Efficient Progressive Minimum k-Core Search

Conggai Li[‡†], Fan Zhang[‡], Ying Zhang[†], Lu Qin[†], Wenjie Zhang[§], Xuemin Lin[§]

[‡]*Guangzhou University,* [†]*CAI, University of Technology Sydney,* [§]*University of New South Wales*
{conggai.li.cs, fanzhang.cs}@gmail.com, {ying.zhang, lu.qin}@uts.edu.au, {zhangw, lxue}@cse.unsw.edu.au

ABSTRACT

As one of the most representative cohesive subgraph models, k-core model has recently received significant attention in the literature. In this paper, we investigate the problem of the minimum k-core search: given a graph G, an integer k and a set of query vertices $Q = \{q\}$, we aim to find the smallest k-core subgraph containing every query vertex $q \in Q$. It has been shown that this problem is NP-hard with a huge search space, and it is very challenging to find the optimal solution. There are several heuristic algorithms for this problem, but they rely on simple scoring functions and there is no guarantee as to the size of the resulting subgraph, compared with the optimal solution. Our empirical study also indicates that the size of their resulting subgraphs may be large in practice. In this paper, we develop an effective and efficient progressive algorithm, namely *PSA*, to provide a good trade-off between the quality of the result and the search time. Novel lower and upper bound techniques for the minimum k-core search are designed. Our extensive experiments on 12 real-life graphs demonstrate the effectiveness and efficiency of the new techniques.

PVLDB Reference Format:
Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficient Progressive Minimum k-Core Search. *PVLDB*, 13(3): 361-374, 2019.
DOI: https://doi.org/10.14778/3368289.3368300

1. INTRODUCTION

Graphs are widely used to model relationships in various applications. Query processing and mining with cohesive subgraphs is one of the fundamental problems in graph analytics, where the main aim is to find groups of well-connected graph vertices. k-core is an important cohesive subgraph model based on *k-core constraint*: a subgraph is a k-core (subgraph) if every vertex has at least k neighbors in the same subgraph. Problems related to k-core model have been intensively studied in the literature, with many existing research efforts mainly focusing on the *maximum* k-core computation, which aims to find the *largest* induced subgraph satisfying k-core constraint. Indeed, there are many important applications for maximum k-cores, most notably user engagement [11, 30] and influence evaluation [53, 25, 41]. Nevertheless, in some scenarios especially when one or a set of query vertices are involved, users may prefer small size group because the group size may closely related to the costs (e.g., verification or advertisement/recommendation costs), and the stableness and homophily of the group. In these scenarios, the maximum k-core may contain many extraneous vertices, and it is more natural to find k-core subgraphs containing all query vertices with smallest size. From theoretical perspective, it is also an interesting optimization problem given the degree constraint and optimization goal (i.e., minimizing the subgraph size).

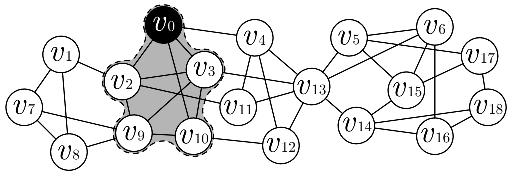


Figure 1: A Toy Example, $k = 3$

In Figure 1, the graph G is a k-core with $k = 3$. However, for the given query vertex v_0, it is more intuitive to return the subgraph $\{v_0, v_2, v_3, v_9, v_{10}\}$ as a k-core subgraph containing v_0 with $k = 3$, instead of using the whole graph G. In this paper, we study the problem of *minimum* k-core search which aims to find the smallest k-core subgraph containing the given query vertex. The applications and challenges associated with this problem are discussed below.

Applications. The importance of the minimum k-core search problem can be reflected through some concrete examples in the following representative applications.

<u>Social Networks.</u> It is a common practice to encourage the engagement of group members in the social network by utilizing the positive influence from their friends in the same group (e.g., [45, 20, 11, 30]). In some applications such as group recommendation with one or a set of query vertices (e.g., specific users), a small k-core subgraph may be preferred because a large k-core subgraph may contain many irrelevant vertices for the query vertex. In online social platforms (e.g., Groupon, Pinduoduo and Meetup), the system

[*]Conggai Li and Fan Zhang are the joint first authors. Fan Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368300

may recommend an item (e.g., ticket or event) to a user based on the information from a social group containing the user. Usually, the item has been adopted (e.g., bought) by some users in the group, and there are certain interactions among these users. Consider the recommendation cost and the stableness of the group, using the smallest k-core subgraph is more feasible than using the large ones. By doing this, the recommendation is more likely to be adopted, and the group members will be better motivated to take action (e.g., buy items) together due to positive peer influence [28].

Biological networks. It has been reported in [2, 7] that in protein-protein interaction (PPI) networks, the proteins in the same k-core subgraph are likely to have the same functionalities. However, our empirical study reveals that such homophily property only holds for small k-core subgraphs, not for large ones. Thus, it is more reliable to use the minimum k-core subgraph if we aim to find a group of proteins such that, with high probability, they have the same functionality as the query vertex (i.e., homophily property). The cell-assembling theory [22] in neural networks is another example. This theory suggests that information processing in the brain is based on the collective action of groups of neurons, which is essential for acquiring memories (e.g., [22, 13, 44, 34, 38]). As shown in [34, 38], a neuron can be fired/activated if a certain number of neighbor neurons have been activated. This implies that, to activate an area of neurons, we can initially stimulate a small k-core subgraph for cost-effectiveness purpose.

Challenges. As shown in [18, 9, 5], the minimum k-core search problem is NP-hard and cannot be approximated with any constraint factor. We remark that this problem is more challenging than finding a k-clique containing q, although the latter is also NP-hard, in the sense that minimum k-core search needs to explore neighbors with more than one hop. As such, it is cost-prohibitive to find optimal solution for minimum k-core search problem in practice given the huge search space involved.

To circumvent this obstacle, existing studies [18, 9] propose greedy algorithms to incrementally include candidate vertices according to their scoring functions till the resulting subgraph is a k-core subgraph. These algorithms are simple and time efficient, but they do not offer any quality guarantee over the size of the resulting k-core subgraph. Therefore, we propose a progressive algorithm by developing novel techniques to incrementally derive lower and upper bounds for the size of the minimum k-core containing the query vertex. By doing this, we can safely terminate the search once the desired approximate ratio is reached.

Contributions. Our principal contributions are as follows.

- We study the problem of the minimum k-core search which aims to find the smallest k-core subgraph containing the given query vertices. An effective and efficient Progressive Search Algorithm, namely *PSA*, is proposed to provide an approximate solution by incrementally computing lower and upper bounds of the optimal solution.
- We investigate three approaches to compute the lower bound of the optimal solution after mapping the problem of lower bound computation to the set multi-cover problem. We also design an *onion-layer* based heuristic algorithm to find small k-core subgraphs, and the smallest k-core subgraphs seen so far will serve as the upper bound as well as the approximate solution. The proposed techniques may shed light on other search problems related to cohesive subgraphs.
- We conduct comprehensive experiments on 12 real-life graphs to evaluate the proposed techniques. It shows that the proposed techniques outperform the state-of-the-art method *S-Greedy* [9] in two ways: (1) by using our upper bound technique alone, the corresponding greedy algorithm, namely *L-Greedy*, dominates *S-Greedy* under all settings in the experiments; and (2) *PSA* algorithm equipped with both lower and upper bounds techniques can further significantly reduce the resulting subgraph size and provide good trade-off between result quality and the search time.

Table 1: Summary of Notations

Notation	Definition
G	an unweighted and undirected graph
$V(G), E(G)$	the vertex set and edge set of G
u, v	vertex in the graph
n, m	the number of vertices and edges in G
d_{max}	the largest degree value in G
$N(u, G)$	the set of adjacent vertices of u in G
$deg(u, G)$	$\|N(u, G)\|$
k	the degree constraint for k-core subgraph
$C_k(G)$	k-core of G
$P(G)$	partial solution of G

2. PRELIMINARIES

2.1 Problem Definition

Let $G = (V, E)$ be an undirected and unweighted graph, where V and E denote the set of vertices (nodes) and edges respectively. Let $n = |V|$ and $m = |E|$ be the number of vertices and edges respectively. We use $N(u, G)$ to denote the set of adjacent vertices of u in the graph G, which is also known as the neighbor set of u in G. Let $deg(u, G)$ denote the degree of u in G, which is the number of adjacent vertices of u in G. Given a subgraph S of G, we use $V(S)$ to denote its vertices. The size of the subgraph, denoted by $|S|$, is the number of the vertices. When the context is clear, $N(u, G)$, $deg(u, G)$ and $V(S)$ are simplified to $N(u)$, $deg(u)$ and S respectively. By d_{max} we denote the largest vertex degree in the graph. Table 1 summarizes the notations.

Definition 1. ***k*-core subgraph.** *Given a graph G, a subgraph S of G is a k-core subgraph if every vertex in S has at least k neighbors in S.*

For presentation simplicity, we use k-core to represent k-core subgraph in this paper when there is no ambiguity. In the literature, many studies focus on the maximum k-core which is the largest induced subgraph which satisfies the k-core constraint. Note that the maximum k-core is also the *maximal* k-core because the maximal k-core of G is unique for a given k, as shown in [52].

Problem Statement. Given an undirected and unweighted graph $G = (V, E)$, a degree constraint k and a query set $Q = \{q_1, q_2, ...\}$, the minimum k-core search problem finds the **smallest** k-core subgraph containing the query set Q. Due to the NP-hardness of the problem and the huge search space involved to find the exact solution, in this paper we aim to develop an effective and efficient progressive algorithm to enable users to achieve a good trade-off between the quality of the results (i.e., the size of the k-core subgraph retrieved) and the search time.

For presentation simplicity and the ease of understanding, we first focus on computing the minimum k-core containing one query vertex q. The proposed algorithms are adapted for the query of multiple vertices in Section 3.4.

2.2 Existing Solutions

As discussed in Section 1, the search space of the exact solution is prohibitively large, and existing solutions resort to simple heuristic algorithms. In particular, existing solutions follow two search strategies: (1) *shrink* strategy [37], called *global search*; and (2) *expansion* strategy [9, 18], called *local search*. In the global search strategy [37], the maximal k-core is the initial result, which can be efficiently computed in linear time [10]. Then the size of the resulting subgraph will be shrunken by repeatedly removing the vertex while keeping q in the resulting k-core subgraph[1]. As shown in [9], the global search method in [37] is ineffective because the size of maximal k-core is usually very large, and the quality of the resulting k-core subgraph is not competitive with the local search approaches. In recent studies [9, 18], the local search strategy is adopted which starts from the query vertex and then expands the resulting subgraph by incrementally including the most promising candidate vertex at each step following some greedy heuristics. Below, we present the state-of-the-art technique proposed in [9].

State-of-the-art. In [9], a greedy algorithm, named *S-Greedy*, is proposed to find the minimum k-core subgraph containing the query vertex. Let P denote the resulting subgraph, which is initialized as $\{q\}$. By C we denote candidate vertices which are neighbors of the vertices in P, not contained by P. In each iteration, the most promising candidate vertex is chosen for inclusion in P, and the candidate set C is updated accordingly. P is returned when every vertex in P satisfies the k-core constraint. The key of the algorithm is the design of the scoring function to measure the goodness of candidates. In [9], the authors employ two functions to qualitatively measure the advantage and disadvantage of including a vertex u in P, denoted by $p^+(u)$ and $p^-(u)$ respectively. Specifically, $p^+(u)$ records the number of neighbors of u in P with a degree still less than k, i.e.,

$$p^+(u) = |\{v|v \in N(u,P), deg(v,P) < k\}|$$

Intuitively, the larger the $p^+(u)$ value, the better chance that u can make more vertices in P to satisfy the k-core constraint. The cost of including u in P is that an extra number of vertices needed to make u have at least k neighbors in P, i.e.,

$$p^-(u) = max\{0, k - |N(u,P)|\}$$

Then the score of the vertex u is defined as $p^+(u) - p^-(u)$, where the larger value is preferred.

Discussion. The proposed algorithm is time efficient. The time complexity of the score computation at each iteration is $O(d_{max})$ as only the neighbors of the vertex u are involved and it takes $O(\ln(n))$ time to maintain the most promising candidate vertex. Thus, the time complexity of *S-Greedy* is $O(s(d_{max} + \log(n)))$ where s is the size of the resulting subgraph. In [9], the maximal k-core will be computed for the following computation and hence s is bounded by the size of maximal k-core, which is usually a large number in practice. Experiments show that *S-Greedy* significantly outperforms other competitors by means of the resulting subgraph size. However, our empirical study indicates that there is still a big gap between its resulting subgraph and the optimal solution. Moreover, all existing algorithms do not have the quality guarantee about the size of the resulting subgraph and hence it is difficult for users to make a trade-off between result quality and search time. This motivates us to develop a progressive algorithm in this paper.

[1]There is no k-core subgraph containing the query vertex q if it is not in the maximal k-core.

3. PROGRESSIVE SEARCH ALGORITHM

3.1 Motivation and Framework

In this paper, we devise a <u>p</u>rogressive <u>s</u>earch <u>a</u>lgorithm, namely *PSA*. Given a vertex set V_t as a partial solution, we find that it is feasible to compute the upper/lower bounds of the minimum size of a k-core containing V_t (the details are introduced in later sections). Thus, if we can progressively converge the size upper/lower bounds of the partial solutions in a search, it is possible to compute a k-core with guaranteed size approximation ratio regarding the size of optimally-minimum k-core. Then, the framework of *PSA* is designed as a <u>B</u>est-<u>F</u>irst <u>S</u>earch (BesFS) which computes the result in an expansion manner and visits the most promising branch at each search step.

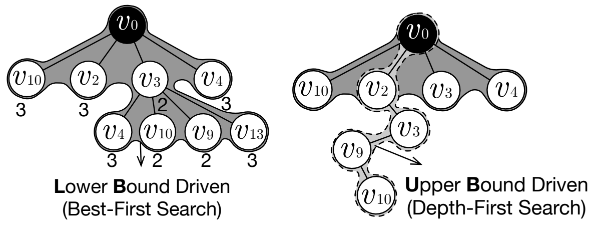
A search tree of BesFS is constructed along with the procedure of BesFS where the root is the query vertex and every tree node contains one vertex. For each tree node t, its partial solution V_t contains the vertex in the node and all the vertices in the ancestor nodes. In *PSA*, when a tree node t is visited and t contains the vertex u, we add the child nodes of t to the search tree where each child node contains a unique neighbor of a vertex in V_t with vertex id larger than u. Then, the search step at node t is processed as follows:

(i) Lower bound driven. For the partial solution V'_t of every child node t' of the node t, we compute a size lower bound $s^-(t)$ of the minimum k-core containing V'_t (introduced in Section 3.2). The next node to visit (the most promising node) is the one with the smallest lower bound $s^-(\cdot)$ from all the leaf nodes in current search tree.

(ii) Upper bound driven. For the partial solution V'_t of every child node t' of the node t, we conduct a <u>D</u>epth-<u>F</u>irst <u>S</u>earch (DFS) to compute a minimal k-core containing V'_t by heuristics (introduced in Section 3.3), to update the global size upper bound s^+ of the optimally-minimum k-core.

The algorithm *PSA* will return if $\frac{s^+}{s^-} \le c$ is satisfied for current search node, where c is the approximate ratio.

Example 1. *Figure 2 illustrates a part of the search tree $\mathcal{T}$ of our PSA algorithm, where the root is the query vertex v_0. There are 4 neighbors of v_0 with vertex id larger than v_0: v_2, v_3, v_4 and v_{10}. When v_0 is visited, for each neighbor of v_0, we attach a child node to the current visited node in $\mathcal{T}$, where each child node contains exactly one unique neighbor of v_0. The partial solution for the node containing v_3 is $\{v_0, v_3\}$. In the BesFS, suppose the node t has the smallest size lower bound of the minimum k-core containing*



(a) Lower Bound Driven (b) Upper Bound Driven

Figure 2: Tree Construction

its partial solution, t will be explored in the next search step, e.g., the node containing v_3 in Figure 2(a).

For each child node with partial solution V_t, we also compute a minimal k-core containing V_t in a DFS which heuristically adds vertices to V_t. In Figure 2(b), a k-core induced by v_0, v_2, v_3, v_9 and v_{10} is computed for $V_t = \{v_0, v_2\}$, and the size upper bound is updated to $s^+ = 5$ if it is smaller than existing upper bound.

Iteratively, the search will return when it finds a k-core satisfying approximation ratio c regarding the size of the optimally-minimum k-core.

Algorithm 1: PSA(G, k, q, c)

```
Input  : $G$ : a graph, $k$ : degree constraint,
         $c$ : approximation ratio, $q$ : query vertex
Output : $R$ : the approximate minimum $k$-core
1  if $q \notin$ the maximal $k$-core of $G$ then return $\emptyset$;
2  $t \leftarrow$ the (root) node of search tree $\mathcal{T}$, where $t.v = q$;
3  $\mathcal{Q}.push(t)$; $R :=$ GetUpper($V_t$);
4  $s^+ := |R|$; $s^-(t) := 1$;
5  while $\mathcal{Q} \neq \emptyset$ do
6      $t \leftarrow \mathcal{Q}$.pop(); //$\mathcal{Q}$ is a priority queue with key on
       $s^-(t)$;
7      for every $u \in N(t.v)$ with $id(u) > id(t.v)$ do
8          $t' \leftarrow$ the child node of $t$, where $t'.v := u$;
9          $s^-(t') :=$ GetLower($V_{t'}$);
10         if $s^-(t') < s^-(t)$ then $s^-(t') := s^-(t)$;
11         if $s^-(t') < s^+$ then
12             $R' :=$ GetUpper($V_{t'}$); $s^+(t') := |R'|$;
13             if $s^+(t') < s^+$ then
14                 $R := R'$; $s^+ := s^+(t')$;
15             $\mathcal{Q}$.push($t'$); attach child node $t'$ to $t$ in $\mathcal{T}$;
16         else
17             $s^-(t') \leftarrow +\infty$ ;
18     $s^- \leftarrow$ smallest key value among nodes in $\mathcal{Q}$ ;
19     if $\frac{s^+}{s^-} \leq c$ then return $R$;
20 return $R$
```

Algorithm 1 shows the pseudo-code of our progressive search algorithm. We use t to denote a tree node in the search tree $\mathcal{T}$. The vertex set V_t of a node t consists of the vertex $t.v$ in t and all the vertices in the ancestor nodes of t. Let s^+ denote the size upper bound of optimally-minimum k-core, and $s^-(t)$ denote the size lower bound of the minimum k-core containing V_t. Similar to the A* search [8, 47], we use a set $\mathcal{Q}$ to denote the the leaf nodes in $\mathcal{T}$ to be visited, where the key of a node t is $s^-(t)$ in ascending order. *GetUpper*(V_t) computes a minimal k-core R containing V_t by heuristics (introduced in Section 3.3). Lines 2-4 initializes the above notations.

In each iteration (Lines 6-19), the node t with the smallest lower bound value $s^-(t)$ is popped at Line 6. To avoid duplicated computations, for current visited vertex $t.v$, we expand it by each neighbor u of a vertex in V_t with $id(u) > id(t.v)$ (Line 7), where $id(u)$ is the identifier of u. At Lines 8-9, for each child node t' of t, *GetLower*(V_t) computes the size lower bound of the minimum k-core containing V_t' (introduced in Section 3.2), where V_t' contains $t'.v$ and all the vertices in V_t. At Line 10, $s^-(t')$ is assigned by $s^-(t)$ if $s^-(t')$ is smaller, because the size lower bound of V_t' should not be smaller than V_t. For the size upper bound $s^+(t')$, at Line 12, we conduct a heuristic search to incrementally add promising vertices to V_t' till it becomes a k-core subgraph, denoted by R', with $s^+(t') = |R'|$. The global upper bound s^+ and current best solution R will be updated by $s^+(t')$ and R' if $s^+(t') < s^+$, since smaller k-core subgraph is preferred (Lines 13-14). Note that a search branch following t' can be safely terminated (Line 17) if $s^-(t') \geq s^+$ because the resulting subgraph derived from V_t' cannot outperform the current best solution R.

The global lower bound s^- is updated as the smallest $s^-(\cdot)$ among all nodes in $\mathcal{Q}$ at Line 18. The algorithm will return if $\frac{s^+}{s^-} \leq c$ is satisfied at Line 19 or the queue is empty. R is returned as an approximate solution of the minimum k-core containing q.

Time complexity. The time cost of Algorithm 1 is $O(l \times (t_l + t_u))$ where l is the number of iterations and t_l (resp. t_u) denotes the computing cost of the lower (resp. upper) bounds at Lines 9 (resp. 12).

Correctness. Every subgraph R' retrieved at Line 12 is a k-core subgraph containing q, and hence the upper bound s^+ is correctly maintained in Algorithm 1. Given the correctness of the lower bound s^-, we have $s^- \leq |R^*| \leq s^+$, where R^* is the optimal solution. Thus, the termination of the search branches at Line 17 is safe as all possible search branches are considered for lower bound computation. When Algorithm 1 terminates, we will return R (best solution obtained so far) with $\frac{s^+}{s^-} \leq c$.

3.2 Lower Bounds Computation

Given a partial solution $P = \{v_1, v_2, ...\}$ and a vertex $v \in P$, we use $d(v)$ to denote the *demand* of v, i.e., the number of extra neighbors (supports) needed from vertices outside of P such that v can satisfy the degree constraint, i.e., $d(v) = \max\{k - deg(v, P), 0\}$. For every vertex $v \in P$ with $d(v) > 0$, we must include at least $d(v)$ vertices in $N(v, G) \setminus P$ such that P can be expanded to a k-core subgraph. Let M^* be the minimal subset of vertices from $V \setminus P$, such that there are at least $d(v)$ neighbors in M^* for every vertex $v \in P$. Clearly, $|P| + |M^*|$ is a size lower bound of any k-core subgraph derived from P, denoted by $L(P)$, because M^* is the minimal set of vertices required to satisfy the degree constraint for vertices in P, without considering the degree constraint for vertices in M^*.

We show that the computation of M^* for a given partial solution P can be converted to a variant of the set cover problem as follows.

Set Multi-Cover Problem [16]. Let U be the universe of n elements with $U = \{e_1, e_2, ..., e_n\}$, and there is a count for each element e_i. We use $C = \{c_1, c_2, \ldots, c_n\}$ to denote the counts of the elements. For a family of m subsets of U, $X = \{S_1, S_2, ..., S_m\}$, where S_i is a subset of U. The goal is to find a set $I \subseteq \{S_1, S_2, ..., S_m\}$, such that every element $e_i \in U$ is covered by at least c_i subsets from I.

Mapping of the problem. Each vertex v in the partial solution P with $d(v) > 0$ corresponds to an element e_i of U. The element count c_i is set to $d(v)$, i.e., the demand of v. Let $N(P)$ denote the neighbor vertices of the partial solution P with $N(P) = (\bigcup_{v \in P \& d(v)>0} N(v, G)) \setminus P$, every neighbor vertex u in $N(P)$ corresponds to a subset S which consists of vertices in P (i.e., U) adjacent to u. By doing this, the optimal solution of the set multi-cover problem, denoted by I^*, corresponds to M^* in the above lower bound computation.

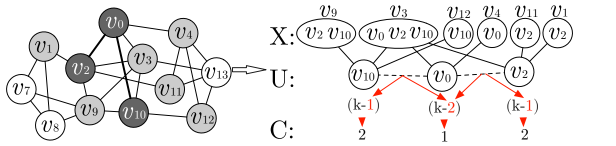


Figure 3: Map to Set Multi-cover

Example 2. *Consider the graph in Figure 1 with $k = 3$, and the partial solution $P = \{v_{10}, v_2, v_0\}$. We show the related part of Figure 1 in the left of Figure 3. For P, we have $d(v_0) = 1$, $d(v_2) = 2$, $d(v_{10}) = 2$, and the neighbor set of P is $N(P) = \{v_9, v_3, v_{12}, v_4, v_{11}, v_1\}$. Mapping to set multi-cover problem, we have $U = \{v_{10}, v_2, v_0\}$, $C = \{2, 2, 1\}$, and $X = \{\{v_2, v_{10}\}, \{v_0, v_2, v_{10}\}, \{v_{10}\}, \{v_0\}, \{v_2\}, \{v_2\}\}$, e.g., the set $\{v_2, v_{10}\}$ is associated with the vertex $v_9 \in N(P)$ as shown in the figure.*

Lower bound computation. For the given partial solution $P = P_u \cup \{v\}$ at Line 9 of Algorithm 1, we can construct an instance of the set multi-cover problem accordingly. We aim to derive a size lower bound of its optimal solution I^*, denoted by L. Then we can use $|P| + L$ as the lower bound since $|P| + L \leq |P| + |I^*| = |P| + |M^*|$. Here, we stress that our focus is the computation of L, not a feasible solution for the set multi-cover problem.

In this subsection, we introduce three approaches to compute L: greedy-based approach (L^g), structure relaxation based approach (L^{sr}) and inclusion-exclusion based approach (L^{ie}).

3.2.1 Greedy-based Lower Bound

In [19], Dobson proposed a greedy algorithm for the set multi-cover problem with an approximation ratio $\ln(\delta)$, where δ is the largest size of the subsets in X. Specifically, the greedy algorithm repeatedly chooses a subset S_i from $X \setminus I$ that covers the largest number of elements in U, which are not yet fully covered by the current sets in I. It stops and returns the chosen subsets in I when every element $e_i \in U$ are covered by at least c_i chosen subsets. Let $|I|$ denote the number of subsets in I when the greedy algorithm terminates, $\frac{|I|}{\ln(\delta)}$ is a *lower bound* of the optimal solution $|I^*|$, i.e., $|I^*| \geq \lfloor \frac{|I|}{\ln(\delta)} \rfloor$. In this paper, we use L^g to denote this greedy heuristic based lower bound for I^*.

Example 3. *Consider the set multi-cover problem in Figure 4, where $U = \{e_1, e_2, e_3, e_4, e_5\}$, $C = \{3, 2, 3, 2, 2\}$,*

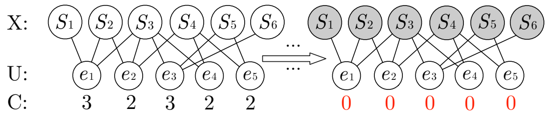


Figure 4: Greedy Based Lower Bound

and $X = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. The elements in a set S_i are the ones connected to S_i as shown in the figure. In a greedy manner, S_3 is the first subset to be chosen, which covers e_1, e_2, e_3, and e_4 first. The count set $C = \{2, 1, 2, 1, 2\}$ is updated. Iteratively, S_4, S_5, S_2, S_1 and S_6 are chosen sequentially. The greedy result is $I = \{S_3, S_4, S_5, S_2, S_1, S_6\}$. Since the maximum size subset in X is $|S_3| = 4$ (i.e., $\delta = 4$), the lower bound is $L^g = \lfloor \frac{|I|}{\ln(\delta)} \rfloor = \lfloor \frac{6}{\ln(4)} \rfloor = 4$.

Discussion. Although the computation of the lower bound can be naturally mapped to the set multi-cover problem, our empirical study indicates that the δ value is usually not small on real-life graphs, which may lead to the poor performance of the greedy-based lower bound. We note that the main focus of the greedy algorithm in [19] is to find a *feasible solution* I for the set multi-cover problem, which meanwhile can derive the lower bound of $|I^*|$. Considering that our purpose is to derive the lower bound of $|I^*|$, in the following two subsections we develop two new techniques which aim to design some heuristic approaches to *directly* derive a tighter lower bound L for $|I^*|$, without considering the feasible solution to the problem.

3.2.2 Structure Relaxation Based Lower Bound.

In this subsection, we introduce the `Structure Relaxation Based Lower Bound`, denoted by L^{sr}.

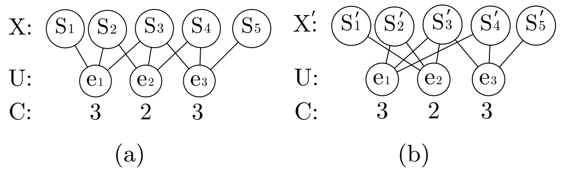


Figure 5: L^{sr} Motivating Example

Motivation. The key idea is to directly obtain a lower bound for $|I^*|$ by re-constructing subsets $\{S_i\}$ in X (i.e., a structure relaxation if we treat set U and X as a bipartite graph as shown in Figure 5). Suppose we have $U = \{e_1, e_2, e_3\}$, $C = \{3, 2, 3\}$ and $X = \{S_1, S_2, S_3, S_4, S_5\}$ in the set multi-cover problem as shown in Figure 5(a). For instance, we have $S_4 = \{e_2, e_3\}$. We have $I^* = \{S_1, S_2, S_3, S_4, S_5\}$ in this example since all S_i must be included, i.e., $|I^*| = 5$. However, suppose we allow each S_i to include arbitrary $|S_i|$ elements (i.e., change with the size constraint), denoted by S'_i. Then we may re-construct X, denoted by X', as shown in Figure 5(b) with $|S_i| = |S'_i|$. For instance, we have $S'_4 = \{e_1, e_3\}$. Now we have $I' = \{S'_2, S'_3, S'_4, S'_5\}$ (i.e., $L = 4$) to cover all elements under the new setting, with $L = |I'| \leq |I^*|$. Note that although I' is not a valid solution for the original set multi-cover problem, $|I'|$ indeed can serve as the lower bound of $|I^*|$.

Lower bound computation. In this subsection, we introduce how to re-construct X such that we can easily derive a valid lower bound for $|I^*|$. Intuitively, we should encourage the set with a large size (i.e., covering power) to include elements with high count values in the re-construction. Algorithm 2 illustrates the details of the structure relaxation based lower bound computation. Initially, Line 1 orders $\{S_i\}$ in X in descending order of their sizes. Meanwhile, the elements are organized by a maximal priority queue Q where the key of an element e_i is its count c_i (Line 2). In Lines 3-11, we will sequentially choose subsets $\{S_i\}$ from X based on their sizes. For each S_i chosen, we will use S_i to cover the first $|S_i|$ elements with the largest count values in Q, denoted by T (Line 5); that is, count c_k will be decreased by one if $e_k \in T$ (Line 7). We remove an element e_k from Q at Line 8 if it has been covered by c_k times, i.e., c_k is decreased to 0. Then the maximal priority queue is updated due to the change of count value at Line 9. Algorithm 2 will be terminated if all elements have been fully covered (Line 11) and l, i.e., $\{S_i\}$ visited so far, will be returned as the lower bound of $|I^*|$ at Line 12.

Algorithm 2: StructureRelaxationLB(U, C, X)

```
Input  : $U$ : a universe of elements,
         $C$ : the counts of the elements,
         $X$ : a family of subsets of $U$
Output : $L^{sr}$: the lower bound of $|I^*|$
1  $l := 0$; put all $S_i \in X$ to $F$ with descending order of $|S_i|$;
2  put all $e_i \in U$ to a maximal priority queue $Q$ with key $c_i$;
3  for $j = 1$ to $|X|$ do
4      $S_i \leftarrow$ the $j$-th subset in $F$; $l := l + 1$;
5      $T \leftarrow$ the top $|S_i|$ elements in $Q$;
6      for each $e_k$ in T do
7          $c_k := c_k - 1$;
8          remove $e_k$ from $Q$ If $c_k = 0$ ;
9          Update $Q$ due to the change of $c_k$;
10     if $c_k = 0$ for every $k \in [1, |U|]$ then
11         Break;
12 return $l$
```

Example 4. *Figure 6 illustrates the procedure of a set multi-cover problem which has $U = \{e_1, e_2, e_3, e_4, e_5\}$, $C = \{3, 2, 3, 2, 2\}$, and $X = \{S_1, S_2, S_3, S_4, S_5, S_6\}$ initially. To compute L^{sr}, the first step is to construct $F = \{S_3, S_4, S_2, S_5, S_1, S_6\}$ and $Q = \{e_1, e_3, e_2, e_4, e_5\}$. S_3 is processed first because it has the most elements. The counts of the top $|S_3|$ elements ($T = \{e_1, e_3, e_2, e_4\}$) in Q decrease by one: $C = \{2, 1, 2, 1, 2\}$. This is followed by the update of the lower bound l and Q. Iteratively, S_4, S_2, S_5, and S_1 are chosen sequentially until $c_x = 0$ for every x from 1 to $|U|$. So, we have $L^{sr} = l = 5$, which is better than $L^g = 4$.*

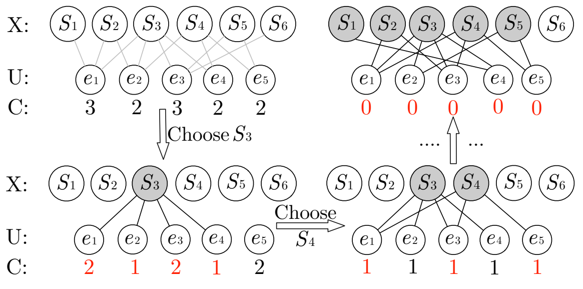


Figure 6: Structure Relaxation Based Lower Bound

Time complexity. The dominated cost is the update of Q, which triggers $\sum_{k=1}^{|U|} c_k$ times. The updating of Q costs $\mathcal{O}(\log(|Q|)$. Thus the time complexity of computing L^{sr} is $\mathcal{O}(\log(|U|) * \sum_{k=1}^{|U|} c_k)$.

Correctness. Note that it is not necessary that $e_k \in S_i$ at Line 5. In this sense, we re-construct S_i to cover different $|S_i|$ elements in U, and hence may end up with an invalid result set I in Algorithm 2 for the set multi-cover problem, i.e., not every element e_k in U will be covered by c_k times by I. Nevertheless, the theorem below suggests that $|I|$ obtained by Algorithm 2 is indeed a lower bound of $|I^*|$, which is sufficient for our problem.

By Theorem 1 we prove that the L^{sr} is a lower bound of I^* for the set multi-cover problem.

Theorem 1. *l obtained in Algorithm 2 is a lower bound of $|I^*|$.*

Proof. We consider the set multi-cover problem with regard to $U = \{e_1, e_2, \dots\}$, $C = \{c_1, c_2, \dots\}$, and $X = \{S_1, S_2, \dots\}$. In order to prove the theorem, we define a relaxed problem $\texttt{Rex}(U, C, H)$ as follows: Given a set of elements $U = \{e_1, e_2, \dots\}$, $C = \{c_1, c_2, \dots\}$, and size constraints $H = \{h_1, h_2, \dots\}$ with $|H| = |X|$ and $h_i = |S_i|$, we want to find a set $I = \{I_{k_1}, I_{k_2}, \dots\}$ where $1 \le k_1 < k_2 < \dots \le |H|$, $I_{k_i} \subseteq U$, and $|I_{k_i}| \le h_{k_i}$, such that e_j is contained in c_j subsets from I and $|I|$ is minimized. Suppose I is the optimal solution of $\texttt{Rex}(U, C, H)$, and I^* is the optimal solution for the set multi-cover problem regarding U, C and X, we have $|I| \le |I^*|$, since $\texttt{Rex}(U, C, H)$ is a relaxation for the set multi-cover problem (each set in I can contain any elements in U with only a size constraint). Next, we prove that our Algorithm 2 obtains the optimal solution I for the problem $\texttt{Rex}(U, C, H)$.

Suppose w.l.o.g. that $h_1 \ge h_2 \ge \dots$ and $c_1 \ge c_2 \ge \dots$, we show that there exists an optimal solution with $k_1 = 1$ i.e., the size of I_{k_1} in I is bounded by h_1. This is because if $k_1 \ne 1$ we can replace k_1 with 1 and the constraint on I_{k_1} is increased, thus we do not obtain a worse solution. Next, we prove that there exists an optimal solution s.t. I_1 contains the first h_1 elements in U (i.e., the top-h_1 elements with the largest c_i values). Suppose there is an optimal solution with $I' = \{I'_1, I'_2, \dots\}$ s.t. in I'_1 there exists an $x < y$ with $e_i \in I'_1$ for $i < x$, $e_x \notin I'_1$ and $e_y \in I'_1$, we prove that we can construct an optimal solution I with $e_i \in I_1$ for $i \le x$. Since C is sorted in non-increasing order, we have $c_x \ge c_y$, therefore, we can always find an I'_j ($j > 1$) that contains e_x but does not contain e_y. So if we move e_x from I'_j to I'_1 and move e_y from I'_1 to I'_j, the constraints on I will not change and in this way we obtain a solution I that is not worse than I' and has $e_i \in I_1$ for $i \le x$. As a result, statement that I_1 contains the first h_1 elements in U holds. Similarly, we can prove that I_2 contains the first h_2 elements in U' by deducing those covered by I_1. Since the selection procedure is the same as that in Algorithm 2, our algorithm obtains the optimal solution for the problem $\texttt{Rex}(U, C, H)$. Therefore, the theorem holds. □

3.2.3 *Inclusion-exclusion Based Lower Bound*

The structure relaxation based approach only considers the size constraint when it re-constructs X, which may end

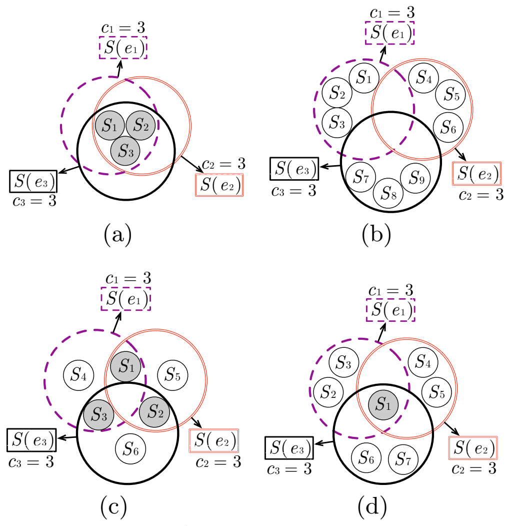


Figure 7: L^{ie} Motivating Example

up with a loose lower-bound. In this subsection, we introduce an `inclusion-exclusion based lower bound`, denoted by L^{ie}.

Motivation. Suppose we have $U = \{e_1, e_2, e_3\}$ and $C = \{3, 3, 3\}$. Let $S(e)$ denote the sets of $\{S_i\}$ in X which contain element e (i.e., $e \in S_i$). Meanwhile, for optimal solution I^*, we use $I^*(e)$ to denote the set of $\{S_i\}$ in I^* which contain the element e. Clearly, for any optimal solution I^* regarding a given X, we must have $|I^*(e_i)| \geq 3$ for elements $\{e_1, e_2, e_3\}$ since $C = \{3, 3, 3\}$. According to the *inclusion-exclusion principle*, we have

$$\begin{aligned}|I^*| &= |I^*(e_1) \cup I^*(e_2) \cup I^*(e_3)| \\ &= |I^*(e_1)| + |I^*(e_2)| + |I^*(e_3)| \\ &- |I^*(e_1) \cap I^*(e_2)| - |I^*(e_1) \cap I^*(e_3)| - |I^*(e_2) \cap I^*(e_3)| \\ &+ |I^*(e_1) \cap I^*(e_2) \cap I^*(e_3)|\end{aligned}$$

Given the fact that $I^*(e) \subseteq S(e)$ for any element $e \in U$, we now investigate the possible value of $|I^*|$ if some knowledge about the overlap size among $\{S(e)\}$ in X is available. If there is no constraint regarding the size of $|S(e_i) \cap S(e_j)|$ $(1 \leq i < j \leq 3)$ and $|S(e_1) \cap S(e_2) \cap S(e_3)|$, we can easily come up with a X such that $|I^*| = 3+3+3-3-3-3+3 = 3$ as shown in Figure 7(a), in which each S_i $(1 \leq i \leq 3)$ covers all three elements. While if we know that $|S(e_i) \cap S(e_j)| = 0$ $(1 \leq i < j \leq 3)$, Figure 7(b) constructs a X such that $|I^*| = 3+3+3-0-0-0+0 = 9$. Note that we have $|I^*(e_i) \cap I^*(e_j)| = 0$ $(1 \leq i < j \leq 3)$ and $|I^*(e_1) \cap I^*(e_2) \cap I^*(e_3)| = 0$ immediately. Using a similar argument, we may have $|I^*| = 3+3+3-1-1-1+0 = 6$ in Figure 7(c) if $|S(e_i) \cap S(e_j)| = 1$ $(1 \leq i < j \leq 3)$ and $|S(e_1) \cap S(e_2) \cap S(e_3)| = 0$. In Figure 7(d), we have $|I^*| = 3+3+3-1-1-1+1 = 7$ given $|S(e_i) \cap S(e_j)| = 1$ $(1 \leq i < j \leq 3)$ and $|S(e_1) \cap S(e_2) \cap S(e_3)| = 1$.

The above example implies that we may come up with a tighter lower bound of $|I^*|$ if we know the overlap sizes of sets among $\{S(e)\}$. For computing efficiency, we only consider the intersection size of pairwise sets, i.e., $|S(e_i) \cap S(e_j)|$ for all $1 \leq i < j \leq n$, in our inclusion-exclusion based lower bound computation.

Lower bound computation. Algorithm 3 gives details of the inclusion-exclusion based lower bound computation. Initially, the elements in U are organized by a maximal priority queue Q, the key of an element e_i is its count c_i (Line 1). We generate a set $S(e_i)$ for each element. $S(e_i)$ denotes the sets of $S \in X$ which cover the element e_i, i.e., $e_i \in S$. In Lines 3-11, we sequentially choose an element e_i from Q based on its count. For each e_i chosen, the other elements in Q, that is, the count c_k will be decreased by Δ, where $\Delta = \min\{S(e_i) \cap S(e_k), c_i\}$. We remove e_k from Q at Line 8 if $c_k \leq 0$, i.e., e_k has been covered c_k times. Then the maximal priority queue is updated due to the change in the count value (Line 11). Algorithm 3 will be terminated if all elements have been fully covered, i.e., Q is empty and l will be returned as the lower bound of $|I^*|$ at Line 12.

Algorithm 3: **Inclusion-Exclusion LB**(U, C, X)

```
Input  : $U$ : a universe of elements need to be covered,
         $C$ : the count of every element in $U$,
         $X$ : a family of subsets of $U$
Output : $L^{ie}$ : Inclusion-exclusion based lower bound
1  put all $e_i \in U$ to a maximal priority queue $Q$ with key $c_i$;
2  generate $S(e_i)$ for each element $e_i \in U$; $l := 0$;
3  while $Q \neq \emptyset$ do
4      $e_i \leftarrow Q$.top(); $Q$.pop(); $l := l + c_i$; $c_i := 0$;
5      for each $e_j \in Q$ do
6          $\Delta := \min\{|S(e_j) \cap S(e_i)|, c_i\}$;
7          if $\Delta \geq c_j$ then
8              remove $e_j$ from $Q$;
9          else
10             $c_j := c_j - \Delta$;
11         Update $Q$;
12 return $l$
```

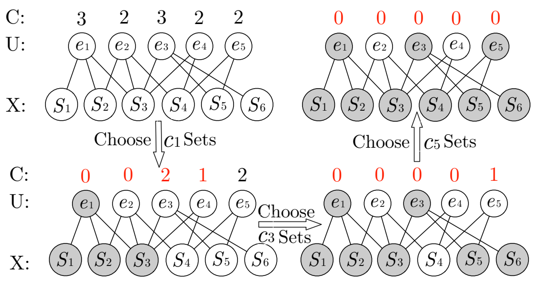


Figure 8: Inclusion-exclusion Based Lower Bound

Example 5. *Consider the set multi-cover problem in Figure 8, where $U = \{e_1, e_2, e_3, e_4, e_5\}$, $C = \{3, 2, 3, 2, 2\}$, and $X = \{S_1, S_2, S_3, S_4, S_5, S_6\}$, where $S_i \subseteq U$. Let $S(e_i)$ denote the sets of $\{S_j\}$ in X that cover the element e_i (i.e., $e_i \in S_j$), shown as $S(e_1) = \{S_1, S_2, S_3\}$ in the figure. We construct the priority queue $Q = \{e_1, e_3, e_2, e_4, e_5\}$. Here e_1 is processed first and $l = 0 + c_1 = 3$. The set C is updated accordingly: $\Delta_2 = \min\{|S(e_1) \cap S(e_2)|, c_1\} = 2$. Thus $\Delta_2 \geq c_2$, e_2 is removed from Q. We have $C = \{0, 0, 2, 1, 2\}$ and $Q = \{e_3, e_5, e_4\}$ after completing the first round. Next, e_3 is processed and l is updated to $l = 3 + c_3 = 5$. Similarly, we have $C = \{0, 0, 0, 0, 1\}$ and $Q = \{e_5\}$ after finishing this round. Then e_5 is processed in the next step, and $l = 6$, followed by $c_5 = 0$ and $Q = \emptyset$. As a result, the lower bound is $L^{ie} = l = 6$.*

Time complexity. The dominant cost of Algorithm 3 is the computation of $S(e_i) \cap S(e_j)$ at Line 6. With the help of the data structure HashMap in c++ STL, we can get $|S(e_i) \cap S(e_j)|$ in $|X| \times \mathcal{O}(1)$ time, where $\mathcal{O}(1)$ is the time complexity of one search in HashMap. As each $S(e_i)$ can be pre-sorted, the cost is $O(|U|^2|X|)$.

Space complexity. As each set $S(e_i)$ takes $\mathcal{O}(|X|)$ space at most. Each HashMap takes $\mathcal{O}(|X|)$ space. The space complexity is $O(|U||X|)$ at most.

Correctness. The following theorem shows the correctness of the inclusion-exclusion based lower bound computation.

Theorem 2. *l obtained in Algorithm 3 is a lower bound of $|I^*|$.*

Proof. Suppose the set multi-cover problem is related to $U = \{e_1, e_2, e_3, \dots, e_h\}$, $C = \{c_1, c_2, c_3, \dots\}$, and $X = \{S_1, S_2, \dots\}$, and the optimal solution for this problem is I^*. We use $S(e_i)$ to denote the sets of $\{S_j\}$ in X which contain element e_i (i.e., $e_i \in S_j$).

We prove the theorem by induction. Suppose the theorem holds for any U with $h-1$ elements, we prove the theorem holds for any U with h elements. We start from an element $e_1 \in U$: Suppose in the optimal solution I^*, we use l_1^* subsets to cover e_1, apparently, we have $l_1^* \geq c_1$. Let l_0^* be the number of subsets to cover all other elements in $U \setminus \{e_1\}$ in the optimal solution. So we have $|I^*| = l_1^* + l_0^*$. Suppose after selecting l_1^* subsets $I^*(e_1)$ to cover e_1, other elements in U need to be covered $C^* = \{c_2^*, c_3^*, \dots\}$ times. We have $c_2^* = c_2 - |I^*(e_1) \cap S(e_2)|$, $c_3^* = c_3 - |I^*(e_1) \cap S(e_3)|$, ...

For Algorithm 3, firstly, we select l_1 subsets to cover e_1, i.e., $l_1 = c_1$, apparently, $l_1^* \geq l_1$. Let l_0 be the number of subsets that we get from Algorithm 3 to cover all elements in $U \setminus \{e_1\}$. Thus, $l = l_1 + l_0$. Suppose after selecting l_1 subsets to cover e_1, the elements in $U \setminus \{e_1\}$ need to be covered by $C' = \{c'_2, c'_3, \dots\}$ times, and according to the algorithm, we have $c'_2 = c_2 - |S(e_1) \cap S(e_2)|$, $c'_3 = c_3 - |S(e_1) \cap S(e_3)|$, ... Since $I^*(e_1) \subseteq S(e_1)$, we have $c_2^* \geq c'_2$, $c_3^* \geq c'_3$, ...

According to the induction condition, l_0 is the lower bound of the problem with regard to $U' = \{e'_2, e'_3, \dots, e'_h\}$ with $S(e'_i) = S(e_i) \setminus S(e_1)$, and $C' = \{c'_2, c'_3, \dots\}$. We know that l_0^* is the optimal solution to the problem with regard to $U' = \{e'_2, e'_3, \dots, e'_h\}$ with $S(e'_i) = S(e_i) \setminus S(e_1)$, and $C^* = \{c_2^*, c_3^*, \dots\}$. Therefore, l_0 is also a lower bound of the problem with regard to U' and C^*, i.e., $l_0 \leq l_0^*$.

Consequently, we have $l = l_1 + l_0 \leq l_1^* + l_0^* = |I^*|$. The theorem holds. □

3.2.4 Putting the lower bounds together

Intuitively, we can use three lower bounds L^g, L^{sr} and L^{ie} together and choose the maximal one as the lower bound at Line 9 of Algorithm 1, i.e., $L = \max(L^g, L^{sr}, L^{ie})$. However, as shown in our empirical study, the greedy-based lower bound L^g is not useful because of its poor performance in terms of pruning power and computing cost. Thus, we only use the structure relaxation based lower bound L^{sr} and the inclusion-exclusion based lower bound L^{ie} in our implementation of *PSA*, i.e., $L = \max(L^{sr}, L^{ie})$.

3.3 Upper Bound Computation

Any k-core subgraph containing query q can immediately serve as an upper bound of the optimal solution. Thus, we may continuously maintain an upper bound s^+ at Line 14 of Algorithm 1 by keeping the smallest size k-core containing q obtained so far. In this subsection, we present a heuristic algorithm to find a k-core starting from a partial solution P in Algorithm 1. The key idea is to incrementally expand the partial solution based on the *importance* of the vertices. Particularly, we use the concept of the *onion layer* proposed in [52] to prioritize the access order. The onion layer can be regarded as a refinement of the k-shell.

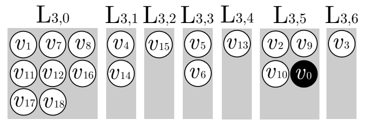


Figure 9: Onion Layer Structure, $k = 3$

Given a graph G, we use $C_k(G)$ to denote the maximal k-core of G. The **coreness** of a vertex u, denoted by $cn(u)$, is the largest value of k such that $u \in C_k(G)$. The k-shell of G, denoted by $S_k(G)$, is the set of vertices with core number k, i.e., $S_k(G) = C_k(G) \setminus C_{k+1}(G)$.

Coreness has been used to measure the importance/influence of the vertices in a variety of applications (e.g., [11, 52]). Given the k-shell $S_k(G)$ of graph G, where every vertex in $S_k(G)$ has the same coreness value k, the onion layer further partitions these vertices into different layers using an *onion-peeling-like* algorithm. Given the maximal k-core $C_k(G)$, we use the $L_{k,0}$ to denote the vertices in $C_k(G)$ which do not satisfy the $k+1$ degree constraint, i.e., $L_{k,0} = \{u \mid deg(u, C_k(G)) < k+1\}$. Then we use $L_{k,1}$ to denote the vertices which do not satisfy the $k+1$ degree constraint after removing the vertices in $L_{k,0}$. The above procedure is repeated until $L_{k,l+1}$ is empty, and we have $S_k(G) = \cup_{0 \leq j \leq l} L_{k,j}$. Through this procedure, each vertex $u \in G$ will be assigned to a unique onion layer $L_{k,j}$. Given any two vertices u and v in onion layers L_{k_1,j_1} and L_{k_2,j_2} respectively, we say u has *higher onion layer* than v if $k_1 > k_2$ or $j_1 > j_2$ given $k_1 = k_2$.

Example 6. *Consider the graph in Figure 1 with $k = 3$. The graph itself is a 3-core. We start to peel the graph: firstly, $L_{3,0} = \{v_1, v_7, v_8, v_{11}, v_{12}, v_{16}, v_{17}, v_{18}\}$, because these vertices do not satisfy the $k+1$ degree constraint. After removing the vertices in $L_{3,0}$, v_4 and v_{14} do not satisfy the $k+1$ degree constraint, so we have $L_{3,1} = \{v_4, v_{14}\}$; Repeating the above procedure, we have $L_{3,2}, \dots, L_{3,6}$ as shown in Figure 9 because $v_{13} \in L_{3,4}$ and $v_1 \in L_{3,0}$, v_{13} have a higher onion layer than v_1.*

Onion Layer Based Upper Bound. Algorithm 4 illustrates details of the onion-layer based upper bound computation. In Lines 1-6, we sequentially assess the vertices $\{v\}$ in partial solution P which still do not satisfy the degree constraint, i.e., $deg(v, P) < k$. For each vertex v, we add $r = k - \deg(v, P)$ neighbors of v that are not in P with the highest *onion-layer* values, denoted by X (Line 4). The partial solution P will be updated accordingly with $P := P \cup X$ (Line 5). At Line 6, we ensure the resulting subgraph is a small k-core subgraph by removing the redundant vertices in P. Note that we say a vertex $u \neq q$ is redundant if every neighbor v of u in P has $deg(v, P) \geq k+1$. Algorithm 4 will be terminated if P grows to a k-core subgraph (Line 1), and P is a minimal k-core since we remove the redundant vertices. Then P is returned at Line 7, which can serve as the upper bound of the minimum k-core containing q. Note that the construction of onion layer is independent to the query vertex q, which can be pre-computed in $O(m+n)$ time as shown in [52].

Algorithm 4: **L-Greedy**(P)

```
  Input  : $P$ : the partial solution
  Output : a $k$-core subgraph containing $P$
1 while $P$ is not a $k$-core subgraph do
2     $v \leftarrow$ the vertex in $P$ with $deg(v,P) < k$ ;
3     $r := k - deg(v,P)$;
4     $X \leftarrow r$ neighbors of $v$ not in $P$ with highest
      onion-layer values;
5     $P := P \cup X$ ;
6     remove redundant vertices from $P$;
7 return $P$
```

Time complexity. For each vertex v accessed at Lines 1-6, it takes $O(d_{max}\log(d_{max}))$ time to find the set X at Line 4, where d_{max} is the highest degree in graph G. Meanwhile, we only need to consider the redundancy for its neighbor vertices, with cost $O(d^2_{max})$. Therefore, the time complexity of Algorithm 4 is $O(|U| \times d^2_{max})$.

Correctness. The correctness of the algorithm is immediate because the set P returned is a k-core subgraph.

3.4 Processing Multiple Query Vertices

In addition to one simple query vertex q, it is interesting to consider to find a minimum k-core subgraph containing a set of query vertices $Q = \{q_1, q_2, ...\}$, which is also NP-hard. The *PSA* algorithm proposed in this paper can be easily extended to tackle this problem by enforcing every partial solution to contain all query vertices. The computation of the lower and upper bounds are the same. Algorithm 1 can be adjusted as follows.
(i) Replace the input with "$PSA(G,k,Q,c)$";
(ii) Replace Line 1 with "**if** $\exists q \in Q$ *and* $q \notin$ the maximal k-core of G **then**";
(iii) Replace the initialization (Lines 2-4) part with "(1) $u \leftarrow$ the vertex in Q with the largest id; $t \leftarrow$ the (root) node of search tree $\mathcal{T}$; $t.v = u$; (2) $V_t \leftarrow Q$ with increasing order of $id(x)$; $s^-(t) :=$ **GetLower**(V_t); $\mathcal{Q}.push(t)$; and (3) $R :=$ **GetUpper**(V_t); $s^+ := |R|$".

Time complexity. The time cost of Algorithm 1 is $\mathcal{O}(l \times (t_l + t_u))$, where l is the number of iterations. The main difference between the multiple query vertices and one query vertex is the number of iterations involved and the computing cost of lower (resp. upper) bounds is the same at each iteration. Thus, the time cost of updated Algorithm 1 for multiple query vertices is also $\mathcal{O}(l \times (t_l + t_u))$.

4. EXPERIMENTAL EVALUATION

Algorithms. We mainly evaluate the following algorithms:

- `S-Greedy`: The state-of-the-art technique [9] for the problem of minimum k-core search, which is outlined in Section 2.2. Authors in [9] kindly provided the source code implemented by Java and we rewrite it in C++ for fair comparison of search time.
- `L-Greedy`: A greedy algorithm which only uses the upper bound technique proposed in Section 3.3; that is, Algorithm 4 is invoked with partial solution $P = \{q\}$ and the resulting k-core subgraph will be returned.
- `PSA`: The progressive search framework (Algorithm 1) proposed in Section 3.1, equipped with two lower bound techniques L^{sr} (Section 3.2.2) and L^{ie} (Section 3.2.3) and the upper bound technique *L-Greedy*. Note that *PSA* does not use the greedy-based lower bound L^g (Section 3.2.1) due to its poor performance.
- `PSA-S`: The progressive search framework equipped with *S-Greedy* as the upper bound and L^g (Section 3.2.1) as the lower bound.
- `PSA-L`: The progressive search framework equipped with *L-Greedy* as the upper bound and L^g as the lower bound.

Table 2: Statistics of Datasets

Dataset	Nodes	Edges	d_{avg}	d_{max}	k_{max}
`Email`	36,692	183,831	5.01	1383	43
`Epinion`	75,879	405,740	5.3	3044	67
`Gowalla`	99,563	456,830	21.9	9967	43
`DBLP`	510,297	1,186,302	2.3	340	25
`Yelp`	249,440	1,781,885	7.1	3812	105
`Yeast`	12,782	2,007,134	157.1	3322	277
`YouTube`	1,134,890	2,987,624	2.6	28754	51
`Google`	875,713	4,322,051	4.9	6332	44
`Wiki`	2,394,385	4,659,565	1.9	100029	131
`Flickr`	1,715,255	15,555,041	9.1	27236	568
`UK2002`	18,483,186	292,243,663	15.8	194955	943
`Webbase`	118,142,155	1,019,903,190	8.6	816127	1506

Table 3: Comparison for CS solutions on `Gowalla`

Metrics	diameter	degree	density	CC	avg. size	Engage
k-`Core`	8.12	24.45	0.001	0.32	23196.29	35%
k-`Truss`	4.61	6.92	0.63	0.33	13020.85	43%
k-`Ecc`	7.62	25.18	0.02	0.33	22561.75	36%
k-`Clique`	5.18	15.49	0.38	0.63	15775.21	42%
`Graph Clustering`	13.87	8.59	0.11	0.32	81232.53	30%
`Min` k-`Core`	3.39	10.09	0.37	0.57	52.02	49%

Datasets. Twelve real-life networks are deployed in our experiments. `Yeast` is a protein interaction network [39]. `Flickr` is the network for sharing content [32]. `UK2002` and `Webbase` are downloaded from WebGraph [12]. `Yelp` is downloaded from Yelp [48]. The remaining datasets are downloaded from SNAP [26]. `Gowalla` is a location-based social network. Vertices without check-ins are removed. We transfer directed edges to undirected edges. `DBLP` is a co-author network, where each vertex represents an author and there is an edge between two authors iff they have co-authored at least 3 papers. Table 2 shows the statistics of all the datasets.

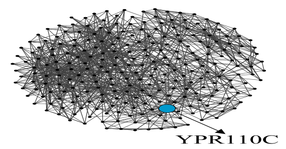
Parameters and query generation. The default values for k and c are 10 and 1.8 respectively. In the experiments, k varies from 5 to 25 and c varies from 4.0 to 1.6. Each query vertex is randomly selected from the k-core. In each test, 100 queries are randomly generated and their average result size or search time is reported. Each computation is terminated if it cannot finish within half an hour.

All programs are implemented in C++ and compiled with g++. All experiments are conducted on a machine with Intel Xeon 2.3GHz CPU running Redhat Linux.

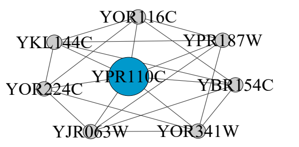
4.1 Effectiveness

4.1.1 *Comparison with community search methods*

We compare several representative algorithms introduced in the survey of community search (CS) [21] (i.e., k-Core, k-Truss, k-Ecc, k-Clique) and the structural graph clustering [42] with our *PSA* on `Gowalla` dataset. For each query vertex v, the setting of the input k is same to the evaluation in the survey [21], e.g., k is the coreness of v for k-core

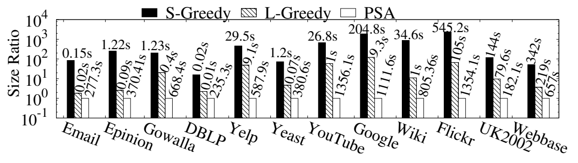


(a) S-Greedy

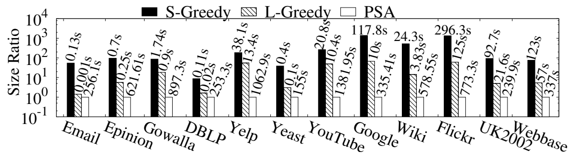


(b) PSA

Figure 10: Case Studies on Yeast, $k = 5$



(a) Single Query, $|Q| = 1$



(b) Multi Queries, $|Q| = 4$

Figure 11: Average Result Size, $k = 10$, $c = 1.8$

and k is the trussness of v for k-Truss. For the clustering algorithm, the similarity threshold for two nodes is set as $\epsilon = 0.6$ (the default value in [42]) and the threshold for the cardinality of ϵ-neighborhood is set as $\mu = cn(v)$. To evaluate the result, we use all the quality metrics used in the survey [21] and two additional important metrics: diameter, average degree, density, clustering coefficient (CC) and average size of the communities, as well as the user engagement (Engage). Engage is the proportion of active users in a community where a user is active if the user has at least 1 check-in during 2018-08-04T00:00:01 and 2018-08-10T23:59:59. Note that the chosen query vertices are also active during this period.

Table 3 shows the scores of the communities returned by each evaluated method. For *PSA*, the average size of our communities is much smaller than the others where the size is reasonable in real-life. Besides, our k-cores have the best diameter and Engage scores, benefiting from the k-core constraint and the small community size.

4.1.2 Case studies

We compare *S-Greedy* and *PSA* ($k = 5$, $c = 1.8$) on `Yeast` to show different results from two approaches. As shown in Figure 10(a), *S-Greedy* returns a large k-core subgraph where some vertices are not closely connected. Besides, only 32 of the 254 proteins detected by *S-Greedy* have at least one common function with the query protein YPR110C. In Figure 10(b), *PSA* identifies 7 nearby proteins of YPR110C where each of these proteins has at least one common function with the query protein.

To further evaluate the homophily property of vertices in a minimal k-core subgraph, we compare it with the community search methods introduced in Section 4.1.1, in terms of the common protein functions in the communities. Specifically, we say a community C is homogeneous iff the query protein has the function f which is the most common function among all the proteins in C. It can be verified by the enrichment analysis in David [24]. Table 4 reports the percentage of homogeneous protein communities over all detected communities for each method.

Table 4: Percentage of Homogeneous Communities

Alg.	k-Core	k-Truss	k-Ecc	k-Clique	Graph Clustering	Min k-Core
%	61%	70%	61%	86%	52%	95%

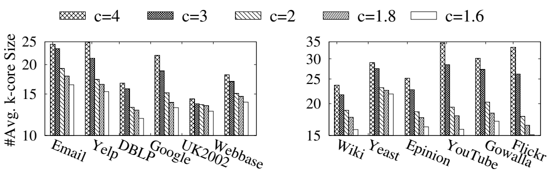


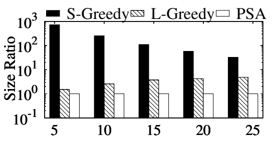
Figure 12: Effect of c, $k = 10$

4.1.3 Evaluation on result size

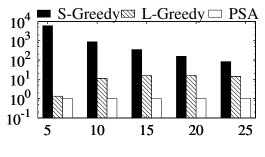
Evaluation on different graphs. Figure 11 reports the average size ratios of k-core subgraphs returned by *S-Greedy*, *L-Greedy* and *PSA* on all the datasets, where $k = 10$ and $c = 1.8$. The average size of the k-cores returned by *PSA* is regarded as the base value, i.e., 1. The sizes of the k-core subgraphs found by *S-Greedy* and *L-Greedy* are much larger than *PSA*. Regarding result size, *L-Greedy* outperforms *S-Greedy* due to better heuristics. In Figure 11, we also mark the running time of each setting. Given the benefit of consistently retrieving small size k-cores, it is cost-effective to apply the *PSA* algorithm. Given a larger dataset, we observe that the time cost of *PSA* is not necessarily higher. This is because, although the increase of graph size leads to larger search space, *PSA* may have a larger possibility to fast complete the query on a $(k-1)$-clique in the search, in which we only need to explore 1-hop neighbors.

Varying the approximate ratio c. Figures 12 shows the average sizes of k-core subgraphs returned by *PSA* on all the datasets when c varies from 4.0 to 1.6. Note that the size of a k-core returned by *S-Greedy* or *L-Greedy* does not have a approximation guarantee, and is irrelevant with the value of c. As expected, the average size decreases when c became smaller. It implies a reasonable ratio can be applied to trade-off the efficiency and the quality.

Varying the degree constraint k. Figure 13 reports the average size of a k-core returned by *S-Greedy*, *L-Greedy* and *PSA*, by varying k from 5 to 25 on `Epinion` and `Wiki`. The margin of *PSA* and *S-Greedy* becomes smaller when the input of k grows, because the size of a k-core from *S-Greedy* is related to the size of maximal k-core which decreases with a larger k, and our *PSA* performs well on different k. The margin of *PSA* and *L-Greedy* becomes larger when the input of k grows, because it is harder for *L-Greedy* to constraint the size of returned k-core given a larger k.

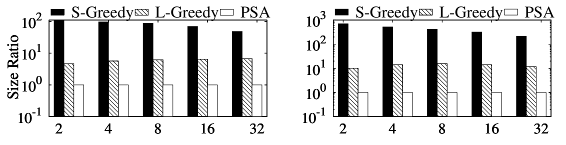


(a) Epinion



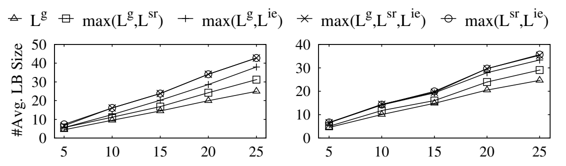
(b) Wiki

Figure 13: Effect of k, $c = 1.8$



(a) Epinion (b) Wiki

Figure 14: Effect of $|Q|$, $k = 10$



(a) Epinion (b) UK2002

Figure 15: Effect of Lower Bounds, $c = 1.8$

Varying the size of query set $|Q|$. Figure 14 shows the results of three algorithms by varying the size of query set from 2 to 32 on `Epinion` and `Wiki`. The average size of a k-core returned by *L-Greedy* or *S-Greedy* is much larger than *PSA*. As expected, the minimal k-core found by *PSA* becomes larger given more query vertices. Since the minimal k-core returned by *S-Greedy* is more related to the size of maximal k-core, the margin between *PSA* and *S-Greedy* becomes smaller with a larger query set.

4.2 Efficiency

Evaluating lower bounds. Figure 15 shows the size of different lower bounds derived from the lower bound technique L^g, L^{sr} and L^{ie}. Note that a large value is preferred in the evaluation of the lower bound. The performance of the single L^g is beaten by all the other methods for all the evaluated settings. This is not surprising because it is difficult for a greedy algorithm to get a good approximation due to the factor $\ln(\delta)$, where δ is the largest size of the subsets in X (Section 3.2). The bound $\max\{L^g, L^{sr}\}$ is to choose the larger one from L^g and L^{sr}, which outperforms L^g. As analyzed in Section 3.2, sometimes the lower bound derived by L^{sr} is not tight enough, while it can improve the result from L^g. We further evaluate $\max\{L^g, L^{sr}, L^{ie}\}$ which adds L^{ie} to $\max\{L^g, L^{sr}\}$. Figure 15 shows that $\max\{L^g, L^{sr}, L^{ie}\}$ achieves the best results. It implies that L^{ie} produces a tighter lower bound, compared with L^{sr}. The reason is that L^{ie} considers more information of the subsets in X. To further validate the effectiveness of different bounds, we evaluate $\max\{L^g, L^{ie}\}$. Compared with $\max\{L^g, L^{sr}\}$, $\max\{L^g, L^{ie}\}$ produces a tighter bound. Considering the computing cost of L^g is expensive, we also investigate the performance of $\max\{L^{sr}, L^{ie}\}$, which shows similar performance with $\max\{L^g, L^{sr}, L^{ie}\}$. Because the time cost of $\max\{L^{sr}, L^{ie}\}$ is dominated by L^{ie}, we deploy $\max\{L^{sr}, L^{ie}\}$ in our *PSA* algorithm.

Evaluating Memory Cost. Figure 16 shows the memory cost of *S-Greedy*, *L-Greedy*, and *PSA*, respectively. The memory cost gradually grows with a larger graph, because the graph data dominates the memory cost. For instance, when the edge number of a graph is larger than $3M$, the memory cost of three methods is almost the same.

Evaluation on different graphs. Figure 17 shows the time cost of *PSA-S*, *PSA-L* and *PSA*, respectively, when $k =$ 10 and $c = 1.8$. *PSA-L* always outperforms *PSA-S* because the upper bound technique *L-Greedy* can better reduce the size of the resulting k-core subgraphs, compared with *S-Greedy*. As shown in Figure 15, the combination of L^{sr} and L^{ie} performs better than L^g. Thus, in Figure 17, *PSA* outperforms *PSA-L* in all the settings, benefiting from the superior upper bound technique (*L-Greedy*) and two lower bound techniques L^{sr} and L^{ie}.

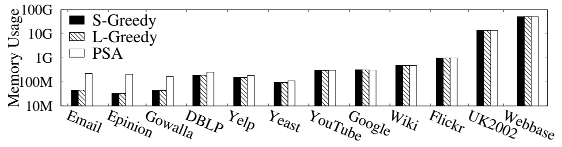
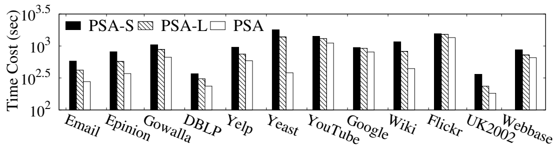
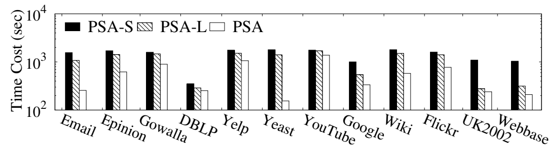


Figure 16: Memory Cost, $k = 10$, $|Q| = 1$



(a) Single Query $|Q| = 1$

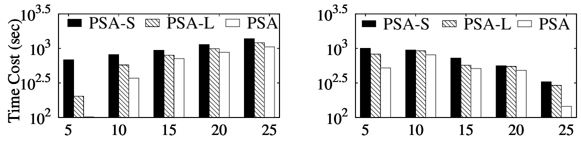


(b) Multi Queries $|Q| = 4$

Figure 17: Performance of PSA, $k = 10$, $c = 1.8$

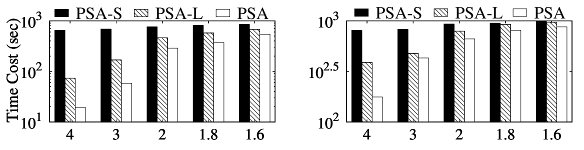
Varying degree constraint k. Figure 18 shows the average running time of the three algorithms when k varies from 5 to 25. Similar to Figure 17, *PSA-L* always outperforms *PSA-S*. The increase of degree threshold k may affect the time cost in two ways: (1) more vertices need to be explored s.t. the runtime increases; and (2) the number of candidate vertices decreases because the size of maximal k-core becomes smaller s.t. the runtime decreases. In Figure 18(a), (1) dominates the effect on different k. In Figure 18(b), when $k > 10$, (2) becomes the major effect for different k values. Thus, the trend of a larger input of k is different on different settings.

Varying the approximate ratio c. Figure 19 shows that *PSA* is the most efficient among the three algorithms, and *PSA-L* consistently outperforms the *PSA-S*, given different values of c. With a relatively large c, *PSA* can find a proper result in a reasonable time. For example, when $c = 4.0$, the time cost of *PSA-L* or *PSA* on `Epinion` is less than 100 seconds. The size of the resulting k-core can be smaller if we allow more time cost for the algorithm, which leads to a trade-off between result quality and efficiency.



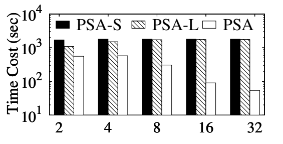
(a) Epinion (b) Google

Figure 18: Varying k, $c = 1.8$



(a) Epinion (b) Google

Figure 19: Varying c, $k = 10$



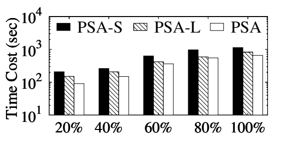


Figure 20: Effect of $|Q|$ **Figure 21: Varying $|V|$**

Varying the query size $|Q|$. Figure 20 shows that *PSA* outperforms *PSA-L* and *PSA-S* when $|Q|$ varies from 2 to 32 on `Wiki`. *PSA-S* performs the worst due to the limited performance of its upper/lower bound techniques. Given a larger query set, both the upper bound and the lower bound values may increase while usually the lower bound has a larger increase ratio due to its smaller value. Thus, the runtime of *PSA* decreases with a larger query set Q.

Scalability evaluation on `Webbase`. We randomly sample vertices from 20% to 100% in the original graph. For each sampled vertex set, we obtain the induced subgraph of the vertex set as the input data. Figure 21 shows the running time of *PSA-S*, *PSA-L* and *PSA* under different percentages, respectively. It shows that the growth of running time of the algorithms are not very significant with respect to the growth of the graph size. Although the increase of graph size leads to larger search space, we may have a higher chance to quickly end up the query with a $(k-1)$-clique in the search, in which we only need to explore 1-hop neighbors.

Evaluating different kinds of queries. We compare the efficiency of our *PSA* algorithm for three kinds of query vertices: *Hubs*, *Nonhubs*, and *Random*. *Hubs* contains the hub vertices which are the top 10% vertices of the k-core w.r.t the vertex degree in k-core. *Nonhubs* contains the other 90% vertices in the graph. *Random* is to randomly choose query vertices from the k-core. We randomly select 100 vertices from each category. Figure 22 shows that the running time for queries on *Hubs* is much faster, because the query on hub vertices is more likely to quickly end up with a clique, in which only direct neighbors need to be accessed. Queries on *Random* are faster than queries on *Nonhubs* because there are both hub and non-hub vertices in *Random*.

5. RELATED WORK

k-Core. The model of k-core [35] is widely used in many applications, such as social contagion [40], community discovery [50, 51], user engagement [52, 54], hierarchical structure analysis [4], influence studies [25], dense subgraph problems [6, 15], graph visualization [3, 56], event detection [31], anomaly detection [36], and protein function prediction [2, 46]. Batagelj and Zaversnik [10] design a linear in-memory algorithm to derive core numbers of vertices in a graph. Wen et *al.* [43] and Cheng et *al.* [17] present I/O efficient algorithms for core decomposition. Locally estimating core

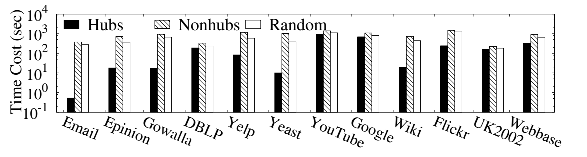


Figure 22: Different Query Types, $k = 10$, $c = 1.8$

numbers is studied in [33]. Several papers study the dynamic of k-core against edge addition or deletion [1, 27, 55, 57].

Cohesive subgraph search. The aim of the cohesive subgraph search is to find subgraphs that contain a set of query nodes, in which the nodes are intensively connected to the others with respect to a particular goodness metric. k-core is one of the popular model to capture the structural cohesiveness of a subgraph. Some existing works [37, 18, 9] have been proposed to find the small size k-core subgraphs with different search heuristics. However, they cannot guarantee the size of the returned k-core subgraph. Thus, the k-core subgraph identified may be very large in practice.

Recently, Ma et *al.* [29] studied the size constrained k-core search problem, which is to find a k-core with exact size h and the smallest closeness among all size h subgraphs containing a query vertex on a weighted graph. This model is suitable to applications when the cohesive subgroup size is pre-fixed. Note that it is infeasible to apply this technique to minimum k-core search since the problem itself is also NP-hard and it is difficult to find a proper size to start with.

In addition to k-core, other cohesive models such as k-truss, k-clique and k-ecc are investigated in the context of local community search [23, 49, 14]. Please refer to [21] for a recent survey.

6. CONCLUSION AND DISCUSSION

In this paper, we investigated the problem of the minimum k-core search which aims to find the smallest k-core subgraph containing the query vertex set. Some existing studies on this problem are based on greedy heuristic following a variety of scoring functions, while they cannot provide any theoretical guarantee on the quality of the results. Motivated by these issues, we propose a progressive search algorithm *PSA*, based on novel lower and upper bound techniques. The proposed algorithm achieves a good trade-off between the quality of the result and the search time. Our extensive experiments on 12 real-life graphs demonstrate the effectiveness and efficiency of our proposed techniques.

In addition to minimal k-core, our progressive framework and lower/upper bound techniques may shed light on the search of other minimal cohesive subgraphs. For instance, the computing framework can be adopted to find the minimal k-truss or k-ecc. The size bounds of k-core proposed in this paper may inspire the bounds for other models to reduce the search space as well.

7. ACKNOWLEDGMENTS

Ying Zhang is supported by ARC DP180103096 and FT170100128. Lu Qin is supported by ARC DP160101513. Wenjie Zhang is supported by ARC DP180103096. Xuemin Lin is supported by 2018YFB1003504, NSFC61232006, ARC DP180103096 and DP170101628.

8. REFERENCES

[1] H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and Ö. Ulusoy. Distributed k-core view materializationand maintenance for large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2439–2452, 2014.

[2] M. Altaf-Ul-Amine, K. Nishikata, T. Korna, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori, et al. Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics*, 14:498–499, 2003.

[3] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*, pages 41–50, 2005.

[4] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *NHM*, 3(2):371–393, 2008.

[5] O. Amini, D. Peleg, S. Pérennes, I. Sau, and S. Saurabh. On the approximability of some degree-constrained subgraph problems. *Discrete Applied Mathematics*, 160(12):1661–1679, 2012.

[6] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *WAW*, pages 25–37, 2009.

[7] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.

[8] S. Bandi and D. Thalmann. Path finding for human motion in virtual environments. *Comput. Geom.*, 15(1-3):103–127, 2000.

[9] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *Data Min. Knowl. Discov.*, 29(5):1406–1433, 2015.

[10] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[11] K. Bhawalkar, J. M. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma. Preventing unraveling in social networks: The anchored k-core problem. *SIAM J. Discrete Math.*, 29(3):1452–1475, 2015.

[12] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.

[13] P. M. Cahusac, E. T. Rolls, Y. Miyashita, and H. Niki. Modification of the responses of hippocampal neurons in the monkey during the learning of a conditional spatial response task. *Hippocampus*, 3(1):29–42, 1993.

[14] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang. Index-based optimal algorithms for computing steiner components with maximum connectivity. In *SIGMOD*, pages 459–474, 2015.

[15] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, pages 84–95, 2000.

[16] C. Chekuri, K. L. Clarkson, and S. Har-Peled. On the set multi-cover problem in geometric settings. *CoRR*, abs/0909.0537, 2009.

[17] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[18] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.

[19] G. Dobson. Worst-case analysis of greedy heuristics for integer programming with nonnegative data. *Math. Oper. Res.*, 7(4):515–531, 1982.

[20] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[21] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, Jul 2019.

[22] D. O. Hebb. The organization of behavior. a neuropsychological theory. 1949.

[23] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[24] X. Jiao, B. T. Sherman, D. W. Huang, R. M. Stephens, M. W. Baseler, H. C. Lane, and R. A. Lempicki. DAVID-WS: a stateful web service to facilitate gene/protein list analysis. *Bioinformatics*, 28(13):1805–1806, 2012.

[25] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse. Identification of influential spreaders in complex networks. *Nature physics*, 6(11):888, 2010.

[26] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[27] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014.

[28] X. Luo, M. Andrews, Y. Song, and J. Aspara. Group-buying deal popularity. *Journal of Marketing*, 78(2):20–33, 2014.

[29] Y. Ma, Y. Yuan, F. Zhu, G. Wang, J. Xiao, and J. Wang. Who should be invited to my party: A size-constrained k-core problem in social networks. *J. Comput. Sci. Technol.*, 34(1):170–184, 2019.

[30] F. D. Malliaros and M. Vazirgiannis. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*, pages 469–478, 2013.

[31] P. Meladianos, G. Nikolentzos, F. Rousseau, Y. Stavrakas, and M. Vazirgiannis. Degeneracy-based real-time sub-event detection in twitter stream. In *ICWSM*, pages 248–257, 2015.

[32] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM*, pages 29–42, 2007.

[33] M. P. O'Brien and B. D. Sullivan. Locally estimating core numbers. In *ICDM*, pages 460–469, 2014.

[34] M. Rubinov and O. Sporns. Complex network measures of brain connectivity: Uses and interpretations. *NeuroImage*, 52(3):1059–1069, 2010.

[35] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[36] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowl. Inf. Syst.*, 54(3):677–710, 2018.

[37] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*, pages 939–948, 2010.

[38] O. Sporns. *Networks of the Brain.* MIT press, 2010.

[39] D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, L. J. Jensen, and C. von Mering. STRING v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic Acids Research*, 47:D607–D613, 2019.

[40] J. Ugander, L. Backstrom, C. Marlow, and J. M. Kleinberg. Structural diversity in social contagion. *Proc. Natl. Acad. Sci.*, 109(16):5962–5966, 2012.

[41] D. Vogiatzis. Influence study on hyper-graphs. In *AAAI*, 2013.

[42] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin. Efficient structural graph clustering: An index-based approach. *PVLDB*, 11(3):243–255, 2017.

[43] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. In *ICDE*, pages 133–144, 2016.

[44] S. Wirth, M. Yanike, L. M. Frank, A. C. Smith, E. N. Brown, and W. A. Suzuki. Single neurons in the monkey

hippocampus and learning of new associations. *Science*, 300(5625):1578–1581, 2003.

[45] S. Wu, A. D. Sarma, A. Fabrikant, S. Lattanzi, and A. Tomkins. Arrival and departure dynamics in social networks. In *WSDM*, pages 233–242, 2013.

[46] S. Wuchty and E. Almaas. Peeling the yeast protein network. *Proteomics*, 5(2):444–449, 2005.

[47] J. Yao, C. Lin, X. Xie, A. J. A. Wang, and C. Hung. Path planning for virtual human motion using improved a star algorithm. In *ITNG*, pages 1154–1158, 2010.

[48] Yelp. Yelp Dataset Challenge: Discover what insights lie hidden in our data. `https://www.yelp.com/dataset/challenge`, 2015.

[49] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang. Index-based densest clique percolation community search in networks. *IEEE Trans. Knowl. Data Eng.*, 30(5):922–935, 2018.

[50] F. Zhang, X. Lin, Y. Zhang, L. Qin, and W. Zhang. Efficient community discovery with user engagement and similarity. *The VLDB Journal*, pages 1–26, 2019.

[51] F. Zhang, L. Yuan, Y. Zhang, L. Qin, X. Lin, and A. Zhou. Discovering strong communities with user engagement and tie strength. In *DASFAA*, pages 425–441, 2018.

[52] F. Zhang, W. Zhang, Y. Zhang, L. Qin, and X. Lin. OLAK: an efficient algorithm to prevent unraveling in social networks. *PVLDB*, 10(6):649–660, 2017.

[53] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. Finding critical users for social network engagement: The collapsed k-core problem. In *AAAI*, pages 245–251, 2017.

[54] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. When engagement meets similarity: Efficient (k, r)-core computation on social networks. *PVLDB*, 10(10):998–1009, 2017.

[55] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *ICDE*, pages 337–348, 2017.

[56] F. Zhao and A. K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.

[57] Z. Zhou, F. Zhang, X. Lin, W. Zhang, and C. Chen. K-core maximization: An edge addition approach. In *IJCAI*, pages 4867–4873, 2019.