = \{5, 6, 7, 8\}$. At first, $S^2 = \emptyset$. We can start at vertex $u = 5$ which makes $S^2 = \{5\}$ and consider the first of its neighbors, that is, vertex $v = 2$. From the algorithm, we can add vertex 2 to S^2 and the edge $(2, 5)$ to E^2. Now, we scan the neighbors of 2 and we can add an edge to E^2 if any of the neighboring vertices of 2 are present in S^2. Since no new neighboring vertices of 2 (essential vertex 1) are present in S^2, we do not add any new edges to E^2. So we have $S^2 = \{5, 2\}$ and $E^2 = \{(2, 5)\}$. Then, we examine the other neighboring vertices of 5, namely, 6, 7, and 8. As all these vertices belong to S^1 and none of them are in S^2, no action is taken. Then, we move on to the next member in S^1, that is, $u = 6$. We add 6 to S^2 and examine the neighbors of 6. We have vertex $v = 3$ that can be added to S^2 and the edge $(3, 6)$ can be added to E^2. Now, $S^2 = \{5, 2, 6, 3\}$ and $S^2 \setminus S^1 = \{2, 3\}$, implying that edge $(3, 2)$ has to be added to E^2. Since no other edge can be added, we then move on to the next neighbor of 6, namely 5. When considering this vertex, edge $(6, 5)$ can

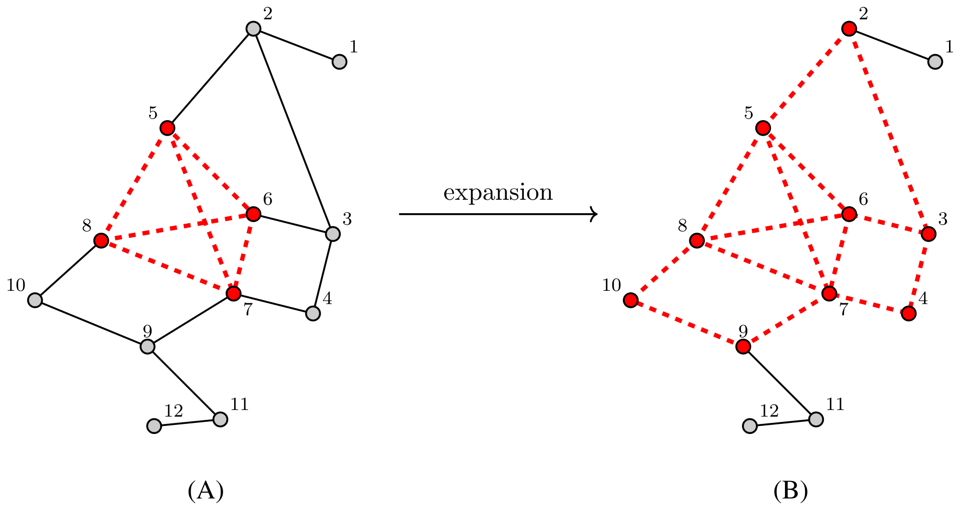

FIGURE 8 Expansion phase example [Color figure can be viewed at wileyonlinelibrary.com]

be added to E^2 as 5 is in both S^1 and S^2. As all the neighbors of 6 have been considered, we move on to the next vertex in S^1, that is, 7. We continue doing the above process for all the members in S^1 until we get the expanded subgraph, shown in Figure 8(B).

In the third phase, an exact algorithm is applied to the graph obtained by the expansion phase. Typically this graph is much smaller than the original one, allowing a fast execution of the exact algorithm. In addition, if the flow-based algorithm is used, the greedy solution value, combined with the 2-approximation guarantee of the method, produces good initial lower and upper bounds for the value of the density, which can be used to speed up the binary search. The biggest caveat is that there are instances for which the greedy peeling algorithm produces very large subgraphs. In this situation, the expansion phase may require a very long computing time, and often returns the original graph, making this approach impractical.

5 | COMPUTATIONAL EXPERIMENTS

5.1 | Setup and programming

All the algorithms described in this article were implemented in C++ using standard containers, like `std::vector`, `std::queue`. We used the GCC compiler with a high level of optimization enabled (`-O3`). All our experiments were executed on a computer equipped with an Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10 GHz CPU and 16 GB of RAM; all computing times (t) given subsequently are expressed in milliseconds.

In the following, we report the results obtained using the three algorithms, namely:

- The greedy peeling discussed in Section 3.2.
- The hybrid algorithm of Section 4.
- The flow-based exact algorithm (Goldberg's algorithm) of Section 3.1. This algorithm embeds a push-relabel algorithm to compute the max-flow (min-cut) with $\mathcal{O}(|V|^3)$ time complexity. It should be noted that this algorithm requires us to construct an augmented network which has more than twice the number of edges than the original graph network. As a result, the augmented network could occupy very large space in memory, and hence the algorithm may fail for memory requirement on very large instances.

We analyzed both weighted and unweighted instances (see below). For weighted instances, the greedy peeling was implemented using binary heaps as this solution turned out to be much more efficient than using the degree-list implementation. As to Goldberg's algorithm, it required very minor modifications for handling the weighted case as well. The hybrid algorithm uses the greedy peeling algorithm and the Goldberg algorithm as modular components to create and solve the expanded subgraph respectively, while the expansion phase clearly is not affected by the presence of weights on the edges.

5.2 | Testbed

All the instances, both unweighted and weighted, were taken from Suite Sparse Collection [8]. To select a meaningful set of instances, we considered graphs that:

(i) are classified as *undirected graph* or *undirected weighted graph* or *undirected graph with communities* or *undirected random graph*;

(ii) have at least 20 000 vertices;
(iii) have at most 65 000 000 vertices and 150 000 000 edges; and
(iv) only have positive weights (for weighted instances).

As discussed in Section 2, the definition of density for directed graphs is significantly different than for undirected graphs and hence (i) we only considered the latter. Since the DSE is solvable in polynomial time, and hence optimal solutions are obtained with limited effort for small graphs, (ii) we chose to only consider instances that are "not too small" and may therefore be challenging for our algorithms. We also imposed some upper bounds on the size of the graphs as any graphs which are bigger than the ones mentioned in (iii) cannot be solved by any algorithm on our machine as all of them run out of memory. And finally, (iv) we only considered graphs with positive weights, as all algorithms discussed in this article do not have a straightforward extension when negative weights are considered. For instance, Goldberg's algorithm would fail in case of negative weights.

This produced a testbed with 170 instances. The benchmark includes 50 census-based weighted graphs (like xx2010 in Table 8) that have very similar characteristics. To avoid presenting very similar results, we decided to consider only the ten largest among these instances. In addition, we have also considered three large directed graphs (called Wikipedia instances) that were present in the computational analysis in [19]; for these instances, minor modifications were required, for example, converting directed arcs to undirected edges and removing duplicated edges. Finally, we do not present the results on some graphs where the greedy algorithm fails.

The majority of the graphs in our testbed are unweighted and hence we have further partitioned them into different buckets, depending on their size. The *medium* bucket contains those instances which have less than 1 000 000 vertices. The *large* bucket contains instances having more than 1 000 000 vertices but less than 10 million vertices and less than 50 000 000 edges. Finally, the *massive* bucket includes all the remaining instances.

5.3 | Analysis of instances in the medium bucket

In this section we report the outcome of our computational experiments on the instances in the first bucket, that contains 41 instances.

Table 1 gives the results and reports, for each instance, the following information:

- The name of the instance and the main characteristics of the graph.
- For the greedy peeling algorithm: the required computing time t_G and the associated density value f_G.
- For the hybrid algorithm: the computing time for the expansion phase and for the exact phase (t_2 and t_3, respectively), the overall computing time t_H of the algorithm and the density value f_H of the best solution found.
- For the exact algorithm (Goldberg's algorithm): the required computing time t_E and the density value f^*.

If an algorithm runs out of memory during its execution, we report the failure by "–." The bold numbers in the table indicate the best density found. In case of ties, the density of the fastest algorithm is boldfaced.

Results in Table 1 show that Goldberg's algorithm can handle this set of instances quite efficiently: the required computing time is equal to 162 seconds on average and no failure was experienced due to memory reasons. The greedy peeling algorithm, though having a worst-case performance ratio equal to 2, gives a very tight approximation on the optimal density in practice, as the average gap with respect to the optimal density is 3.12%. In addition, this algorithm is very fast, the average CPU time being around 0.09 seconds.

The hybrid algorithm has good performances, as it improves over the greedy solution in 27 cases, but it runs out of memory for instance mycielskian17; on the remaining 40 instances, the average percentage gap of the algorithm is about 1.14%. The table shows that there are a number of instances for which the hybrid algorithm performs poorly in terms of computing time. In Table 2, we report all the instances where the ratio of $\frac{t_H}{t_E} > 0.75$ and where the hybrid algorithm runs out of memory. For each such instance, the table gives:

- The name of the instance.
- The number of vertices $|S^1|$ in the subgraph produced by greedy peeling and the ratio between $|S^1|$ and the total number of vertices V.
- The number of vertices and edges ($|S^2|$ and $|E^2|$, respectively) in the expanded subgraph and the ratio between $|S^2|$ and the total number of vertices V.

Table 2 shows that the hybrid algorithm encounters difficulties while dealing with instances where the solution produced by greedy peeling has almost the same number of vertices as the whole graph. And sometimes, even when greedy peeling produces a smaller and more compact solution, the expansion phase produces either the original graph or almost the original graph. For these specific instances, which are identified by the ratio $|S^2|/|V|$ being close to 1, the expansion phase may be time consuming,

TABLE 1 Results on instances in the medium bucket

Graph properties			Greedy peeling		Hybrid				Goldberg's	
Instance	$\lvert V\rvert$	$\lvert E\rvert$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
144	144 649	1 074 393	53	7.4416	30 114	259 526	289 694	7.4559	280 444	**7.4559**
598a	110 971	741 934	37	6.8043	5 066	35 843	40 947	**6.8792**	73 151	6.8792
as-22july06	22 963	48 436	3	**19.9423**	11	737	751	19.9423	1317	19.9423
auto	448 695	3 314 611	181	7.4495	89 415	310 410	400 007	7.5211	622 512	**7.5213**
ca-CondMat	23 133	93 439	4	12.5000	<1	8	13	**13.3667**	2336	13.3667
caidaRouterLevel	192 244	609 066	50	25.5167	9	223	282	**25.7750**	23 785	25.7750
citationCiteseer	268 495	1 156 647	96	12.0019	205	6268	6570	**12.1808**	59 115	12.1808
coAuthorsCiteseer	227 320	814 134	60	**43.0000**	4	58	123	43.0000	25 229	43.0000
coAuthorsDBLP	299 067	977 676	78	57.0000	5	74	158	**57.0690**	35 584	57.0690
com-Amazon	334 863	925 872	104	3.8327	2163	8674	10 942	**4.8041**	53 902	4.8041
com-DBLP	317 080	1 049 866	94	56.5000	6	77	178	**56.5652**	38 550	56.5652
coPapersCiteseer	434 102	16 036 720	296	**422.0000**	229	3316	3842	422.0000	205 527	422.0000
coPapersDBLP	540 486	15 245 729	358	**168.0000**	67	2324	2752	168.0000	233 183	168.0000
cs4	22 499	43 858	2	1.9493	247	8059	8309	**1.9526**	9008	1.9526
dblp-2010	326 186	807 700	62	**37.0000**	4	15	82	37.0000	26 000	37.0000
delaunay_n15	32 768	98 274	4	**2.9991**	710	12 997	13 712	2.9991	14 375	2.9991
delaunay_n16	65 536	196 575	10	**2.9995**	2835	50 002	52 848	2.9995	49 406	2.9995
delaunay_n17	131 072	393 176	23	**2.9997**	11 103	147 668	158 794	2.9997	145 617	2.9997
delaunay_n18	262 144	786 396	47	**2.9999**	44 140	394 457	438 645	2.9999	427 411	2.9999
delaunay_n19	524 288	1 572 823	96	**2.9999**	176 182	1 740 164	1 916 442	2.9999	1 768 799	2.9999
dictionary28	52 652	89 038	5	**12.5000**	1	4	11	12.5000	2634	12.5000
fe_body	45 087	163 734	7	3.9043	2	159	168	3.9213	5421	**4.0490**
fe_ocean	143 437	409 593	22	2.8734	6533	49 005	55 561	2.8964	80 359	**2.8966**
fe_rotor	99 617	662 431	27	6.6571	12 459	146 689	159 176	**6.6920**	159 632	6.6920
fe_tooth	78 136	452 591	20	5.9171	2319	25 546	27 885	5.9778	58 032	**5.9801**
loc-Brightkite	58 228	214 078	11	40.5571	12	492	515	**40.5591**	6124	40.5591
loc-Gowalla	196 591	950 327	62	43.8000	174	11 902	12 139	**43.8018**	32 753	43.8018
luxembourg_osm	114 599	119 666	10	1.1548	2	2	15	1.2667	4338	**1.5238**
m14b	214 765	1 679 018	79	7.8266	71 078	185 721	256 879	7.8694	238 330	**7.8694**
mycielskian15	24 575	5 555 555	101	**333.5567**	30 001	97 961	128 064	333.5567	107 600	333.5567
mycielskian16	49 151	16 691 240	322	**530.8705**	175 641	305 244	481 208	530.8705	344 396	530.8705
mycielskian17	98 303	50 122 871	1092	**845.8977**	–	–	–	–	1 165 647	845.8977
rgg_n_2_15_s0	32 768	160 240	6	7.5500	<1	1	8	7.6522	3336	**7.8947**
rgg_n_2_16_s0	65 536	342 127	16	7.6471	<1	2	19	**9.000**	7824	9.0000
rgg_n_2_17_s0	131 072	728 753	39	8.0000	1	1	42	8.2083	20 552	**8.9200**
rgg_n_2_18_s0	262 144	1 547 283	87	10.0769	3	2	93	**10.4242**	45 015	10.4242
rgg_n_2_19_s0	524 288	3 269 766	190	8.9474	5	1	197	**10.1667**	125 960	10.1667
t60k	60 005	89 440	5	1.4905	1036	91 590	92 632	1.4914	83 854	**1.4914**
usroads	129 164	165 435	19	1.5789	1	<1	21	1.6250	11 992	**1.7528**
usroads-48	126 146	161 950	18	1.5714	2	1	22	1.6250	14 238	**1.7528**
wing	62 032	121 544	9	1.9596	1897	46 318	48 225	**1.9627**	53 894	1.9627

Note: All times are in milliseconds.

and the application of Goldberg's algorithm after *expansion* requires similar computing time as applying Goldberg's algorithm to the original instances. Thus, the hybrid algorithm may overall be even slower than the direct application of Goldberg's algorithm on the initial graph instances. The average computing time taken by the hybrid algorithm for instances in the medium bucket is around 115 seconds; if we exclude the 14 pathological instances listed in Table 2, the time taken by the hybrid algorithm falls to around 21 seconds.

5.4 | Tuning of the algorithm

In this section, we present some additional results for evaluating variants of the hybrid algorithm. In particular, we consider:

- `H1`: This algorithm is aimed at evaluating the effect of the expansion phase. In this scheme we simply disabled the expansion phase of the hybrid algorithm and executed Goldberg's algorithm on the subgraph produced by greedy peeling. However, since the output of the latter consists of a set of vertices only (S^1), the associated edges have to be reconstructed and stored.

TABLE 2 Instances for which the hybrid algorithm can take a very long time

Graph properties	Greedy peeling		Hybrid		
Instance	$\|S^1\|$	$\|S^1\|/\|V\|$	$\|S^2\|$	$\|E^2\|$	$\|S^2\|/\|V\|$
144	137 542	0.9509	138 830	1 032 694	0.9598
cs4	22 498	1.0000	22 499	43 858	1.0000
delaunay_n15	32 767	1.0000	32 768	98 274	1.0000
delaunay_n16	65 535	1.0000	65 536	196 575	1.0000
delaunay_n17	131 071	1.0000	131 072	393 176	1.0000
delaunay_n18	262 143	1.0000	262 144	786 396	1.0000
delaunay_n19	524 287	1.0000	524 288	1 572 823	1.0000
fe_rotor	98 214	0.9859	98 971	658 472	0.9935
m14b	206 912	0.9634	210 693	1 647 651	0.9810
mycielskian15	9078	0.3694	24 575	5 555 555	1.0000
mycielskian16	16 436	0.3344	49 151	16 691 240	1.0000
mycielskian17	28 496	0.2899	98 303	50 122 871	1.0000
t60k	59 866	0.9977	59 935	89 313	0.9988
wing	61 852	0.9971	61 994	121 461	0.9994

TABLE 3 Average computing time and percentage gap for different variants of the hybrid algorithm

	Greedy peeling		`H1`		Hybrid		`H2`		`H3`	
Instances	t	%gap	t	%gap	t	%gap	t	%gap	t	%gap
ALL	68	3.2015	87 320	2.1986	115 199	1.1427	205 276	1.1420	134 274	1.1419

Note: All times are in milliseconds.

- `H2`: This algorithm is used to evaluate possible solution improvements obtained by repeatedly performing the expansion and exact phases. The algorithm operates in two steps: first, it executes the hybrid algorithm and stores the associated solution. Then, this solution is expanded again using the expansion phase and Goldberg's algorithm is invoked on the resulting subgraph.
- `H3`: This algorithm is intended to evaluate if a different way to expand could produce a larger graph, allowing Goldberg's algorithm to determine a subgraph with better density. In particular, given the set S^1 of vertices produced by greedy peeling, this algorithm executes the expansion phase twice in sequence. In this way, the graph which is used as input for Goldberg's algorithm includes the neighbors of the vertices in S^1 and also the neighbors of the neighbors.

In these experiments, we consider again the 40 instances in the medium bucket except for mycielskian17. Table 3 gives results for the three schemes above, as well as for the greedy peeling and for the hybrid algorithm described in Section 4. For each algorithm, we report the average values of the computing time (in milliseconds) and average percentage gap with respect to the optimal solution. The statistics are computed with respect to all instances in the medium bucket except for the instance mycielskian17.

These results show that variants `H2` and `H3`, while requiring additional computational effort when compared to the hybrid algorithm, only produce negligible improvements in the solution quality.

The situation is less clear for `H1`, which is computationally less expensive than the hybrid algorithm, but also finds solutions of lower quality; for this reason, we analyzed the performance of these two algorithms on a restricted subset of instances. According to the results in Table 2, the hybrid algorithm performs badly if Goldberg's algorithm is applied to a graph whose size is comparable with the original one. For this reason, we removed all instances for which our solution approach has small probability of being successful, and selected the instances for which $|S^2|/|V|<0.85$ (respectively, $|S^1|/|V|<0.85$ for `H1`). Table 4 reports the statistics with respect to these instances only. As the number of these instances depends on the algorithm, we also report the number of instances that are used for comparison. On this restricted benchmark, `H1` has an average percentage gap of around 3% and an average computing time of 18 seconds, while the *hybrid* algorithm has a slightly larger average computing time (around 21 seconds), but the average percentage gap is almost halved (around 1.7%), thus showing the robustness of our design choices.

5.4.1 | Disconnected graphs

As previously mentioned, when the DSE problem is solved for a disconnected graph, there always exists an optimal solution corresponding to a connected subgraph. Hence, instead of running an algorithm on the whole graph, a possible strategy involves

TABLE 4 Average computing time and percentage gap for `H1` and hybrid algorithm on a selected subset of instances

	`H1`			Hybrid		
Instances	**# inst.**	***t***	**%gap**	**# inst.**	***t***	**%gap**
SELECTED	29	18 531	3.0322	27	20 864	1.6928

Note: All times are in milliseconds.

TABLE 5 Performance of greedy peeling and hybrid algorithm on disconnected graphs

	Original graph				**Biggest component**			
	Greedy peeling		**Hybrid algorithm**		**Greedy peeling**		**Hybrid algorithm**	
Instances	***t***	**% gap**	***t***	**% gap**	***t***	**% gap**	***t***	**% gap**
DISCONNECTED	41	5.5019	121	1.7912	34	9.8064	113	5.5338

Note: All times are in milliseconds.

executing it on each component separately. Although checking for connectivity in a given graph is an easy task, its practical difficulty depends on the size of the graph and on the way it is described. For example, when the graph is described as a list of edges, extracting and storing the connected components requires an increased memory usage, which can be a problem for huge graphs.

Analyzing instances in our benchmark, we found that most of them are connected graphs. As to the remaining disconnected ones, each of them typically has few large components with many vertices, and a very large number of small components that have very few vertices. Hence, in order to evaluate possible improvements obtained by considering each component separately, we tested both greedy peeling and hybrid algorithm where we: (i) first detect the connected components of the input graph by using the *depth-first search* algorithm, and then (ii) run the specific algorithm on the biggest connected component only.

Table 5 reports the statistics for the 12 instances in the medium bucket that are disconnected. It shows us the average computing time (in milliseconds) and the average percentage gap for both greedy peeling and hybrid algorithm executed on the original graph and on the biggest component in the said original graph. The table shows that component detection has limited impact in terms of computing time, as for both algorithms we observe a small reduction (a few milliseconds) of the average time. However, we also observe a non-negligible worsening of the solution quality (the average gap increases by around 4% in both cases). This is due to the fact that for one instance, the optimal subgraph is present in a smaller component and not in the biggest one. This analysis shows us that, in order to find the densest subgraph in a disconnected graph, we can not restrict ourselves to applying the algorithms on the biggest component of the graph. This leads to an increase in the time taken to find the densest subgraph as we have to run the algorithms on all the large components of the disconnected graph.

5.5 | Results on instances in the large and massive buckets

In this section, we present the results of our experiments on instances in the large and massive buckets. Based on the outcome of the results in the previous sections, we do not run the hybrid algorithm for those instances where the greedy solution (or the expanded subgraph) is almost as large as the original graph. In particular, we removed the graphs for which $\frac{|S^2|}{|V|} > 0.85$, namely, the instances in the series delaunay, hugebubbles, hugetrace, and hugetric, as well as instances 333SP, adaptive, AS365, channel-500x100x100-b050, M6, NACA0015, and NLR. Note that $|S^2|$ can be computed in negligible time before performing the expansion phase, simply scanning all the edges that are incident to vertices in the greedy solution.

Table 6 addresses the instances in the large bucket and shows that, similar to instances in the medium bucket, the hybrid algorithm consistently improves upon the density value produced by greedy peeling, frequently producing an optimal solution. The hybrid algorithm was able to find the optimal solution in 13 cases out of 21 instances, and in 12 out of these 13 cases it was faster than Goldberg's algorithm. As for the 8 instances that are not solved to optimality, the associated average gap is around 3.5%. The average gap over all the 21 instances is around 1.3%, much smaller than that of the greedy peeling algorithm, which is around 6.7%. As for the computing time, greedy peeling algorithm just takes around 1.3 seconds on average, while the hybrid algorithm takes 215 seconds on average. Goldberg's algorithm takes more than 1050 seconds on average for solving these instances to optimality.

In Table 7, we present the results of the three algorithms for the instances in the massive bucket in our benchmark. These graph instances were derived from real-life applications like gene networks (kmer series), road networks, social networks, and others. It can be immediately seen that Goldberg's algorithm fails for all the instances due to memory limitation. For these instances, greedy peeling finds a dense subgraph within 10 seconds on average, despite running on some graphs having tens of

TABLE 6 Results on instances in the large bucket

Graph properties			Greedy peeling		Hybrid				Goldberg's	
Instance	$\|V\|$	$\|E\|$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
as-Skitter	1 696 415	11 095 298	822	89.1810	2303	37 273	40 399	**89.4009**	388 513	89.4009
asia_osm	11 950 757	12 711 603	1342	1.7778	135	<1	1478	1.7778	703 145	**1.8513**
belgium_osm	1 441 295	1 549 970	185	1.6000	15	<1	200	1.6000	77 872	**1.6750**
com-LiveJournal	3 997 962	34 681 189	3129	190.9845	82	695	3907	**193.5136**	1 226 155	193.5136
com-Youtube	1 134 890	2 987 624	341	45.5778	1608	60 642	62 592	**45.5988**	157 970	45.5988
germany_osm	11 548 845	12 369 181	1734	1.6250	133	<1	1868	1.6667	784 833	**1.7500**
great-britain_osm	7 733 822	8 156 517	1039	1.8710	93	1	1134	**1.9583**	465 254	1.9583
italy_osm	6 686 493	7 013 978	743	1.6250	80	<1	824	1.6667	365 157	**1.7778**
netherlands_osm	2 216 688	2 441 238	298	1.6667	29	<1	328	**1.7143**	190 545	1.7143
packing-500x100x100-b050	2 145 852	17 488 243	640	8.5361	147 977	612 576	761 195	8.7361	2 931 714	**8.8078**
rgg_n_2_20_s0	1 048 576	6 891 620	415	11.1212	14	4	433	11.6250	276 226	**11.6346**
rgg_n_2_21_s0	2 097 152	14 487 995	930	9.3934	22	7	960	**11.9048**	667 290	11.9048
rgg_n_2_22_s0	4 194 304	30 359 198	1937	10.5503	58	25	2021	**12.550**	1 806 177	12.5500
road_central	14 081 816	16 933 413	3285	1.6002	179	20	3485	1.7750	6 231 763	**1.9029**
roadNet-CA	1 971 281	2 766 607	315	1.6743	48	233	597	**1.9677**	313 535	1.9677
roadNet-PA	1 090 920	1 541 898	177	1.6441	14	14	205	1.8571	234 657	**1.8783**
roadNet-TX	1 393 383	1 921 660	216	1.7656	17	7	241	**2.0769**	82 250	2.0769
venturiLevel3	4 026 819	8 054 237	672	2.0014	1 001 420	111 528	1 113 531	2.0613	351 929	**2.0613**
wikipedia-20051105	1 634 989	18 540 603	1561	126.5925	14 248	418 588	434 379	**127.0162**	872 899	127.0162
wikipedia-20060925	2 983 494	35 048 116	3643	138.7406	43 194	967 617	1 014 476	**140.5966**	1 919 124	140.5966
wikipedia-20061104	3 148 440	37 043 458	3862	140.5598	47 102	1 031 432	1 082 416	**141.6711**	2 063 044	141.6711

Note: All times are in milliseconds.

TABLE 7 Results on instances in the massive bucket

Graph properties			Greedy peeling		Hybrid				Goldberg's	
Instance	$\|V\|$	$\|E\|$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
europe_osm	50 912 018	54 054 660	6869	1.7047	640	26	7,535	**2.0000**	–	–
hollywood-2009	1 139 905	56 375 711	1895	**1104.0000**	14 712	199 468	216 076	1104.0000	–	–
kmer_U1a	67 716 231	69 389 281	26 907	4.0000	862	2	27 771	**4.0455**	–	–
kmer_V2a	55 042 369	58 608 800	20 570	6.9000	691	10	21 271	**7.0909**	–	–
rgg_n_2_23_s0	8 388 608	63 501 393	4072	11.0476	112	25	4210	**13.4000**	–	–
rgg_n_2_24_s0	16 777 216	132 557 200	8571	12.1220	237	15	8824	**13.7143**	–	–
road_usa	23 947 347	28 854 312	4545	1.5974	301	2	4849	**1.8462**	–	–
soc-orkut	4 847 571	106 349 209	9111	**206.9307**	509	–	–	–	–	–

Note: All times are in milliseconds.

millions of vertices and hundreds of millions of edges. The hybrid algorithm consistently improves upon the greedy solution for most instances, the only exceptions being hollywood-2009, where both algorithms give the same solution, and soc-orkut, for which the hybrid algorithm runs out of memory. Ignoring this last instance, the average computing time taken by the hybrid algorithm is around 41 seconds, and the average improvement produced by this algorithm over the greedy peeling algorithm is around 10%.

5.6 | Results on weighted instances

Finally, in this section, we address the weighted instances and report the associated results in Table 8.

The greedy peeling algorithm performs very well, and finds a provable optimal solution in 9 out of 16 cases; for instance, mawi_201512020000 it produces the same density value as the hybrid algorithm, but optimality of the solution cannot be confirmed as Goldberg's algorithm fails. The hybrid algorithm improves over the greedy solution in 5 of the 6 remaining instances, in 4 of these cases finding a provable optimal solution. On average, the greedy peeling algorithm takes 2 seconds,

TABLE 8 Results on weighted instances

Graph properties			Greedy peeling		Hybrid				Goldberg's	
Instance	**$\lvert V\rvert$**	**$\lvert E\rvert$**	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
ca2010	710 145	1 744 683	856	**6 234 021.0000**	21	1	881	6 234 021.0000	103 815	6 234 021.0000
cond-mat-2003	31 163	120 029	16	**17.6000**	1	<1	18	17.6000	3032	17.6000
cond-mat-2005	40 421	175 693	23	**23.0000**	1	<1	25	23.0000	4836	23.0000
fl2010	484 481	1 173 147	496	3 753 682.4620	15	20	538	**3 992 056.5380**	59 637	3 992 056.5380
ga2010	291 086	709 028	257	**3 929 610.0000**	8	4	275	3 929 610.0000	33 232	3 929 610.0000
human_gene1	22 283	12 323 680	311	**62.6766**	26 139	142 612	169 065	62.6766	275 234	62.6766
il2010	451 554	1 082 232	444	**5 508 363.6000**	13	1	489	5 508 363.6000	57 320	5 508 363.6000
mi2010	329 885	789 045	299	6 993 878.8460	10	3	322	7 370 921.5830	39 088	**7 390 000.2310**
mo2010	343 565	828 284	321	**1 666 117.5000**	10	<1	344	1 666 117.5000	41 163	1 666 117.5000
mawi_201512012345	18 571 154	19 020 160	9216	798 116.4286	560	120	9831	**927 951.0000**	–	–
mawi_201512020000	35 991 342	37 242 710	18 174	**1 770 103.0000**	1073	183	19 219	1 770 103.0000	–	–
mouse_gene	45 101	14 461 095	419	27.7563	34 115	217 095	251 631	**28.4702**	505 157	28.4702
ny2010	350 169	854 772	328	2 986 674.1110	11	2	347	**3 289 936.6250**	42 839	3 289 936.6250
oh2010	365 344	884 120	344	**3 826 971.8000**	11	4	360	3 826 971.8000	43 112	3 826 971.8000
pa2010	421 545	1 029 231	420	**3 202 713.0000**	12	<1	442	3 202 713.0000	52 870	3 202 713.0000
tx2010	914 231	2 228 136	1265	6 563 105.3330	27	2	1277	**6 630 141.8000**	120 507	6 630 141.8000

Note: All times are in milliseconds.

while the hybrid algorithm takes 28 seconds. On the other hand, Goldberg's algorithm fails to solve 2 instances and, for the remaining instances, it requires on average almost 98 seconds to find the optimal solution. By removing the two mawi instances, we see that the average percentage gap of the greedy peeling algorithm is around 1.72%, which is reduced to less than 0.02% by the hybrid algorithm.

6 | SUMMARY AND CONCLUSIONS

In this article, we have studied a non-linear graph optimization problem that requires one to determine the densest subgraph in a given graph. While some prior work mentioned that the so-called greedy peeling algorithm is good in practice, we concluded empirically that greedy peeling finds dense subgraphs which are close to the optimal subgraphs across a range of graph sizes. We provided a simple connected instance for which the greedy algorithm shows its worst-case performance. We introduced a new heuristic algorithm that combines this fast and effective greedy algorithm and an exact method from the literature. The extensive experiments done to measure the performance of this new heuristic suggest that, for a sizeable number of real-world instances, we can improve upon the solution provided by the greedy peeling algorithm using our new heuristic. We have presented an efficient implementation of the algorithms to solve both unweighted and weighted instances, with the aim of attacking instances of very large size, like those arising, for example, in social network applications. To the best of our knowledge, this is the most comprehensive computational study of the DSE problem involving instances with tens of millions of vertices and hundreds of millions of edges.

ACKNOWLEDGMENTS

This research was supported by "Mixed-Integer Non Linear Optimisation: Algorithms and Application" consortium, which has received funding from the European Union's EU Framework Programme for Research and Innovation Horizon 2020 under the Marie Skłodowska-Curie Actions Grant Agreement No 764759. The authors are grateful to two anonymous referee and the editors for their comments that helped to significantly improve the quality of the paper.

DATA AVAILABILITY STATEMENT

Data derived from public domain resources.

ORCID

Enrico Malaguti https://orcid.org/0000-0002-5884-9360

REFERENCES

[1] J. Abello, M. G. C. Resende, and S. Sudarsky, *Massive quasi-clique detection*, in *LATIN 2002: Theoretical Informatics*, Springer, Berlin/Heidelberg, Germany, 2002, 598–612.
[2] R. Andersen and K. Chellapilla, *Finding dense subgraphs with size bounds*, in *Algorithms and Models for the Web-Graph*, Springer, Berlin/Heidelberg, Germany, 2009, 25–37.
[3] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama, *Greedily finding a dense subgraph*, J. Algorithms **34** (2000), 203–221.
[4] B. Bahmani, R. Kumar, and S. Vassilvitskii, *Densest subgraph in streaming and MapReduce*, Proc. VLDB Endow. **5** (2012), 454–465.
[5] B. Balasundaram, S. Butenko, and I. V. Hicks, *Clique relaxations in social network analysis: The maximum k-plex problem*, Oper. Res. **59** (2011), 133–142.
[6] M. Charikar, *Greedy approximation algorithms for finding dense components in a graph*, in *Approximation Algorithms for Combinatorial Optimization*, K. Jansen and S. Khuller, Eds., Springer, Berlin/Heidelberg, Germany, 2000, 84–95.
[7] J. Chen and Y. Saad, *Dense subgraph extraction with application to community detection*, IEEE Trans. Knowl. Data Eng. **24** (2012), 1216–1230.
[8] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw. **38** (2011), 1:1–1:25.
[9] S. Fortunato, *Community detection in graphs*, Phys. Rep. **486** (2010), 75–174.
[10] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan, *A fast parametric maximum flow algorithm and applications*, SIAM J. Comput. **18** (1989), 30–55.
[11] A. Gionis and C. Tsourakakis, *Dense subgraph discovery (DSD)*, 2015, available at http://people.seas.harvard.edu/babis/dsd.pdf. Accessed October 16, 2019.
[12] A. V. Goldberg, *Finding a maximum density subgraph, Technical report*, University of California at Berkeley, Berkeley, CA, 1984.
[13] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. ACM **35** (1988), 921–940.
[14] M. Henzinger, A. Noe, C. Schulz, and D. Strash, *Practical minimum cut algorithms*, ACM J. Exp. Algorithmics **23** (2018), 1–8.
[15] S. Khuller and B. Saha, *On finding dense subgraphs*, in *Automata, Languages and Programming*, S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas, Eds., Springer, Berlin/Heidelberg, Germany, 2009, 597–608.
[16] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, *A survey of algorithms for dense subgraph discovery*, in *Managing and Mining Graph Data*, C. C. Aggarwal and H. Wang, Eds., Springer, Boston, MA, 2010, 303–336.
[17] J. C. Picard and M. Queyranne, *A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory*, Networks **12** (1982), 141–159.
[18] C. Tsourakakis, *A novel approach to finding near-cliques: The triangle-densest subgraph problem, Technical report*, ICERM, Brown University, Providence, RI, 2014.
[19] C. Tsourakakis, et al., *Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees*, Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, New York, NY, 2013, pp. 104–112.
[20] H. Yanagisawa and S. Hara, *Discounted average degree density metric and new algorithms for the densest subgraph problem*, Networks **71** (2018), 3–15.

How to cite this article: Gudapati NVC, Malaguti E, Monaci M. In search of dense subgraphs: How good is greedy peeling?. *Networks.* 2021;77:572–586. https://doi.org/10.1002/net.22034