Noname manuscript No.
(will be inserted by the editor)

Efficient and Effective Community Search

Nicola Barbieri · Francesco Bonchi · Edoardo Galimberti · Francesco Gullo

the date of receipt and acceptance should be inserted later

Abstract Community search is the problem of finding a good community for a given set of query vertices. One of the most studied formulations of community search asks for a connected subgraph that contains all query vertices and maximizes the minimum degree.

All existing approaches to min-degree-based community search suffer from limitations concerning efficiency, as they need to visit (large part of) the whole input graph, as well as accuracy, as they output communities quite large and not really cohesive. Moreover, some existing methods lack generality: they handle only single-vertex queries, find communities that are not optimal in terms of minimum degree, and/or require input parameters.

In this work we advance the state of the art on community search by proposing a novel method that overcomes all these limitations: it is in general more efficient and effective—one/two orders of magnitude on average, it can handle multiple query vertices, it yields optimal communities, and it is parameter-free. These properties are confirmed by an extensive experimental analysis performed on various real-world graphs.

1 Introduction

Given a graph and a set of query vertices, the *community search* problem aims at finding a cohesive subgraph containing the query vertices. Community search has recently attracted a great deal of attention, fueled by applications such as advertising and viral marketing, content recommendation, and team formation [5].

A number of criteria to assess the goodness of a community have been proposed, such as random-walk-based measures [18, 11], or density-based measures [4, 10, 14, 19]. Sozio and Gionis [17] introduce the term "community-search problem", which they formalize from a combinatorial-optimization perspective as the problem of finding *a connected subgraph that contains all query vertices and maximizes the minimum degree*. Minimum degree has in fact been shown to be an effective way of measuring the goodness of a community [2, 5].

N. Barbieri, F. Bonchi, F. Gullo
Yahoo Labs - Barcelona, Spain
E-mail: {barbieri,bonchi}@yahoo-inc.com, gullof@acm.org

E. Galimberti
Politecnico di Milano - Italy
E-mail: edoardo.galimberti@mail.polimi.it

Table 1 Comparison of the proposed method to the state of the art. Our method exhibits all desired features: it is more efficient and accurate than all existing methods, can handle multiple query vertices, produces communities with optimal minimum degree, and is parameter-free.

	efficiency	*quality*	*multiple query vertices*	*optimality*	*parameter-free*
Global Search [17]	+	+	yes	yes	yes
Constrained Global Search [17]	+	++	yes	no	no
Local Search [5]	++	++	no	yes	yes
Our method	+++	+++	yes	yes	yes

In this work we focus on the min-degree-based formulation of community search and propose a novel method that overcomes the limitations of existing approaches.

1.1 Limitations of existing community-search methods

Efficiency. Sozio and Gionis [17] show that the community-search problem based on min-degree can be solved in linear time in the size of the input graph. Specifically, the algorithm by Sozio and Gionis is a *global search* (GS) method that needs to visit the entire input graph, thus being computationally expensive on large graphs. A more efficient solution has been recently proposed by Cui *et al.* [5]. Their *local search* (LS) method however works only for single-vertex queries.

In this work we propose an approach that improves upon the efficiency of those methods, including the method by Cui *et al.* on the special case of a single query vertex. We do so while still keeping generality, as our method is able to handle multiple query vertices.

Effectiveness. Community search admits several (optimal) solutions. As remarked by Sozio and Gionis [17], a limitation of their algorithm is that, among the many ones available, it tends to find quite large solutions. This negatively affects accuracy of the discovered communities. In fact, a too large solution is likely to contain different optimal solutions that, if taken individually, would correspond to better communities, as they are inherently more cohesive and denser. Also, a large community is likely to contain redundant/outlying vertices.

Sozio and Gionis attempt to solve this issue by introducing a *constrained* version of the community-search problem, where an upper bound on the size of the output community is admitted. They show that this version of the problem is $\mathbf{NP}$-hard and devise heuristics to solve it. This *constrained global search* (K-GS) method has however a number of limitations: it further degrades efficiency with respect to standard global search, it is not guaranteed to provide communities with optimal minimum degree, and it introduces a parameter, i.e., the upper bound on the community size, which is difficult to set, as it represents only a soft constraint (the output communities may have size much larger than the bound specified).

In the approach we propose in this work we do not explicitly constrain the size of the communities, rather we aim at finding the smallest-sized solution among all the optimal ones. This way, we are able to produce communities that are on average orders of magnitude more effective, i.e., smaller and denser, than global and local search methods. Moreover, unlike constrained global search, our approach achieves high efficiency, identifies optimal communities, and requires no parameters.

1.2 Our approach and contributions

The proposed approach is based on exploiting precomputed information, which is retrieved at query time and further processed to produce the final solution. The precomputation relies

on the graph-theory notion of *core decomposition* of a graph [16]. Given a graph $G = (V, E)$ and a set of query vertices $Q \subseteq V$, let k be the largest integer such that a connected component of the k-core of G contains Q. It holds that all the optimal solutions to the community-search problem for the given query Q, are contained in such k-core. Thanks to this property, our approach starts with retrieving from the precomputed information (i.e., the core decomposition) a subgraph H^* that is itself an optimal solution to the query issued. However, as H^* is also guaranteed to contain *all* other optimal solutions, it may easily be not so good in terms of quality (i.e., size and cohesiveness). For this reason, we formulate the problem of finding the smallest subgraph H^*_{min} of H^* that satisfies the community-search requirements. We show that the problem is $\mathbf{NP}$-hard and devise a heuristic that satisfies both efficiency and effectiveness requirements.

Our method addresses all the weaknesses of the state of the art. Apart from being generally faster and more effective, it does not suffer from any limitation affecting existing approaches: it is in fact, at the same time, general (it handles multiple query vertices), optimal (it finds communities with optimal minimum degree), and parameter-free. These characteristics are better highlighted in Table 1.

In summary, our contributions are as follows.

- We advance the state of the art on community search by aiming at improving efficiency, effectiveness, and generality of existing methods.
- To address these limitations, we tackle the problem of finding the smallest-size solution to community search and devise an approach that exploits the core decomposition of the graph as precomputed information.
- We propose two different ways of organizing/storing the core decomposition: the first one guarantees faster retrieval of the information needed at query time, while the other one requires less storage space.
- We devise a method for the online query-processing phase that relies on two steps. The first step employs a local greedy strategy to further reduce the subgraph returned as precomputed information. The second step takes this reduced subgraph and extracts the final solution by aiming at the connection and minimum-degree constraints in a sequential fashion.
- An extensive evaluation on a variety of real-world graphs confirms that the proposed method is generally faster, as well as able to produce higher-quality solutions: our communities are on average much smaller in size and denser.

2 Preliminaries

In this section we provide the formal statement of the COMMUNITY SEARCH problem, an overview of existing methods, and a discussion about the notion of core decomposition, which is at the basis of the proposed method.

2.1 Problem definition

Let $G = (V, E)$ be an undirected graph, where V ($|V| = n$) is the set of vertices and $E \subseteq V \times V$ ($|E| = m$) is the set of edges. For a subset of vertices $H \subseteq V$, we denote by $G[H]$ the subgraph of G induced by H, i.e., $G[H] = (H, E[H])$, where $E[H] = \{(u, v) \in$

$E \mid u \in H, v \in H\}$. Finally, we denote by $\delta_H(u)$ the degree of u in $G[H]$, and $\mu(H)$ the minimum degree of a vertex in $G[H]$, i.e., $\mu(H) = \min_{u \in H} \delta_H(u)$.

The COMMUNITY SEARCH problem (CSP) is defined as follows [17,5]:

Problem 1 (COMMUNITY SEARCH) Given a graph $G = (V, E)$ and a set of query vertices $Q \subseteq V$, find

$$H^* = \underset{\substack{Q \subseteq H \subseteq V, \\ G[H] \text{is connected}}}{\arg\max} \quad \mu(H). \qquad \square$$

Due to the connectivity constraint, the state of the art on community search assumes query vertices belonging to the same connected component of the input graph [18,11,4, 17,5,10,14,19]. A simple extension for the case where query vertices come from different connected components is to run community search separately for each connected component and take the union of the various output subgraphs as ultimate solution.

2.2 State of the art

Unconstrained Community Search. Sozio and Gionis [17] show that Problem 1 can be solved by a simple greedy algorithm taking linear time in the size of the input graph. We refer to this approach as Global Search (GS).[1] This algorithm iteratively removes a vertex having the minimum degree along with all its incident edges, and it stops when the next vertex to be removed corresponds to a query vertex. This process generates a set of subgraphs. Among these subgraphs, the one having the maximum minimum degree, and such that all query vertices are connected is returned as output.

Size-bounded Community Search. Sozio and Gionis [17] notice a limitation of the min-degree-based CSP formulation: the output communities tend to be large in size. For this purpose, they introduce the size-bounded version of CSP, which includes an upper bound K on the size of the output community. They show that this version of CSP becomes **NP**-hard and propose heuristics to solve it. We refer to this method as K-Global Search (K-GS).

The attempt of mitigating the size issue comes at a price of limited efficiency and accuracy. Indeed, the K-GS method relies on the maximum distance between a query vertex and every other vertex in the graph, which requires a number of costly shortest-path-distance computations. Also, this approach does not guarantee optimality: the minimum degree of a solution to size-bounded CSP may be arbitrarily smaller than the optimal minimum degree of the corresponding unconstrained CSP formulation. Finally, K-GS requires as input parameter K, i.e., the upper bound on the output community size, which is difficult to set as it represents only a soft constraint: the communities returned by K-GS are indeed not guaranteed to have size no larger than K. As we show in our evaluation in Section 5, this gap increases with the number of query vertices and may be quite consistent.

Local Community Search. Cui *et al.* [5] propose a local-search algorithm to improve the efficiency of the GS algorithm. This algorithm, which we refer to as Local Search (LS), iteratively expands the neighborhood of the (unique) query vertex, until a subgraph that is guaranteed to contain an optimal solution has been built. Then, this subgraph is used as a reduced version of the input graph to retrieve the optimal solution. The worst-case time complexity of the LS algorithm is still linear in the size of the whole input graph, but LS has been shown to achieve better efficiency than GS in practice [5]. Nevertheless, a severe limitation of LS is that it works only when a single query vertex is provided as input.

[1] To be precise, the GS algorithm runs in $\mathcal{O}(m \times \alpha(n))$ time, where $\alpha(\cdot)$ denotes the inverse Ackermann function.

2.3 Core decomposition

The k-*core* (or *core* of order k) of a graph $G = (V, E)$ is defined as the maximal subgraph in which every vertex is connected to at least k other vertices within that subgraph. In the following we slightly abuse the notation and identify a k-core with its vertex set, which we denote by C_k. It is easy to see that the order of a core corresponds to the minimum degree of a vertex in that core, i.e., $k = \mu(C_k)$.

The set $\mathbf{C} = \{C_k\}_{k=1}^{k^*}$ forms the *core decomposition* of G [16]. The *core number* (or *core index*) of a vertex $u \in V$, denoted $c(u)$, is the highest order of a core that contains u: $c(u) = \max\{k \in [0..k^*] \mid u \in C_k\}$. All cores are nested into each other: $V = C_1 \supseteq C_2 \supseteq \cdots \supseteq C_{k^*}$, and the "difference" between two consecutive cores, i.e., the set $S_k = C_k \setminus C_{k+1}$, is usually referred to as k-*shell*. The set of all k-shells therefore forms a partition of the vertex set V. Note that a k-core (or a k-shell) does not necessarily induce a connected subgraph, and that the k-cores are not necessarily all distinct, i.e., it may happen that, for some k, $C_k = C_{k+1}$ (and the corresponding k-shell $S_k = \emptyset$).

Batagelj and Zaveršnik [1] show how to compute the core decomposition of a graph G in linear time. The algorithm iteratively removes the smallest-degree vertex and sets the core number of the removed vertex accordingly.

3 Precomputation

Here we present our proposal to solve CSP effectively and efficiently. Our approach is composed of two phases. In the *preprocessing* phase (which we discuss in the remainder of this section), the input graph is processed offline (*una tantum*), in order to compute and store information that can profitably be reused when a query is issued. The *query* processing phase (which we present in Section 4), is in turn divided into two subphases: a *retrieval* phase (Section 4.1), where the proper information computed/stored during the preprocessing is retrieved, and an *online processing* phase (Section 4.2), where the information retrieved is further processed in order to obtain the ultimate answer to the query.

The preprocessing phase of our approach is based on an interesting relationship between core decomposition and CSP. This relationship consists of two main results: the highest-order (i.e., smallest-sized) core containing all query vertices and such that all query vertices are connected (i) is a solution to CSP, and (ii) contains *all* the solutions to CSP. Similar results are also shown in [5], but they hold only for the special case where queries are composed of only one vertex. In the following theorem we provide evidence of these results for the general case where one can have multiple query vertices.

Theorem 1 *Given a graph $G = (V, E)$, its core decomposition $\mathbf{C} = \{C_1, \ldots, C_{k^*}\}$, and query vertices $Q \subseteq V$, let C_Q^* be the highest-order (i.e., smallest-sized) core in $\mathbf{C}$ such that every $q \in Q$ belongs to the same connected component of C_Q^*. It holds that:*

1. *The connected component of C_Q^* that contains Q is a solution to* CSP*;*
2. *The connected component of C_Q^* containing Q contains all solutions to* CSP.

Proof We prove both statements by contradiction. As far as the first statement, let X denote the connected component of C_Q^* that contains all query vertices and let k denote the minimum degree of a vertex in X. Assume that X is not a solution to CSP, i.e., assume that there exists another subgraph $Y \neq X$ of G that is connected, contains all query vertices, and has minimum degree $> k$. By the definition of k-core, this would imply that Y is (part of) a core

Algorithm 1 BuildCoreStruct

Input: A graph $G = (V, E)$; the core decomposition $\mathbf{C} = \{C_1, \ldots, C_{k^*}\}$ of G.
Output: The core index $c(u), \forall u \in V$; The set $\mathcal{H}_k$ of connected components of core $C_k, \forall k \in [0..k^*]$.

```
1: c(u) ← max{k ∈ [0..k*] | u ∈ C_k}, ∀u ∈ V
2: H_{k*} ←connectedComponents(G, C_{k*})
3: for all k = k*−1, ..., 1 s.t. C_k ≠ C_{k+1} do
4:     H' ← H_{k+1}, X ← C_{k+1}, S_k ← C_k \ C_{k+1}
5:     for all u ∈ S_k do
6:         T ← {{u}}
7:         for all v ∈ X | (u, v) ∈ E do
8:             T ← T ∪ {H ∈ H' | v ∈ H}
9:         X ← X ∪ {u}
10:        H* ← ⋃_{T∈T} T
11:        H' ← H' \ T ∪ {H*}
12:    H_k ← H'
```

of order higher than k. This contradicts the hypothesis that C^*_Q is the highest-order core in $\mathbf{C}$ such that every $q \in Q$ belongs to the same connected component of C^*_Q.

Concerning the second statement, assume that there exists a solution Z to CSP that is not contained in X. This implies that (i) the minimum degree of a vertex in Z is equal to k and (ii) the minimum degree of a vertex in $X \cup Z$ is equal to k as well, by the optimality of X. It follows that $X \cup Z$ should correspond to a connected component of the core C^*_Q, but this is in contrast with the maximality condition of the definition of k-core. □

Theorem 1 forms the basis of our preprocessing phase. The idea is to compute the core decomposition of the input graph and organize the k-cores in order to easily retrieve the k-core that contains all solutions to CSP whenever a query is issued. We devise two structures for organizing the core decomposition: the first one (CoreStruct) stores the k-cores in order to guarantee faster retrieval, while the second one (ShellStruct) stores only the k-shells and, as such, minimizes the replication of information, thus requiring less storage space. In the following we report the details of CoreStruct and ShellStruct.

3.1 The CoreStruct structure

Structure. Given the core decomposition $\mathbf{C} = \{C_1, \ldots, C_{k^*}\}$ of the input graph G, the CoreStruct method keeps the following information:

- For all vertices $u \in V$, the core index $c(u)$ of u, which is defined as the index of the highest-order core containing u: $c(u) = \max\{k \in [0..k^*] \mid u \in C_k\}$.
- For all cores $C_k \in \mathbf{C}$, the set $\mathcal{H}_k$ of connected components of C_k, with $\bigcup_{H \in \mathcal{H}_k} H = C_k$.

As pointed out in Section 2.3, two consecutive cores may be non-distinct (i.e., $C_k = C_{k+1}$ for some k). In this case, one can clearly avoid to store multiple copies of the same core, and, at the same time, avoid to replicate the information at the basis of CoreStruct. This remains true also for the other preprocessing method (ShellStruct) that we discuss in the next subsection. Moreover, we remark that the information required by CoreStruct is stored in proper data structures that ensure constant-time access to all the main pieces of information that are needed for the subsequent retrieval phase, i.e., the core index of a vertex, the connected component of a vertex in a core, the set of vertices in a (connected component of a) core.

Building procedure. Algorithm 1 shows how to compute CoreStruct efficiently. The algorithm takes as input the graph G and its core decomposition, and iteratively processes the

k-cores starting from the highest-order one (i.e., C_{k^*}). The connected components of each core C_k are efficiently computed by looking at (i) the neighborhood of every vertex in the shell $S_k = C_k \setminus C_{k+1}$ and (ii) the connected components of the core C_{k+1}, which have already been computed in the previous iteration (Lines 3-15). Let X denote the set of vertices in the core C_{k+1} along with the vertices in S_k that have already been processed. For every vertex $u \in S_k$, the connected components of C_{k+1} of all neighbors of u that are in X are added to the set $\mathcal{T}$ (Lines 6-9). Once all the neighbors of the vertex u have been computed, $\mathcal{T}$ contains the connected components that can be merged to form a unique component to be added to $\mathcal{H}'$ (Lines 11-12).

Complexity. In analyzing the time complexity of Algorithm 1 we assume that the mappings vertex↔shell-index and vertex↔core-index, which are computed during the core decomposition, can be accessed in constant time. Computing the core index of every vertex and the connected components of C_{k^*} (Lines 1-2) take $\mathcal{O}(n)$ and $\mathcal{O}(n+m)$ time, respectively. The main cycle in Lines 3-15 processes every vertex u in the graph exactly once. For each vertex u, the most expensive operations in this cycle are: (i) visiting all its neighbors (Lines 7-9), which takes $\mathcal{O}(m)$ overall time, and (ii) merging the connected components in the set $\mathcal{T}$ (Line 11), which takes $\mathcal{O}(hn)$ overall time in the worst case (with h being the number of distinct k-cores). The overall running time of Algorithm 1 is hence $\mathcal{O}(hn+m)$. We remark that $h \ll n$ in practice (e.g., up to five orders of magnitude less, according to the real-life graphs used in our experiments).

The CoreStruct method needs to store all distinct cores, thus requiring $\mathcal{O}(hn)$ space.

3.2 The ShellStruct structure

CoreStruct can easily be computed and, as we will see in Section 4, it enables a very efficient retrieval phase (logarithmic in the number of cores). These advantages come with a cost: for very large graphs, its $\mathcal{O}(hn)$ space complexity might be prohibitive. For instance, for a graph of 50M vertices and 1k distinct cores, assuming 32 bits for representing a vertex, CoreStruct requires around 190GB, thus likely preventing the possibility of loading it in main memory.

Structure. The idea behind ShellStruct is to avoid replication by storing shells instead of cores. Precisely, ShellStruct still keeps the information about the core index $c(u)$ of every vertex $u \in V$ like CoreStruct, but it does not duplicate every vertex u across all cores that are nested into the core corresponding to $c(u)$. Every vertex u is now stored only once, in correspondence of the appropriate shell.

To allow the retrieval of the information required at query time, the connected components of each core are organized in a tree-like structure having k^* levels.[2] Each level k keeps trace of the information about the k-th shell/core, and the root of the tree corresponds to the shell/core 1 (the whole input graph G). Each tree node at the k-th level (shell) of the tree corresponds to a connected component of the corresponding core C_k. For each tree node H we keep a twofold information: (i) the set of "children" connected components, i.e., the connected components (if any) of the core C_{k+1} that are merged together to form H in core C_k (this determines the tree-like structure), and (ii) the set of all vertices that are part of the k-th shell and belong to connected component H.

[2] In the implementation all levels corresponding to non-distinct cores can be omitted; thus, the actual number of levels of the tree is h.

As far as implementation, the tree is stored in proper data structures that allow to retrieve in constant time (i) the father/children of a tree node, (ii) the content (in terms of vertices of the graph) of every connected component in a core, and, (iii) the connected component to which every vertex in the graph belongs to.

Building procedure and complexity. The procedure to compute ShellStruct is similar to the one used for CoreStruct (Algorithm 1). The only difference is in Line 11: rather than merging the connected components in $\mathcal{T}$ in a single connected component H^*, here we build the tree-like structure by adding a node corresponding to H^* and by linking all connected components in $\mathcal{T}$ as children of H^*. This implementation resembles the union operation of the well-known union-find method. Hence, the time complexity of building ShellStruct is the same as CoreStruct, that is $\mathcal{O}(hn + m)$. However, the time needed for building ShellStruct is expected to be higher than CoreStruct in practice, due to its more complex structure (i.e., the connected-component tree).

For what concerns space, each vertex in the input graph is stored once, thus requiring $\mathcal{O}(n)$ space. Moreover, the space needed for the connected-component tree is $\mathcal{O}(hM)$, where M denotes the maximum number of connected components in a core. However, as both M and h are usually several orders of magnitude less than n, the space of the connected-component tree is in practice negligible compared to the space needed to store vertices.

4 Online query processing

In this section we describe in detail the query phase of our approach. This phase consists of two steps: retrieval of the information computed during preprocessing, and an online processing of this information to provide the final query answer.

4.1 Retrieval

Given a set of query vertices Q, the retrieval phase aims at finding the (connected component of the) highest-order core such that all query vertices in Q are part of that core and are connected (Theorem 1).

Retrieval from CoreStruct. Let $c_{min}(Q)$ be the minimum core index of a query vertex in Q, i.e., $c_{min}(Q) = \min\{c(u) \mid u \in Q\}$. In the retrieval phase, we determine the largest index k in the range $[0, c_{min}(Q)]$ such that there exists a connected component H of core C_k containing Q, that is

$$k_{max} = \max\{k \in [0, c_{min}(Q)] \mid \exists H \in C_k : Q \subseteq H\},$$

and output the corresponding connected component H.

The information stored in CoreStruct allows us to check the condition $\exists H \in C_k : Q \subseteq H$ in constant time for every query vertex, thus requiring $\mathcal{O}(|Q|)$ overall time for a query set Q. It is easy to see that this condition follows a monotonic trend; thus, we can perform binary search to find the desired index k_{max}. As the size of the range $[0, c_{min}(Q)]$ is guaranteed to be $\leq h$ (as explained above, we store information only for the h distinct cores), the overall time complexity of the retrieval phase from CoreStruct is $\mathcal{O}(|Q| \log h)$.

Retrieval from ShellStruct. The retrieval of the desired (connected component of a) core H^* from ShellStruct is more complex: the content of H^* is not directly available like in CoreStruct, but it needs to be reconstructed from the connected-component tree. The details

Algorithm 2 RetrievalShellStruct

Input: Information from ShellStruct; a set of query vertices Q.
Output: A set of vertices $H^* \supseteq Q$.
1: $k \leftarrow \max\{c(u) \mid u \in Q\}$
2: $\mathcal{H} \leftarrow \{$connected components of core C_k containing a vertex from $Q\}$
3: $H^* \leftarrow \bigcup_{H \in \mathcal{H}} vertices(H)$
4: **while** $|\mathcal{H}| \neq 1 \wedge Q \neq \emptyset$ **do**
5: $\quad Q^- \leftarrow \{u \in Q \mid u \in S_k\}$
6: $\quad \mathcal{H}' \leftarrow \{parent(H) \mid H \in \mathcal{H}\} \cup \{$connected components of core i containing a vertex from $Q^-\}$
7: $\quad H^* \leftarrow H^* \cup \bigcup_{H \in \mathcal{H}'} vertices(H)$
8: $\quad Q \leftarrow Q \setminus Q^-,\ \mathcal{H} \leftarrow \mathcal{H}',\ k \leftarrow k-1$

of the procedure are reported as Algorithm 2. In this case, we need to visit all the levels of the tree, starting from level $c_{max}(Q) = \max\{c(u) \mid u \in Q\}$, until a connected component containing all query vertices has been encountered. Since binary search is no longer possible and the content of H^* needs to be reconstructed, the time complexity of the retrieval phase from ShellStruct is $\mathcal{O}(|Q|h+|H^*|)$, where $|H^*|$ is $\mathcal{O}(n)$. The time complexity can be lowered up to $\mathcal{O}(|Q|+|H^*|)$ by using a suitable lowest-common-ancestor (LCA) data structure to store the connected-component tree (e.g., by preprocessing the connected-component tree with the well-known Tarjan's offline lowest common ancestors algorithm which allows constant-time online retrieval [8]). However, as $|H^*|$ is typically the dominant term in the overall time complexity, the efficiency gain would not be really significant in practice.

As anticipated above, CoreStruct compensates its larger storage space with a faster retrieval phase, while ShellStruct offers less space at a price of slower retrieval.

4.2 Online processing

4.2.1 The Minimum Community Search problem

The retrieval phase described above finds the set H^* containing all the solutions to CSP for a given query Q. The goal of the online processing phase is to further refine H^* to extract a solution that is as small as possible. This corresponds to solving the following problem:

Problem 2 (MINIMUM COMMUNITY SEARCH (MIN-CSP)) Given a graph $G = (V, E)$, a set of query vertices $Q \subseteq V$, let $H^* \subseteq V$ be the subgraph of G containing all the solutions to CSP for Q. Find

$$H^*_{min} = \underset{\substack{Q \subseteq H \subseteq H^*: \\ G[H] \text{is connected}, \\ \mu(H) \geq \mu(H^*)}}{\arg\min} |H|. \qquad \square$$

The above problem has been mentioned by Cui *et al.* in their work [5], but they do not study it further (no algorithm is proposed). They just provide a proof of **NP**-hardness by a reduction from MAX CLIQUE. Their proof however holds only for the special case $|Q| = 1$, and, as such, it does not exclude that the problem can admit polynomial-time solutions for the more general case $|Q| > 1$. In the following we show that this is not the case: the problem remains **NP**-hard even for the case $|Q| > 1$. We formally state this result in the next Theorem 2, by resorting to the well-known STEINER TREE problem: given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of terminal vertices $T \subseteq \mathcal{V}$, find a connected subgraph $\mathcal{G}'$ of $\mathcal{G}$ containing all terminal vertices and having minimum number of edges. This problem is **NP**-hard and the fastest existing algorithm with provable guarantees corresponds to the

$(2-2/|Q|)$-approximation algorithm by Kou *et al.* [12], which takes time linear in the number of edges of the input graph by using the Mehlhorn's implementation [15].

Theorem 2 MIN-CSP *is* $\mathbf{NP}$*-hard for query sets* Q *of size* $|Q| > 1$.

Proof We reduce the STEINER TREE problem to MIN-CSP. It is well-known that the optimal solution to any instance of STEINER TREE is actually a tree. As finding a tree with minimum number of edges corresponds to finding a tree with minimum number of vertices, STEINER TREE can easily be reformulated as the problem of finding the minimum number of vertices that make all terminals connected.

Given an instance of STEINER TREE, one can construct an instance of Problem 2 by building a graph G equal to $\mathcal{G}$ and with an additional vertex x connected to (only) one other vertex in $\mathcal{V}$, and setting $Q = T$ and $H^* = \mathcal{V} \cup \{x\}$. In this instance $\mu(H^*) = 1$, which means that the minimum-degree constraint can be ignored; thus solving this instance of Problem 2 corresponds to finding a minimum number of vertices to to make the vertices in Q connected, which is the only one required by STEINER TREE. This implies that any set of vertices H is a solution to this instance of Problem 2 if and only if any spanning tree of the subgraph induced by H. □

4.2.2 The GreedyConnection algorithm

Given that MIN-CSP is $\mathbf{NP}$-hard we devise an efficient heuristic to solve it, which is based on the following idea.

A solution to MIN-CSP needs to satisfy two constraints: (a) query vertices are connected, and (b) the solution has minimum degree no less than $\mu(H^*)$. Among all possible solutions satisfying these constraints, the goal is to output the one with smallest size. Our main intuition in designing an effective heuristic to MIN-CSP is to look at constraints (a) and (b) one at a time. In particular, we aim at fist finding a solution that satisfy constraint (a) only, and then refine this solution in order to make it satisfying constraint (b) too. The motivation is that in real-world graphs with power-law-like degree distribution the minimum degree of a community is typically small (i.e., in the order of tens or even less); thus, it is likely that any solution that satisfies (only) constraint (a) needs just a very few additional vertices to satisfy constraint (b) too. This improves the chance of having an overall solution with size as small as possible, as required by MIN-CSP.

This strategy has however an issue. As shown in more detail later, aiming at a solution that makes query vertices connected (constraint (a)) with as few additional vertices as possible corresponds to solve the STEINER TREE problem. As said above, this problem is $\mathbf{NP}$-hard and the fastest existing approximation/heuristic algorithms need to visit all the edges of the input graph at least once [15]. In our context this corresponds to visiting all the edges in the entire subgraph H^*. As H^* may be in practice quite large, this strategy would easily result in poor efficiency.

To overcome this issue, we devise a two-step method, which we call GreedyConnection. In the first step we attempt to reduce the size of H^* as much as possible (while still keeping constraints (a) and (b) satisfied) by an efficient local greedy procedure that does not need to visit the entire H^*. We call this step Greedy. Then, in the second step, called Connection, we run the aforementioned STEINER TREE-based strategy on the subgraph outputted by Greedy, which, thanks to its reduced size with respect to the original H^*, can now be handled without affecting efficiency too much. In the following we provide the details of each of these steps.

Algorithm 3 Greedy

Input: A graph $G = (V, E)$; a set of query vertices $Q \subseteq V$; a set $H^* \subseteq V$ containing all the solutions to CSP for Q.

Output: A set of vertices $H^*_{min} \subseteq H^*$.

1: $H^*_{min} \leftarrow Q,\ \mathcal{A} \leftarrow \emptyset,\ \mu^* \leftarrow \mu(H^*),\ P \leftarrow \emptyset$
2: add all $q \in Q$ to P with priority $+\infty$
3: **while** $|\mathcal{A}| \neq 1 \ \wedge\ \mu(H^*_{min}) < \mu^*$ **do**
4: $\quad u \leftarrow P.poll()$
5: $\quad H^*_{min} \leftarrow H^*_{min} \cup \{u\}$
6: $\quad \alpha(u) \leftarrow |neigh(u, W^*_{min})|$
7: $\quad$ **for all** $v \in neigh(u, H^*) \setminus H^*_{min} \setminus P$ **do**
8: $\quad\quad$ compute (from scratch) $p'(v), p''_+(v), p''_-(v), p''(v), p(v)$ [Eq.(1)–(5)]
9: $\quad\quad$ add v to P with priority $p(v)$
10: $\quad \mathcal{A}' \leftarrow \{\{u\}\}$
11: $\quad$ **for all** $v \in neigh(u, H^*_{min})$ **do**
12: $\quad\quad \alpha(v) \leftarrow \alpha(v) + 1$
13: $\quad\quad$ **if** $\alpha(v) = \mu^*$ **then**
14: $\quad\quad\quad$ **for all** $w \in neigh(v, H^*) \cap P$ **do**
15: $\quad\quad\quad\quad p''_+(w) \leftarrow p''_+(w) - 1,\ p''(w) \leftarrow p''_+(w) - p''_-(w)$
16: $\quad\quad\quad\quad p(w) = \langle p'(w), p''(w) \rangle$
17: $\quad\quad \mathcal{A}' \leftarrow \mathcal{A}' \cup \{A \in \mathcal{A} \mid v \in A\}$
18: $\quad A^* \leftarrow \bigcup_{A \in \mathcal{A}'} A$
19: $\quad \mathcal{A} \leftarrow \mathcal{A} \setminus \mathcal{A}' \cup \{A^*\}$
20: $\quad$ **for all** $A \in \mathcal{A}'$ **do**
21: $\quad\quad$ **for all** $w \in \bigcup_{v \in A}(neigh(v, H^*) \cap P)$ **do**
22: $\quad\quad\quad p'(w) \leftarrow p'(w) - 1$
23: $\quad$ **for all** $w \in \bigcup_{v \in A^*}(neigh(v, H^*) \cap P)$ **do**
24: $\quad\quad p'(w) \leftarrow p'(w) + 1$
25: $\quad\quad$ **if** $u \in neigh(w, H^*)$ **then**
26: $\quad\quad\quad p''_+(w) \leftarrow p''_+(w) + \mathbb{1}[\alpha(u) < \mu^*]$
27: $\quad\quad\quad p''_-(w) \leftarrow \max\{0, p''_-(w) - 1\}$
28: $\quad\quad\quad p''(w) \leftarrow p''_+(w) - p''_-(w)$
29: $\quad\quad p(w) = \langle p'(w), p''(w) \rangle$

The Greedy *step*

This is a greedy bottom-up method that starts from a solution H^*_{min} containing the query vertices Q only, and adds vertices from $H^* \setminus H^*_{min}$ until H^*_{min} has become an optimal solution to CSP for Q, i.e., until (*i*) the subgraph $G[H^*_{min}]$ induced by H^*_{min} is connected, and (*ii*) the minimum degree of a vertex in $G[H^*_{min}]$ is equal to the minimum degree of $G[H^*]$, i.e., $\mu(H^*_{min}) = \mu(H^*)$.

Greedy selection. To select the vertex to be added to H^*_{min} at each step, Greedy takes into account how well a vertex contributes to the two requirements of connection and minimum degree. More precisely, for every vertex $u \in H^* \setminus H^*_{min}$, we define its *connection* score $p'(u)$ as the number of distinct connected components in $G[H^*_{min}]$ that would be merged by adding v to H^*_{min}. Formally, given a function cc that maps each vertex in H^*_{min} to its corresponding connected component in $G[H^*_{min}]$, and denoting by $neigh(u, X)$ the neighbors of u in $G[X]$, the connection score $p'(u)$ is defined as:

$$p'(u) = |\{cc(v) \mid v \in neigh(u, H^*_{min})\}| - 1. \tag{1}$$

Let μ^* denote the minimum degree of a vertex in H^*, i.e., $\mu^* = \mu(H^*)$. The minimum-degree condition asks for all vertices in H^*_{min} to have degree $\geq \mu^*$. The *minimum-degree* score $p''(u)$ of a vertex u considers two aspects. First, as a gain effect, the inclusion of a vertex u into the current solution H^*_{min} could contribute to increase the degree of some vertices in H^*_{min} and make it closer to the desired μ^*. However, as a penalty effect, the candidate vertex u will need to reach the desired degree μ^*.

Thus, the minimum-degree score $p''(u)$ can be defined as:

$$p''(u) = p''_{+}(u) - p''_{-}(u), \tag{2}$$

where $p''_{+}(u)$ (gain effect) is the number of neighbors of u in $G[H^*_{min}]$ having degree still less than μ^*:

$$p''_{+}(u) = |\{v \in neigh(u, H^*_{min}) \mid |neigh(v, H^*_{min})| < \mu^*\}|, \tag{3}$$

and $p''_{-}(u)$ (penalty effect) is the number of u's neighbors required to be added to H^*_{min} for having its degree at least μ^*:

$$p''_{-}(u) = \max\{0,\ \mu^* - |neigh(u, H^*_{min})|\}. \tag{4}$$

The ultimate score that is exploited for greedy selection is defined as

$$p(u) = \langle p'(u), p''(u) \rangle \tag{5}$$

and the queue for selecting the next node follows an inverse lexicographical order (u precedes w if $p'(u) > p'(w)$ or if $p'(u) = p'(w)$ and $p''(u) > p''(w)$). This way, we favor the connectivity constraint over the minimum-degree score.

Algorithm. The pseudocode of Greedy is reported as Algorithm 3. The algorithm takes as input a graph G, a set of query vertices Q, and the set H^* containing all the solutions to CSP for Q, and returns a smallest-sized, yet optimal, solution $H^*_{min} \subseteq H^*$. The algorithm keeps (Line 1): the current solution H^*_{min}, the set $\mathcal{A}$ of connected components of $G[H^*_{min}]$, the minimum degree μ^* that the ultimate solution is required to have, and a priority queue P that stores the vertices $u \in H^* \setminus H^*_{min}$ with priority equal to the score $p(u)$ reported in Equation (5). At the beginning, all query vertices are added to the queue with priority $+\infty$ (Line 2), so as to implicitly initialize H^*_{min} as equal to the query vertex set Q.

The main cycle of the algorithm (Lines 3-38) runs until H^*_{min} has become a valid solution to CSP for the query vertices Q, i.e., until there is a single connected component in $\mathcal{A}$ and the minimum degree of $G[H^*_{min}]$ is equal to μ^*. At each step of the cycle, a vertex u is extracted from the queue and added to H^*_{min} (Lines 4-6). Then, various operations need to be performed on the neighbors of u in H^*.

First of all (Lines 7-10), all u's neighbors v that are not in the queue are added to it (by firstly computing their priority $p(v)$). Afterwards (Lines 11-23), the connected components in $\mathcal{A}$ are updated: the connected components of each u's neighbor are collected in an auxiliary set $\mathcal{A}'$ (Line 20), and they are eventually merged in a unique connected component (Lines 22-23). At the same time, the degree $\alpha(v)$ of every u's neighbor v in H^*_{min} is increased by one (Line 13), and, if $\alpha(v)$ becomes equal to the desired μ^* value, the minimum-degree score $p''(w)$ of every v's neighbor w in the queue is updated accordingly (Lines 13-19).

Given that the connected components in $\mathcal{A}$ (may) have changed, we need to recompute also the connection score p' of every neighbor of a u's neighbor that belongs to a connected component among the ones merged, i.e., among the ones in $\mathcal{A}'$ (Lines 24-37). In particular, for every of such vertices w its connection score $p'(w)$ needs to be (i) decreased by the number of connected components in $\mathcal{A}'$ containing a w's neighbor (Lines 24-28), and (ii) increased by one because of the new connected component A^* created (Line 30). Additionally, for every of such vertices w that have u as a neighbor, its minimum-degree score $p''(w)$ is also updated (Lines 31-35).

Running time. We analyze now the time complexity of Greedy. Let $\widetilde{H}$ denote the set of vertices given by the union of the vertices in the output solution H^*_{min} and the neighbors

Algorithm 4 Connection

Input: A graph $G = (V, E)$; a set of query vertices $Q \subseteq V$; a set $H^*_{min} \subseteq H^* \subseteq V$ outputted by the Greedy step.
Output: A set of vertices $H^{**}_{min} \subseteq H^*$.
1: $\widehat{H}^*_{min} \leftarrow SteinerTree(G[H^*_{min}], Q)$
2: $H^{**}_{min} \leftarrow$Greedy$(G, \widehat{H}^*_{min}, H^*_{min})$

of each vertex in H^*_{min} that are part of H^*, i.e., $\widetilde{H} = H^*_{min} \cup \bigcup_{u \in H^*_{min}} neigh(u, H^*)$. It is easy to see that the largest part of the input graph visited by Greedy corresponds to the subgraph $G[\widetilde{H}]$ induced by $\widetilde{H}$. Let $\tilde{n}$ and $\tilde{m}$ denote the number of vertices and edges of $G[\widetilde{H}]$, respectively. The time complexity of Greedy is $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$, as proved next.

Theorem 3 *Greedy runs in $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$ time.*

Proof First of all, we assume that the operations of insertion, extraction, and update from the queue P are performed in time logarithmic in the size of P. Also, by keeping the neighbors of a vertex in the input graph G organized based on their core index, it is possible to visit the neighbors $neigh(u, H^*)$ of a vertex u in time $\mathcal{O}(|neigh(u, H^*)|)$ instead of $\mathcal{O}(\delta(u))$.

Let us analyze the most expensive steps of Greedy:

- The initialization phase (Lines 1-2) takes $\mathcal{O}(|Q| \log |Q|)$, while the extraction of a vertex from P (Line 4) takes $\mathcal{O}(\tilde{n} \log \tilde{n})$, as each vertex in $\widetilde{H}$ is extracted from P at most once.
- In the cycle at Lines 7-10, all neighbors of every vertex in $\widetilde{H}$ are visited, and this happens only once ($\mathcal{O}(\tilde{m})$ time). Moreover, this is the only point where a vertex can enter the priority queue ($\mathcal{O}(\tilde{n} \log \tilde{n})$ time) and that its p score is computed from scratch ($\mathcal{O}(\tilde{m})$ time, as, to compute the $p(u)$ score of a vertex u from scratch, all u's neighbors need to be visited, cf. Equations (1)–(5)). In summary, the overall runtime of this block is $\mathcal{O}(\tilde{m} + \tilde{n} \log \tilde{n})$.
- In the block at Lines 11-23, all neighbors of a vertex $u \in \widetilde{H}$ are visited again. Here is also the only point of the algorithm where a vertex v can reach the desired degree μ^* (Lines 14-19): in that case, all v's neighbors are visited again, and, for each of them, either constant-time (update of the p'' score, Line 16) or logarithmic-time operations (update of the priority in the queue, Line 17) are performed. This therefore takes $\mathcal{O}(\tilde{m} \log \tilde{n})$. Moreover, merging the connected components takes $\mathcal{O}(|Q|\tilde{n})$, as the connected components are at most $|Q|$, while each of them has size $\mathcal{O}(\tilde{n})$.
- Finally, the block at lines 24-37 considers each vertex in $\widetilde{H}$ as many times as it is merged in another connected component, i.e., a number of times at most $|Q|$. For each vertex, its neighbors are visited again (Lines 24-25, Line 29), and for each of such neighbors, either constant-time (Line 26, Lines 32-34) or logarithmic-time (Line 36) operations are performed. This block therefore takes $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$ time.

Considering all the above partial times, the overall time complexity of Greedy results to be $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$. □

As shown in our experiments in Section 5, usually $\tilde{n} \ll n$ and $\tilde{m} \ll m$; this makes the Greedy step very efficient.

The Connection step

The second step of the proposed overall GreedyConnection algorithm takes as input the subgraph produced by the Greedy step and performs a sequential procedure that first ensures connection among the query vertices (substep 1), and then aims at satisfying the constraint on the minimum degree (substep 2).

Algorithm. The first substep of Connection looks for a connected solution using as few more vertices as possible. This corresponds to solving the following problem:

Problem 3 Given a graph $G = (V, E)$, a set of query vertices $Q \subseteq V$, and a set of vertices $H^* \subseteq V$ containing all the solutions to CSP for the query Q, find a minimum-sized set of vertices $\widehat{H}^* \subseteq H^* \setminus Q$ such that $G[Q \cup \widehat{H}^*]$ is connected. □

Problem 3 can be shown to be $\mathbf{NP}$-hard by a reduction from STEINER TREE.

Theorem 4 *Problem 3 is* $\mathbf{NP}$*-hard.*

Proof Given an instance of STEINER TREE on an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of terminals $T \subseteq \mathcal{V}$, one can build an instance of Problem 3 by setting $G = \mathcal{G}$, $Q = T$, and $H^* = \mathcal{V}$. Using the same arguments as Theorem 2, it is easy to see that a set of vertices H of G is a solution to the instance of Problem 3 constructed this way if and only if any spanning tree of the subgraph induced by H is a solution to the original STEINER TREE problem instance. □

The analogy between Problem 3 and STEINER TREE allows us to borrow the theory developed for the latter problem. Particularly, we solve the first substep of Connection (i.e., Problem 3) by using the aforementioned algorithm by Kou *et al.* [12], which gives an approximation guarantee of $2 - 2/|Q|$. Then, in the second substep, the preliminary (connected) solution $\widehat{H}^*_{min}$ outputted by the first substep constitutes the set of query vertices to be given as input to Algorithm 3, in order to make it satisfying the minimum-degree constraint too. Note that, as $\widehat{H}^*_{min}$ is connected and Algorithm 3 adds at each step only vertices that are connected to the current solution, we use here a simplified version of Algorithm 3, where all operations needed for ensuring the connection constraint (i.e., computing/updating connected components and connection scores p') are discarded.

The outline of Connection is reported as Algorithm 4.

Running time. The first substep of Connection, i.e., running the approximation algorithm by Kou *et al.* [12] on the subgraph induced by H^*_{min}, takes $\mathcal{O}(\tilde{m} + \tilde{n} \log \tilde{n})$ time by using the fast implementation by Mehlhorn [15]. The second substep of Connection takes instead the time needed to run the simplified version of Algorithm 3 on the query vertices $\widehat{H}^*_{min}$ outputted by the first substep. The query vertices are now already connected, thus the connection constraint can be discarded and a simplified version of Algorithm 3 can be used. From the complexity analysis in Theorem 3 it is easy to see that this simplified version of Algorithm 3 takes $\mathcal{O}(\tilde{m} \log \tilde{n})$ time (rather than $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$ of the general case). The overall time complexity of Connection is thus $\mathcal{O}(\tilde{m} \log \tilde{n})$.

The overall GreedyConnection algorithm

The overall proposed GreedyConnection algorithm consists in running the Greedy and Connection steps sequentially. Based on the time complexity analyses reported above, the overall running time of GreedyConnection is $\mathcal{O}(|Q|\tilde{m} \log \tilde{n})$.

Table 2 Characteristics of the selected datasets: # vertices (n), # edges (m), # distinct cores (h), graph type.

	n	m	h	type
Email	$33,696$	$180,811$	43	Communication network
Web-NotreDame	$325,729$	$1,090,108$	41	Web graph
Web-Google	$791,822$	$3,815,994$	40	Web graph
Youtube	$1,134,889$	$2,987,623$	51	Social network
Flickr	$1,624,992$	$15,476,835$	567	Social network

GreedyConnection achieves the best tradeoff between efficiency and accuracy. Indeed, based on the above argument about small minimum degree exhibited by real-world communities, running only the Greedy step would be very efficient but not that accurate (it produces larger communities). On the other hand, running only the Connection step, without profitably exploiting the input reduction provided by Greedy, would achieve high quality (smaller communities) but poor efficiency.

5 Experiments

In this section we empirically evaluate the performance of the proposed method GreedyConnection (for short, GrCon) and compare it to the state-of-the-art methods described in Section 2.2, i.e., GS and K-GS (for multiple-vertex queries), and LS (for one-vertex queries). We consider LS only for the special case of a single query vertex since that method cannot handle multiple-vertex queries.

We use real-world, publicly-available graphs whose main characteristics are reported in Table 2.[3] For each graph we take its largest connected component.

We aim at assessing both efficiency and quality of the proposed method and its competitors. As far as quality, we evaluate communities H produced by each method in terms of size $|H|$, density $|E[H]|/\binom{|H|}{2}$, and query-biased density [19]. Density is a well-established measure for assessing the quality of a community and it represents a desirable "side effect" that a community identified by optimizing any other criterion should in principle exhibit. Query-biased density is a quality measure that has been recently introduced to effectively assess the goodness of a community without suffering from the so-called *free-rider effect* (vertices attached by weak links to a strong group) [19].

The quality of our GreedyConnection is independent from the preprocessing strategy used (CoreStruct or ShellStruct). Preprocessing only affects efficiency: unless otherwise specified, the runtimes reported for our GreedyConnection always comprise the retrieval time from the slowest one among the two preprocessing methods (i.e., ShellStruct).

For each set of experiments, we sample 100 (sets of) query vertices at random (from either the whole graph or different parts of the graph—more details on this later), and, for each measure, we report the average over this set of queries, along with statistical significance of the difference among the various methods. For the latter we report p-values computed according to the well-established Wilcoxon signed rank test [6].

All methods are implemented in Java 1.7, and experiments are run on an Intel Xeon CPU at 2.2GHz and 96GB RAM machine, which we limit to 30GB. The source code is available at `http://bit.ly/1b6WbSQ`.

[3] Flickr is available at `http://socialnetworks.mpi-sws.org/datasets`, while all other graphs at `https://snap.stanford.edu/data`.

Table 3 Multiple-vertex query evaluation (number of query nodes $|Q| = 8$): proposed GrCon method vs. GS (top table) and vs. K-GS (bottom table) state-of-the-art methods.

	time(s)			*size*			*density*			*query-biased density*		
	GrCon	GS	p-value	GrCon	GS	p-value	GrCon	GS	p-value	GrCon	GS	p-value
Email	**0.214**	0.277	0.02	**2k**	12k	2.4E-5	**0.045**	0.017	1.1E-5	**7.7E-4**	4.7E-4	0.014
Web-ND	**2**	33	8.8E-33	**12k**	87k	3.8E-4	**0.026**	0.003	3.7E-4	**2.2E-4**	1.3E-5	1.9E-9
Web-G	**8**	95	3.2E-5	**33k**	351k	0.002	**0.095**	0.044	9.6E-8	**0.01**	9.9E-11	1.9E-9
Youtube	**71**	969	5.5E-29	**42k**	290k	0.001	**0.012**	0.002	0.01	**2.7E-5**	3.4E-6	2.4E-6
Flickr	**131**	1,316	7.5E-6	**103k**	546k	0.03	**0.017**	8.8E-4	0.005	**3.9E-5**	1.2E-6	0.027

	GrCon	K-GS	p-value	GrCon	K-GS	p-value	GrCon	K-GS	p-value	GrCon	K-GS	p-value
Email	**0.214**	0.904	2.7E-10	**2k**	4.9k	0.02	**0.045**	0.038	**0.56**	**7.7E-4**	6.1E-4	*0.82*
Web-ND	**2**	8	1.7E-7	**12k**	38k	2.4E-4	0.026	**0.028**	*0.87*	**2.2E-4**	1.8E-4	*0.05*
Web-G	**8**	32	0.001	**33k**	265k	0.01	**0.095**	0.089	0.02	**0.01**	0.009	*0.82*
Youtube	71	**31**	*0.08*	**42k**	71k	0.004	**0.012**	0.004	0.04	**2.7E-5**	1.4E-5	*1*
Flickr	131	**113**	*0.30*	**103k**	281k	0.03	**0.017**	0.002	*0.19*	**3.9E-5**	3.1E-5	*0.28*

5.1 Multiple-vertex queries

In the evaluation for multiple-vertex queries we compare the proposed GrCon to the state-of-the-art GS and K-GS methods. We set the constraint on the community size for K-GS to the size of the corresponding community yielded by our GrCon method. We recall that this is only a soft constraint: K-GS may output communities of size larger than the constraint.

The results of this evaluation are illustrated in Figure 1, where we show results with varying the number of query vertices, and Table 3, which reports a summary of the results for 8 query vertices, by also including statistical significance. For each graph, the (100) sets of query vertices used are sampled uniformly at random from various cores of the graphs, particularly from cores of order 1 (i.e., the whole graph), 2, 4, 8, 16, 32. The reported results are averaged over all those query sets.

Our GrCon method outperforms GS in terms of running time—one/two orders of magnitude faster in all cases, as well as quality—size/density consistently smaller/larger, with size being one order of magnitude smaller in most cases, and density up to two orders larger. Concerning the results over different datasets, the smallest gap between the proposed method and the two competitors (especially K-GS) is on the two smallest-density datasets, i.e., Web-ND and Youtube. This comply with the design principles of our method. In fact, for any given set of query vertices, our method needs to find a subgraph that satisfies two requirements: (i) query vertices are connected, and (ii) the minimum degree of the subgraph is equal to a certain value. The two requirements have to be satisfied by keeping the size of the output subgraph as small as possible. Both these requirements take benefit from a dense graph: in a dense graph, in fact, a few additional vertices typically suffice to connect query vertices and get to the desired minimum-degree value.

K-GS behaves quite well (comparable to and in some cases better than our GrCon) for small numbers (i.e., 2) of query vertices, while its performance degrades as query vertices increase. This complies with the design principles of that method. It indeed consists of a preprocessing phase aimed at reducing the input graph by including the minimum number of vertices that are close to the query vertices and make the resulting graph connected and with size at least K. It is hence apparent that more query vertices lead to retaining a larger graph during such a preprocessing phase, thus making the benefit of working on a reduced graph progressively disappear. Indeed, for query vertices from four on, K-GS is consistently outperformed by GrCon in terms of all metrics, i.e., running time, size, and density. For large query sets (i.e., $|Q| = 16$ or $|Q| = 32$), K-GS performs even worse than GS.

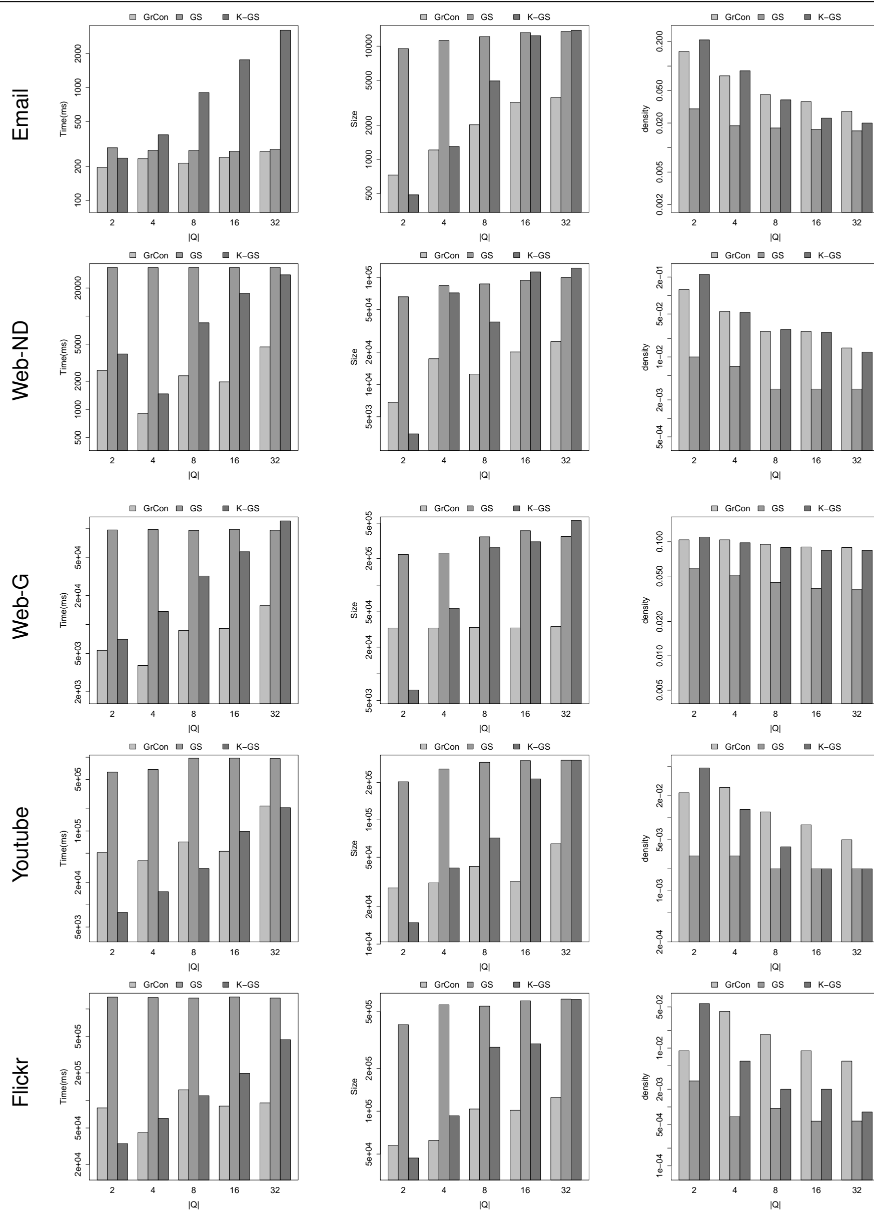

Fig. 1 Multiple-vertex query evaluation with varying number of query vertices $|Q|$: proposed GrCon method vs. GS and K-GS state-of-the-art methods. Results are shown in logarithmic scale.

As a general trend exhibited by all methods, the size (resp. density) of the solutions produced is increasing (resp. decreasing) as the number of query vertices increases. This is expected: the more the query vertices, the larger the number of other vertices needed to make them connected, and, hence, the larger the size and the smaller the density of the resulting solution. Also, running time is increasing with the number of query vertices only for GrCon and K-GS, while it results roughly constant for GS. This is in accordance with the time

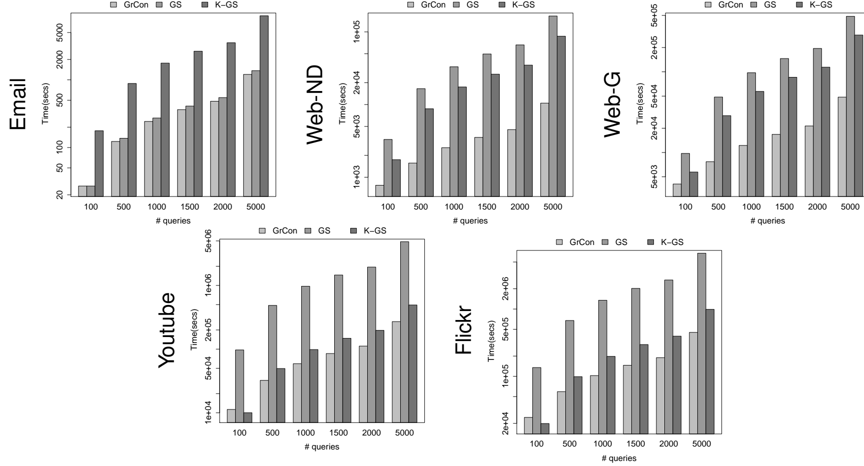

Fig. 2 Multiple-vertex query-evaluation cumulative times (i.e., preprocessing time + query time) of proposed GrCon method vs. GS and K-GS state-of-the-art methods with varying the number of queries. Number of query vertices $|Q| = 16$. Times are shown in logarithmic scale.

Table 4 One-vertex query evaluation (core index of the query vertex $c(v) = 8$): proposed GrCon method vs. LS state-of-the-art method.

	time(s)			*size*			*density*			*query-biased density*		
	GrCon	LS	p-value	GrCon	LS	p-value	GrCon	LS	p-value	GrCon	LS	p-value
Email	**0.065**	0.490	3.1E-27	**0.9k**	2.5k	2.8E-4	**0.190**	0.066	0.008	**0.003**	4.9E-4	9.1E-5
Web-ND	**0.056**	1	4E-7	**1.7k**	16k	2.8E-6	**0.478**	0.151	2.3E-4	**0.04**	0.01	3.9E-5
Web-G	**0.690**	9	6.5E-5	**21k**	76k	0.004	**0.673**	0.281	5.8E-4	**0.07**	0.03	2.9E-5
Youtube	**6**	24	1.7E-7	**23k**	36k	0.03	**0.010**	0.005	1.1E-4	**3.3E-5**	2.6E-6	0.002
Flickr	**92**	762	0.002	**68k**	94k	0.002	**0.011**	0.010	0.002	**1.1E-4**	1.4E-5	0.05

complexity of the three methods: GrCon and K-GS have running time that depends on the number of query vertices, while GS does not (see Sections 2 and 4.2).

The summary reported in Table 3 shows that in almost all cases where our GrCon outperforms a baseline the difference is statistically significant (p-value < 0.05). The few exceptions include the comparison to K-GS in terms of density on the Email and Flickr datasets. For what concerns the few cases where our GrCon is outperformed by K-GS (time on Youtube and Flickr, and density on Web-ND), it is remarkable to note that the difference is not statistically significant, thus meaning that in these cases the two methods are comparable. In Table 3 we also report the results in terms of query-biased density, which show that our GrCon significantly outperforms GS while being comparable to K-GS.

Cumulative time evaluation. In Figure 2 we report an analysis of the efficiency of the various methods in a real-world scenario where (i) cumulative times for a certain number of queries are shown, and (ii) the times of our GrCon comprise the preprocessing time needed to build the offline information, which, we recall, is needed *una tantum* and can be exploited by all the subsequent queries. Particularly, as the preprocessing time, we consider the time needed to compute the most time-consuming offline information, i.e., ShellStruct.

The superiority of our GrCon remains apparent also in this evaluation. Particularly, the advantage over the competitors increases as the number of queries increases: this is expected as more queries allows for better amortizing the time spent in preprocessing.

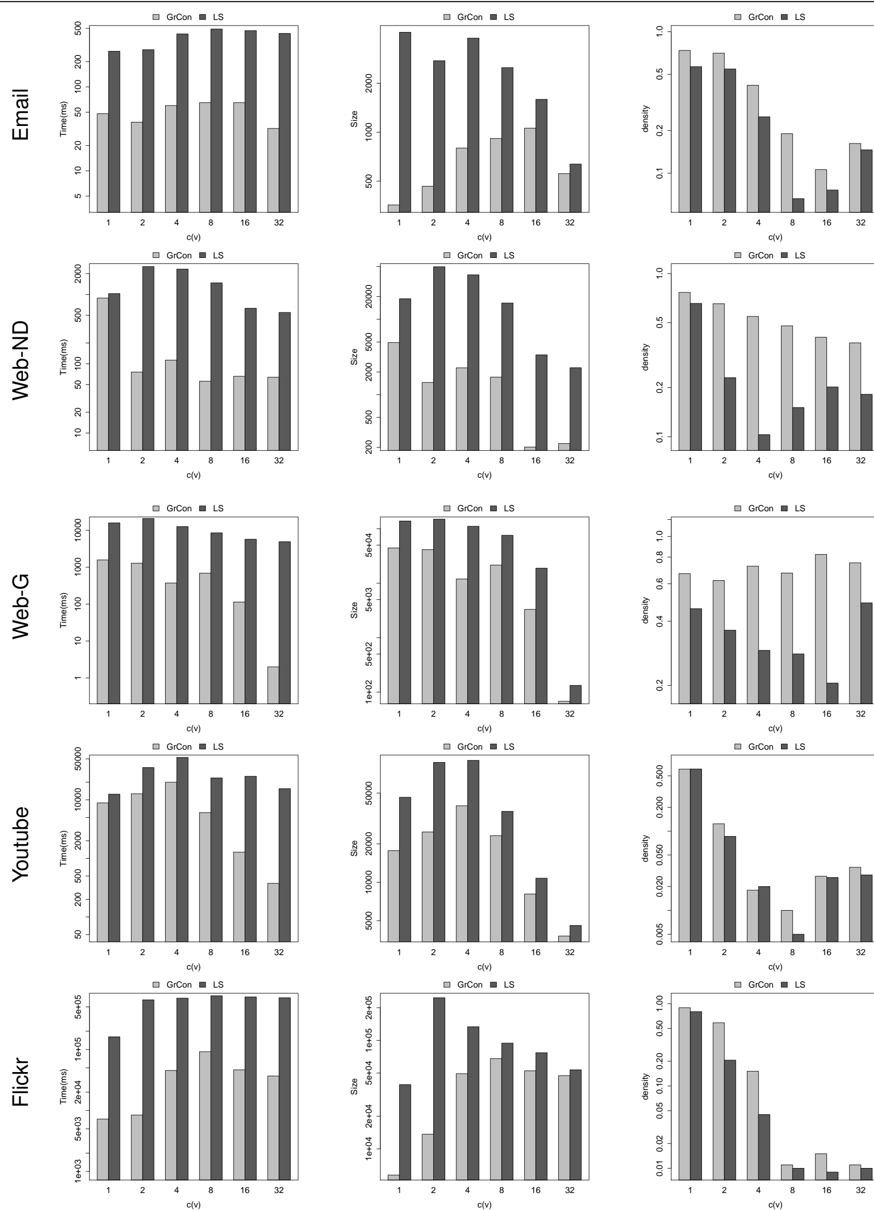

Fig. 3 One-vertex query evaluation with varying core index $c(v)$ of the query vertex v: proposed GrCon method vs. LS state-of-the-art method. Results are shown in logarithmic scale.

5.2 One-vertex queries

For the special case of one-vertex queries we aim at comparing the proposed GreedyConnection against LS, which is the state-of-the-art method for this special type of query and has been recognized as more efficient and effective than GS and K-GS [5]. Note that for this special case no connection constraint among query vertices is required, thus here we use

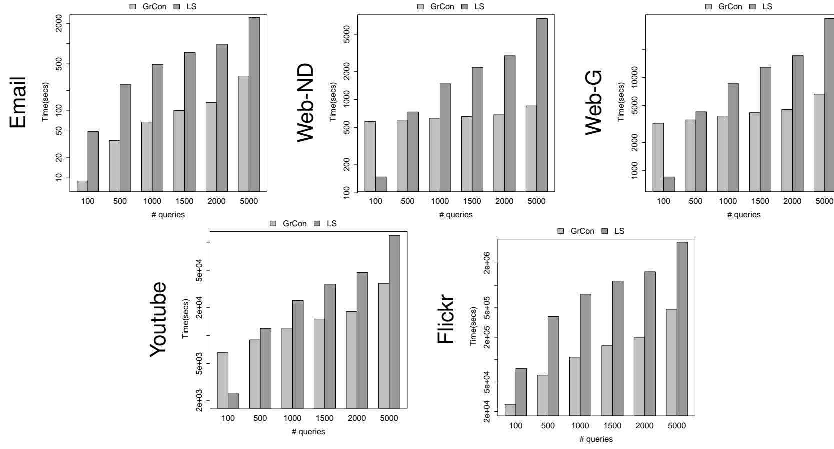

Fig. 4 One-vertex query-evaluation cumulative times (i.e., preprocessing time + query time) of proposed GrCon vs. LS state-of-the-art method varying the number of queries. Core index of the query vertex $c(v) = 8$. Times are shown in logarithmic scale.

Table 5 Building time and storage space of the proposed preprocessing methods.

	building time (s)		*space* (MB)	
	CoreStruct	ShellStruct	CoreStruct	ShellStruct
Email	1	3	7	4
Web-ND	12	574	41	28
Web-G	43	$3,155$	147	90
Youtube	201	$5,946$	130	100
Flickr	$1,765$	$15,865$	564	347

a simplified version of our GreedyConnection method which comprises only the first one of the two steps described in Section 4.2 (i.e., only the Greedy step).

The results of this evaluation are reported in Figure 3 (query vertices sampled uniformly at random from different cores of the graph) and Table 4 (summary on query vertices sampled from core 8 along with statistical-significance evaluation). GrCon consistently outperforms LS in all cases, in terms of accuracy and efficiency. Indeed, GrCon is always at least one/two orders of magnitude faster than LS, and in some cases the gap is even of three orders of magnitude. At the same time, the solutions produced by GrCon have much smaller size and larger density than LS: in most cases the size gap is one order of magnitude while the density exhibited by GrCon's solutions is on average 73% larger than LS.

Table 4 shows that the superiority of our GrCon is always statistically significant, and it is confirmed in terms of the query-biased density measure too.

Finally, in Figure 4 we show cumulative times for various numbers of queries. Times of our method here include the time needed for building the ShellStruct structure.

5.3 Preprocessing-phase evaluation

We now focus on the preprocessing methods CoreStruct and ShellStruct. Table 5 contains building time and storage space, while Table 6 shows running times of the retrieval phases. These results confirm what theoretically stated in Section 3: ShellStruct offers less storage space but requires more building time than CoreStruct. However, for both methods retrieval time is always < 1 ms, thus being negligible if compared to the online-processing times.

Table 6 Retrieval time (μs) of the proposed preprocessing methods with varying $|Q|$.

	$\|Q\| = 1$		$\|Q\| = 2$		$\|Q\| = 4$		$\|Q\| = 8$		$\|Q\| = 16$		$\|Q\| = 32$	
	Core Struct	Shell Struct	Core Struct	Shell Struct	Core Struct	Shell Struct	Core Struct	Shell Struct	Core Struct	Shell Struct	Core Struct	Shell Struct
Email	14	17	18	48	26	115	41	284	69	361	121	153
Web-ND	17	16	18	50	26	106	42	232	76	227	144	420
Web-G	36	19	20	69	30	169	49	242	115	311	197	350
Youtube	13	17	19	39	27	81	47	198	77	335	137	338
Flickr	13	18	18	57	26	128	40	275	68	485	123	129

6 Related Work

Community search. Community search is the problem of finding a good community for a set of query vertices. A number of community-quality measures have been proposed, based on notions such as random walk [18, 11], α-adjacency-γ-quasi-k-clique [4], k-truss [10], influence [14], or query-biased density [19]. Sozio and Gionis [17] are the first to provide a combinatorial-optimization formulation to the community-search problem, and, in particular, they ask for a connected subgraph that contains all query vertices and maximizes the minimum degree. This formulation of community search has been further studied in the literature [5] and is the specific formulation we focus on in this work. All main related research on this topic has already been discussed in more detail in Section 2.2.

Dense-subgraph discovery. Finding dense subgraphs is a long-standing problem in data mining [13]. This problem aims at finding a subgraph of a given input graph that maximizes some notion of density. A density notion widely employed in the literature is the average degree. Finding a subgraph that maximizes the average degree can be solved in polynomial time [9], and approximated within a factor of $\frac{1}{2}$ in linear time [3].

Dense-subgraph discovery differs from community search as it does not permit to specify any query vertices to be contained in the output subgraph/community.

Community detection. *Community detection* and the related problem of *graph partitioning* have been extensively studied in the literature. The goal of such problems is to partition the vertices of a graph into communities so as to maximize edges between vertices in the same community and minimize edges between vertices in different communities. Representative approaches to community detection include spectral, minimum-cut, modularity-based methods, and label propagation [20]—in fact, the literature on the topic is so extensive that we do not attempt to make a proper review here; a comprehensive survey can be found in [7].

The goal of community detection is to find a global community structure for the whole input graph; it thus differs from community search, whose objective is instead to find a community for a set of input query vertices in an online fashion.

7 Conclusions

Given a graph and a set of query vertices, the community-search problem requires to find a cohesive subgraph that contains the query vertices. A well-established criterion to measure the quality of the community to be discovered is the minimum degree of the output subgraph. A number of methods have been proposed for this specific formulation of community search, but they all have shortcomings about efficiency, as they need to visit (large part of) the whole input graph, effectiveness, as they tend to find large and not really cohesive communities, or lack of generality, as they cannot handle multiple query vertices, do not find communities with optimal minimum degree, and/or require input parameters.

This paper advances the state of the art on min-degree-based community search by proposing a method that overcomes all weaknesses of existing approaches: it is orders of magnitude faster and more effective, while, at the same time, being capable of handling multiple query vertices, producing optimal communities, and being parameter-free.

An extensive evaluation performed on several real-world graphs attests the superiority of the proposed method from efficiency, effectiveness, and generality viewpoints.

In the future we plan to further study the problem of minimum community search (Problem 2) by devising alternative heuristics that can further improve efficiency and/or accuracy. Another interesting direction is to carry over the strategies proposed in this paper to community-search-like problems involving more complex types of graph, such as heterogeneous networks, labeled networks, or dynamic graphs.

8 Compliance with Ethical Standards

Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.
2. P. Bogdanov, B. Baumer, P. Basu, A. Bar-Noy, and A. K. Singh. As strong as the weakest link: Mining diverse cliques in weighted graphs. In *Europ. Conf. on Machine Learning and Knowledge Discovery (ECML/PKDD)*, pages 525–540, 2013.
3. M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Int. Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, pages 84–95, 2000.
4. W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 277–288, 2013.
5. W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 991–1002, 2014.
6. J. Demšar. Statistical comparisons of classifiers over multiple data sets. *J. of Machine Learning Research (JMLR)*, 7:1–30, 2006.
7. S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.
8. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *ACM Symposium on Theory of Computing (STOC)*, pages 246–251, 1983.
9. A. V. Goldberg. Finding a maximum density subgraph. Technical report, University of California at Berkeley, 1984.
10. X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 1311–1322, 2014.
11. Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity graphs in networks. *ACM Trans. on Knowledge Discovery from Data (TKDD)*, 1(3), 2007.
12. L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15(2):141–145, 1981.
13. V. E. Lee, N. Ruan, R. Jin, and C. C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. 2010.
14. R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proc. of the VLDB Endowment (PVLDB)*, 8(5):509–520, 2015.
15. K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Inf. Proc. Letters*, 27(3):125–128, 1988.
16. S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.
17. M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 939–948, 2010.
18. H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 404–413, 2006.
19. Y. Wu, R. Jin, J. Li, and X. Zhang. Robust local community detection: On free rider effect and its elimination. *Proc. of the VLDB Endowment (PVLDB)*, 8(5):798–809, 2015.
20. J. Xie and B. K. Szymanski. Labelrank: A stabilized label propagation algorithm for community detection in networks. In *IEEE Network Science Workshop*, 2013.