Global Reinforcement of Social Networks: The Anchored Coreness Problem

Qingyuan Linghu
Guangzhou University
University of New South Wales
q.linghu@unsw.edu.au

Fan Zhang*
Guangzhou University
fanzhang.cs@gmail.com

Xuemin Lin
University of New South Wales
lxue@cse.unsw.edu.au

Wenjie Zhang
University of New South Wales
zhangw@cse.unsw.edu.au

Ying Zhang
University of Technology Sydney
ying.zhang@uts.edu.au

ABSTRACT

The stability of a social network has been widely studied as an important indicator for both the network holders and the participants. Existing works on reinforcing networks focus on a local view, e.g., the anchored k-core problem aims to enlarge the size of the k-core with a fixed input k. Nevertheless, it is more promising to reinforce a social network in a global manner: considering the engagement of every user (vertex) in the network. Since the coreness of a user has been validated as the "best practice" for capturing user engagement, we propose and study the anchored coreness problem in this paper: anchoring a small number of vertices to maximize the coreness gain (the total increment of coreness) of all the vertices in the network. We prove the problem is NP-hard and show it is more challenging than the existing local-view problems. An efficient heuristic algorithm is proposed with novel techniques on pruning search space and reusing the intermediate results. Extensive experiments on real-life data demonstrate that our model is effective for reinforcing social networks and our algorithm is efficient.

KEYWORDS

Social network, User engagement, Core decomposition

*Qingyuan Linghu and Fan Zhang are the joint first authors. Fan Zhang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389744

ACM Reference Format:
Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2020. Global Reinforcement of Social Networks: The Anchored Coreness Problem. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389744

1 INTRODUCTION

The leave of users in a social network may cause negative influence to the engagement level of their neighbors (e.g., friends) in this network, and thus these neighbors may choose to leave [31]. The continuous departure of users may lead to the leave of users with many neighbors and significantly bring down overall user engagement (stability) of a network. For instance, Friendster was a popular social network which had over 115 million users, while it is suspended due to contagious leave of users [21, 38].

Assume that each vertex v incurs an (integer) cost of $k > 0$ to remain engaged and obtains a benefit of 1 from each neighbor of v who is engaged, the natural equilibrium of this model corresponds to the k-core of the social network [5]. The k-core is defined as the maximal subgraph in which every vertex has at least k neighbors in the subgraph [32, 36]. Given a graph, the k-core can be computed by iteratively removing every vertex with degree less than k. Every vertex in the graph has a unique coreness value, that is, the largest k s.t. the k-core contains the vertex. The model of k-core is often used in the study of network stability (engagement) as it well captures the dynamic of user engagement, e.g., [31, 37, 41].

As the size of k-core is a feasible indicator of network stability, Bhawalkar and Kleinburg et al. proposed the *anchored k-core* (AK) *problem* [5, 6]: given a graph G, an integer k and a budget b, anchoring a set of b vertices in the graph s.t. the number of vertices in the k-core is maximized. The degree (the number of neighbors) of an anchored vertex is considered as positive infinity, namely, an anchored vertex will stay in the k-core regardless of its original degree. It is promising to reinforce a network by giving incentives to some users

(e.g., anchored vertices) such that they will keep engaged in the network and support the engagement of other users [6]. The anchored k-core problem has been further studied on different aspects, e.g., the theoretical side [14, 15], the experimental evaluation [21, 44] and the efficient solutions [40, 45].

Nevertheless, the anchored k-core (AK) problem is essentially to reinforce a network in a "local" manner: it focuses on enlarging the size of the k-core with a particular k value. As proved in [45], given an integer k, the AK problem can only increase the corenesses of a partial set of vertices, e.g., the vertices with coreness $k-1$. Besides, for the AK problem, the valid vertices for anchoring are from a small set of vertices, and the anchoring of other vertices cannot enlarge the size of k-core [45]. Moreover, it is very hard to determine a good input value of k for the AK problem.

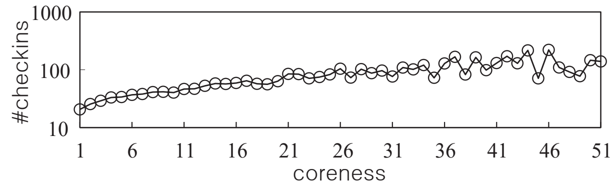


Figure 1: Check-in Number v.s. Coreness Value

As analyzed in the study of Friendster, its collapse may start from the leave of users in either the center cores (k-cores with large k values) [37] or the outside of the center cores [21], i.e, the collapse may happen in a "global" way. As shown in [31], the coreness of a user is the "best practice" for measuring the engagement level of the user in a network. We further examine the matching of coreness and user engagement in real-life social networks. For each integer k, we count the average number of user check-ins (as the ground-truth user engagement) for the users with coreness equals to k. As shown in Figure 1, the coreness value and check-in number in Gowalla [27] are in a positive correlation, except for the disturbance on the center cores due to the small sample. Thus, it is more promising to reinforce a network in a "global" manner: considering the coreness increment of every user. Motivated by the above facts, we propose and study the *anchored coreness* (AC) *problem*: given a graph G and a budget b, anchor a set of b vertices in the graph s.t. the coreness gain (the total increment of coreness) of all the vertices is maximized. The *followers* of an anchor x are the vertices with coreness increased after anchoring x, except x.

Table 1: Anchored k-Core v.s. Anchored Coreness in Fig. 2

Problem	Input	Anchor	Followers	Coreness
AK	$k=3, b=1$	u_1	u_2, u_3, u_4	from 2 to 3
	$k=4, b=1$	u_5	u_6, u_7, u_8	from 3 to 4
AC	$b=1$	u_2	u_3, u_4	from 2 to 3
			u_7, u_8	from 3 to 4

Example 1.1. Figure 2 shows a graph G of 13 vertices and their connections. The coreness of each vertex is marked

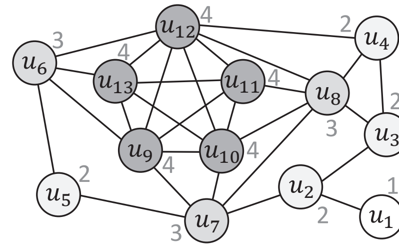


Figure 2: A Toy Example

near the vertex, e.g., the coreness of u_5 is 2. The k-core of G is induced by all the vertices with coreness of at least k, e.g., the 3-core is induced by u_6, u_7, ..., u_{12}, and u_{13}.

Table 1 records the results of anchored k-core (AK) problem and anchored coreness (AC) problem under different inputs. For instance, when $k=3$ and $b=1$, the AK problem anchors u_1 which will increase the coreness of u_2, u_3 and u_4 from 2 to 3. We can find that the anchoring of u_2 according to AC has a larger coreness gain (i.e., 4) compared to that of AK (i.e., 3). Besides, the AC problem improves the vertex coreness from the vertices with different corenesses, while the AK model focuses on a partial set, e.g., the vertices with coreness $k-1$. Thus, AK and AC are inherently different, and the solutions for AK cannot be used to solve the AC problem.

Challenges. To the best of our knowledge, we are the first to study the anchored coreness (AC) problem. We prove the AC problem is NP-hard. Although the coreness gain can be computed in $O(m)$ time by core decomposition [4], a basic exact solution has to exhaustively compute the coreness gain on every possible anchor set with size b, which is cost-prohibitive. We also prove the problem is APX-hard and the coreness gain function is non-submodular. Although it is unpromising to estimate or predict the coreness gain of multiple anchors, we observe that the change of coreness is relatively restricted for one anchored vertex. Thus, we adopt a greedy heuristic to find the best anchor in each iteration, while the candidate anchor set is still very large and a straightforward implementation is still very time consuming.

An efficient algorithm is proposed for the anchored k-core (AK) problem in [45], while the AK model only considers the coreness gain from $k-1$ to k by maximizing the size of k-core with a fixed k. Since the AC problem aims to maximize the coreness gain from all the vertices with different corenesses, the solution in [45] cannot be applied to solve the problem. Besides, the search space of the AC problem is much larger than the AK problem because every vertex in the graph is possible to be a valid anchor to improve the vertex coreness, while only a partial set of vertices related to k-core can be valid anchors to enlarge the size of k-core for AK problem. Therefore, the AC problem is even more challenging than the AK problem. It is critical to design strong strategies to prune unpromising candidate anchors and speed up the computation of coreness gain.

Our Solution. Due to the huge number of candidate anchors, a well-designed reusing strategy is necessary for a greedy heuristic which aims to exhaustively reuse the intermediate results from the executed iterations. To do so, we apply the tree structure $\mathcal{T}$ of core decomposition [4] to divide all the vertices into tree nodes, where each tree node is an atomic unit for deciding whether the computed results associated with the node can be reused. Specifically, with the anchoring of one vertex x, we first prove the coreness of a vertex (except the anchor) can increase by at most 1. Then, the followers of x can be divided into different tree nodes of $\mathcal{T}$. In each iteration, the number of x's followers is the coreness gain of anchoring x. Thus, if x was anchored and the follower set of each vertex was computed (or reused) in the last iteration, for each candidate anchor u in current iteration, we can efficiently decide whether the partial set of u's followers associated with a tree node keeps the same and can be reused.

The proposed computation of coreness gain is adaptive to the reusing mechanism. If a follower unit (in a tree node) cannot be reused, the follower computation is conducted locally, i.e., within the tree node. Besides, we utilize the graph degeneracy ordering (the vertex deletion sequence of core decomposition) to largely speed up the follower computation. We also propose an upper bound of coreness gain to further prune candidate anchors, and well match the technique with the reusing mechanism to improve efficiency. Combining all these techniques, our final GAC algorithm is proposed to efficiently identify the best anchor in each iteration.

Contributions. Our principal contributions are as follows.

- Motivated by many existing studies, we propose and explore the anchored coreness problem to reinforce social networks which considers the engagement of every user. We prove the problem is NP-hard and APX-hard. The problem is shown to be more challenging than the anchored k-core problem which focuses on the engagement of partial users.
- Based on the tree of core decomposition, we introduce a novel mechanism to reuse the intermediate results from the executed iterations. It exhaustively reuses the computed result in each unit represented by a tree node. We also propose the computation of coreness gain which is largely faster than core decomposition. An upper bound of coreness gain is proposed to further prune unpromising candidates. All the techniques are well equipped in the reusing mechanism.
- Comprehensive experiments are conducted on 8 real-life datasets to show that (1) the proposed GAC algorithm is more effective than the other heuristics on improving vertex coreness; (2) the coreness gain from the AC model is much larger than that of the AK model; (3) the coreness values of the anchors and followers are more diverse in the AC model, compared with the AK model; and (4) our proposed techniques largely improve the algorithm efficiency.

Table 2: Summary of Notations

Notation	Definition
G	an unweighted and undirected graph
$V(G)$; $E(G)$	the vertex set of G; the edge set of G
n; m	$\|V(G)\|$; $\|E(G)\|$ (assume $m > n$)
u, v, x	a vertex in G
$E(u)$	the set of edges incident to u
$N(u, G)$	the set of neighbors of u in G
$C_k(G)$	the k-core of G
$c(u, G)$	the coreness of u in G
A	the set of anchor vertices
$deg(u, G)$	$\|N(u, G)\|$ if $u \notin A$, or $+\infty$ if $u \in A$
$c^A(u, G)$	the coreness of u in G with A anchored
b	the budget for the number of anchors
$g(A, G)$	the coreness gain of anchoring A in G
$\mathcal{T}$	the core component tree of G
$\mathcal{F}(x, G)$	the set of followers of x in G
$H_k^i(G)$	i-layer within the k-shell of G
$\mathcal{P}(u)$	the shell-layer pair of a vertex u. If $\mathcal{P}(u) = (k, i)$, u is in the i-th layer of the k-shell, i.e., $u \in H_k^i(G)$.
$x \rightsquigarrow u$	an upstair path from x to u
$CF(x)$	the candidate followers set of x
$d^+(x)$	the degree bound of x
$UB_\sigma(x)$	the upper bound of $\|\mathcal{F}(x)\|$

2 PRELIMINARIES

We consider an unweighted and undirected graph $G = (V, E)$, where $V(G)$ (resp. $E(G)$) represents the set of vertices (resp. edges) in G. $N(u, G)$ is the set of adjacent vertices of u in G, which is also called the neighbor set of u in G. Table 2 summarizes some notations used throughout this paper. Note that we may omit the input graph in the notations when the context is clear, e.g., using $deg(u)$ instead of $deg(u, G)$.

Definition 2.1. **k-core** [32, 36]. Given a graph G, a subgraph S is the k-core of G, denoted by $C_k(G)$, if (i) S satisfies degree constraint, i.e., $deg(u, S) \geq k$ for each $u \in V(S)$; and (ii) S is maximal, i.e., any supergraph $S' \supset S$ is not a k-core.

If $k \geq k'$, the k-core is always a subgraph of k'-core, i.e., $C_k(G) \subseteq C_{k'}(G)$. Each vertex in G has a unique coreness.

Definition 2.2. **coreness**. Given a graph G, the coreness of a vertex $u \in V(G)$, denoted by $c(u, G)$, is the largest k such that $C_k(G)$ contains u, i.e., $c(u, G) = max\{k \mid u \in C_k(G)\}$.

Definition 2.3. **core decomposition**. Given a graph G, core decomposition of G is to compute the coreness of every vertex in $V(G)$.

Algorithm 1: **CoreDecomp**(G, A)

Input : a graph G, an anchor set A
Output : $c^A(u, G)$ for each $u \in V(G)$

```
1 k ← 1;
2 while exist non-anchor vertices in G do
3     while ∃u ∈ V(G) with deg(u) < k do
4         deg(v) ← deg(v) − 1 for each v ∈ N(u, G);
5         remove u and its adjacent edges from G;
6         c^A(u, G) ← k − 1;
7     k ← k + 1;
8 return c^A(u, G) for each u ∈ V(G)
```

In this paper, once a set A of vertices in the graph G is **anchored**, the degrees of the vertices in A are regarded as positive infinity, i.e., for each $x \in A$, $deg(x, G) = +\infty$. Every anchored vertex is called an **anchor** or an **anchor vertex**. The existence of anchor vertices may change the corenesses of other vertices. We use $c^A(u, G)$ (resp. $c^x(u, G)$) to denote the coreness of u in G with the anchor set A (resp. vertex x).

The computation of core decomposition with anchors is the same as that without anchors [4], in which we recursively delete the vertex with the smallest degree in the graph G. The time complexity is still $O(m)$, because the only difference is that we do not delete the anchors in the core decomposition. The pseudo-code is shown in Algorithm 1.

Definition 2.4. **coreness gain**. Given a graph G and an anchor set A, the coreness gain of G regarding A, denoted by $g(A, G)$, is the total increment of coreness for every vertex in $V(G) \setminus A$, i.e., $g(A, G) = \sum_{u \in V(G) \setminus A} (c^A(u) - c(u))$.

Problem Statement. Given a graph G and a budget b, the *anchored coreness problem* aims to find a set A of b vertices in G such that the coreness gain regarding A is maximized, i.e., $g(A, G)$ is maximized.

3 PROBLEM ANALYSIS

THEOREM 3.1. *Given a graph G, the anchored coreness problem is NP-hard.*

PROOF. We reduce the maximum coverage (MC) problem [24], which is NP-hard, to the anchored coreness problem. Given a number b and a collection of sets where each set contains some elements, the MC problem is to find at most b sets to cover the largest number of elements.

Consider an arbitrary instance H of MC with c sets $T_1, .., T_c$ and d elements $\{e_1, .., e_d\} = \cup_{1 \le i \le c} T_i$, we construct a corresponding instance of the anchored coreness problem on a graph G. W.l.o.g., we assume $b < c < d$. Figure 3 shows a construction example from 3 sets and 4 elements.

The graph G contains three parts: M, N, and some cliques. The part M contains c vertices, i.e., $M = \cup_{1 \le i \le c} w_i$ where each w_i corresponds to the set T_i in the MC instance H. The part N contains d vertices, i.e., $N = \cup_{1 \le i \le d} v_i$ where each v_i corresponds to the element e_i in H. For every i and j, if $e_i \in T_j$ in H, we add an edge between v_i and w_j. For each v_i in N, we create d cliques where each clique is a $(d+2)$-clique (a clique of size $d + 2$), and connect v_i to one vertex of each clique. The construction of G is completed.

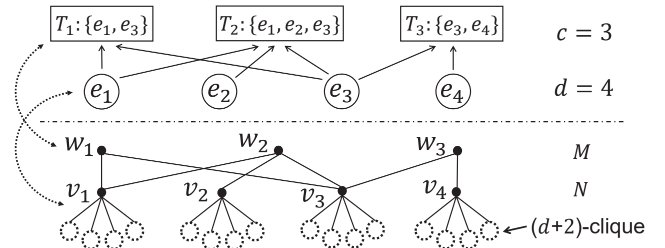


Figure 3: Construction Example for Hardness Proofs

Assume each element in H is contained by at least 1 set, for each $w_i \in M$ and $v_j \in (V(G) \setminus M)$, we have $deg(w_i) \le d < deg(v_j)$. Recall that the core decomposition of G iteratively deletes the vertices with degree less than k and assigns the coreness of $k - 1$ to the deleted vertices in current iteration, from $k = 1, 2, ...$ to $k = k_{max}$. Thus, the coreness of each $w_i \in M$ is $deg(w_i)$, as w_i can only be deleted when $k = deg(w_i) + 1$. The coreness of each $v_j \in N$ is d, as v_j is not deleted when $k = d$ (due to the d cliques), and v_j is deleted when $k = d + 1$ (due to the deletion of every $w_i \in M$). Similarly, the coreness of every vertex in a $(d + 2)$-clique is $d + 1$.

For each $w_i \in M$, even if all the neighbors of w_i are anchored, the coreness of w_i keeps same, as w_i will still be deleted when $k = deg(w_i) + 1$. As we assume $b < c < d$, for the anchoring of any b vertices, each non-anchor vertex u in a $(d + 2)$-clique will still be deleted when $k = d + 2$ (coreness of u keeps same), and thus the anchoring of multiple anchors cannot increase the coreness of any non-anchor $v_i \in N$ to larger than $d + 1$. So, for each non-anchor $v_i \in N$, the coreness of v_i increases by 1 (from d to $d + 1$) iff at least one v_i's neighbor in M is anchored. The optimal anchor set A for anchored coreness problem corresponds to the optimal set collection C for MC problem, where each vertex $w_i \in A$ corresponds to the set $T_i \in C$. If there is a polynomial time solution for the anchored coreness problem, the MC problem will be solved in polynomial time. □

Then, we prove that there is no PTAS for the anchored coreness problem and thus it is APX-hard unless P=NP.

THEOREM 3.2. *For any $\epsilon > 0$, the anchored coreness problem cannot be approximated in polynomial time within a ratio of $(1 - 1/e + \epsilon)$, unless P=NP.*

PROOF. We use the reduction from the MC problem same to the proof of Theorem 3.1. For any $\epsilon > 0$, the MC problem cannot be approximated in polynomial time within a ratio of $(1 - 1/e + \epsilon)$, unless $P = NP$ [20]. We have an anchor set A for anchored coreness problem on G corresponding

to a set collection C for MC problem, where each $w_i \in A$ corresponds to $T_i \in C$. Let $\gamma > 1 - 1/e$, if there is a solution with γ-approximation on the coreness gain for the anchored coreness problem, there will be a γ-approximate solution on optimal element number for the MC problem. □

Besides, the function of coreness gain is not submodular.

Theorem 3.3. *The function $g(\cdot)$ of coreness gain is not submodular.*

Proof. For two arbitrary anchor sets A and B, if $g(\cdot)$ is submodular, it must hold that $g(A)+g(B) \geq g(A \cup B)+g(A \cap B)$. We consider a graph G where the vertex set $V = \cup_{1 \leq i \leq 6} v_i$, the vertices in $\cup_{2 \leq i \leq 5} v_i$ form a 4-clique, v_1 connects to v_2 and v_3, and v_6 connects to v_4 and v_5. If $A = \{v_1\}$ and $B = \{v_6\}$, $g(A) + g(B) = 0 < g(A \cup B) + g(A \cap B) = 4$. □

4 OUR APPROACH

The hardness of the problem motivates us to develop an efficient heuristic algorithm. We adopt a greedy heuristic which iteratively finds one best anchor in each of the b iterations, i.e., the vertex with the largest coreness gain if anchored. To find the best anchor in one iteration, we compute the coreness gain of every candidate anchor. The time complexity of this heuristic is $O(b \cdot n \cdot m)$. However, as our latter theorems indicate, for the anchoring of one vertex, the change of coreness for other vertices is restricted and the computation cost may be largely reduced. Also, Our experiments on real graphs find that the coreness gain from this greedy heuristic is much larger than other heuristics. To improve the efficiency of the greedy algorithm, we aim to significantly reduce (1) the number of candidate anchors and (2) the time cost of computing the coreness gain of one anchor.

We firstly review the tree structure of core decomposition, which can be used to speed up the greedy algorithm (Section 4.1), and the theorems of finding the candidate followers which may increase the coreness due to the anchoring (Section 4.2). Based on the tree and the theorems, we propose a mechanism to reuse the intermediate results across iterations (Section 4.3), and the algorithm to compute the coreness gain of one anchor by partially exploring the tree (Section 4.4). Combining the above with an upper bound technique for candidate anchors pruning, our final GAC algorithm is presented (Section 4.5).

4.1 Core Component Tree

Definition 4.1. k-**core component**. Given a graph G and the k-core $C_k(G)$, a subgraph S is a k-core component if S is a connected component of $C_k(G)$.

According to the definition of k-core, for every integer k, we have *disjointness* property: every k-core component is disjoint from other k-core components in the same k-core; and *containment* property: a k-core component is contained by exactly one (k-1)-core component.

Table 3: Summary of Notations for $\mathcal{T}$

Notation	Definition
$\mathcal{T}[v]$	the tree node which contains the vertex v
TN	a tree node
$TN.K$	a specific coreness k associated with node TN
$TN.V$	the set of vertices in tree node TN
$TN.I$	the smallest vertex id in $TN.V$
$TN.P$	the parent tree node of TN
$TN.C$	the child tree node set of TN
$CC(TN)$	the $(TN.K)$-core component containing $TN.V$
$tca[u][id]$	the set of u's neighbors in $TN.V$ with $TN.I = id$
$sn(u)$	the tree node id set where $id \in sn(u)$ iff $\exists v \in N(u)$ having $c(v) \geq c(u)$ & $\mathcal{T}[v].I = id$
$pn(u)$	the tree node id set where $id \in pn(u)$ iff $\exists v \in N(u)$ having $c(v) < c(u)$ & $\mathcal{T}[v].I = id$
$F[x][id]$	the follower set of x at tree node id, i.e., $v \in F[x][id]$ iff $v \in \mathcal{F}(x)$ & $\mathcal{T}[v].I = id$

Algorithm 2: **BuildCCT**(G, PN)

```
Input  : G : a connected graph, PN : a tree node
Output : T : the core component tree of G
1  k_min ← the smallest coreness from the vertices in V(G);
2  TN ← an empty tree node ;
3  TN.K := k_min; TN.P := PN; PN.C := PN.C ∪ TN;
4  for each unassigned u ∈ V(G) with c(u) = k_min do
5      u is set assigned;
6      TN.V := TN.V ∪ {u};
7      T[u] := TN;
8  TN.I := the smallest vertex id from the vertices in TN.V;
9  for each unassigned u ∈ V(G) in ascending coreness order do
10     G' ← the c(u)-core component containing u;
11     T ← T ∪ BuildCCT(G', TN);
12 return T
```

Tree Structure ($\mathcal{T}$)**.** Given a graph G, the *core component tree* of G, denoted by $\mathcal{T}$, organizes $V(G)$ based on the k-core components with different k. Specifically, $\mathcal{T}$ contains all the vertices in $V(G)$ and each vertex is exclusively contained in one tree node. Given a vertex v, $\mathcal{T}[v]$ is the tree node containing v.

We then clearly introduce the tree structure. Let TN denote a tree node. $TN.K$ is the coreness value associated with TN. The vertices in the subtree rooted at TN induce a subgraph that is a $(TN.K)$-core component, denoted by $CC(TN)$. We use $TN.V$ to denote the set of vertices in the tree node TN and they are the vertices with coreness equal to $TN.K$. We assume each vertex in $V(G)$ has its unique identifier, i.e., id. Let $TN.I$ denote the smallest vertex id from the vertices in $TN.V$. We use $TN.P$ to denote the only parent tree node

of TN, and $TN.C$ to denote the child tree node set of TN. The notations for $\mathcal{T}$ are summarized in Table 3.

Algorithm 2 illustrates the structure of a core component tree. It can be implemented in $O(m)$ time as shown in [32]. If G is not connected, we build a tree for each connected component of G. Given a connected graph G, we execute BuildCCT(G, $\emptyset$) to construct the tree. Initially, every vertex in $V(G)$ is *unassigned*. In each iteration, the algorithm constructs a tree node TN and sets up its domains, e.g., $TN.K$ (Line 2-3). Let k_{min} be the smallest coreness from $V(G)$, every unassigned vertex with coreness k_{min} is pushed into $TN.V$ and set to be *assigned* (Line 4-7). Note that the assigned or unassigned status of a vertex is global. The construction follows a recursive DFS resulting in the expected parent-child relation between two nodes (PN and TN) based on the containment relation of k-core components (Line 9-11).

Some notations for the tree are defined as follows.

Definition 4.2. **tree node classified adjacency** (tca)**.** For a given graph G, we scan the adjacent neighbors of each vertex and use the structure tca to organize them. We partition the neighbors of a vertex according to the tree nodes they belong to, i.e., for a vertex u, $tca[u][id]$ is the set of u's neighbors in the tree node TN with $TN.I = id$.

Definition 4.3. **subtree adjacent nodes set** (sn) Given a vertex u in a graph G, the *subtree adjacent nodes set* of u, denoted by $sn(u)$ is the id set of adjacent tree nodes with the associated coreness not less than $c(u)$, i.e., $id \in sn(u)$ iff $\exists v \in N(u, G)$ having $c(v) \geq c(u)$ & $\mathcal{T}[v].I = id$.

Definition 4.4. **parent adjacent nodes set** (pn) Given a vertex u in a graph G, the *parent adjacent nodes set* of u, denoted by $pn(u)$ is the id set of adjacent tree nodes with the associated coreness less than $c(u)$, i.e., $id \in pn(u)$ iff $\exists v \in N(u, G)$ having $c(v) < c(u)$ & $\mathcal{T}[v].I = id$.

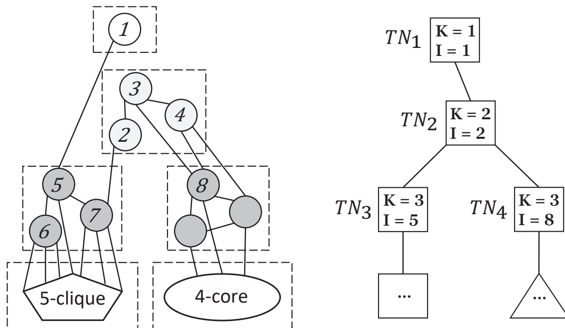


Figure 4: Core Component Tree

Example 4.5. In Figure 4, we have a graph G at left and its corresponding $\mathcal{T}$ at right. Each solid-line box of the right is a tree node which corresponds to a dotted box of the left. We have $\mathcal{T}[\mathbf{2}] = TN_2$, $TN_2.K = 2$ and $TN_2.I = 2$, $\mathcal{T}[\mathbf{7}] = TN_3$, $TN_3.K = 3$ and $TN_3.I = 5$. For tca, sn and pn, for some instances, $tca[\mathbf{2}][5] = \{\mathbf{7}\}$, $tca[\mathbf{2}][2] = \{\mathbf{3}\}$, $tca[\mathbf{7}][2] = \{\mathbf{2}\}$ and $tca[\mathbf{7}][5] = \{\mathbf{5}\}$; $sn(\mathbf{2}) = \{2, 5\}$ and $pn(\mathbf{7}) = \{2\}$.

Note that, tca, sn and pn are the structures associated with $\mathcal{T}$ and can be retrieved along with the building of $\mathcal{T}$.

4.2 Restriction of Candidate Followers

If a vertex x is anchored, the set of candidate vertices which may increase their corenesses is restricted.

Theorem 4.6. *If a vertex x is anchored in G, any non-anchor vertex $u \in V(G)$ can increase its coreness by at most 1.*

Proof. In the rest of the paper, please find the proofs in Section 8 for all the theorems. □

Tree Node Classified Follower Set (F)**.** Every non-anchor vertex with coreness increased by anchoring x is named as a **follower** of x. The follower set of x in G is denoted by $\mathcal{F}(x, G)$ that contains all its followers. According to Theorem 4.6, $g(\{x\}) = |\mathcal{F}(x)|$. We define F to divide the followers of an anchor based on *tree node classified adjacency*. Specifically, for $x \in V(G)$, $v \in F[x][id]$ iff $v \in \mathcal{F}(x)$ & $\mathcal{T}[v].I = id$.

A fast method to compute the followers will be introduced in Section 4.4. Note that, when we record the follower sets, we do not store the specific followers of a vertex x but only store the number of followers of x regarding each adjacent tree node, so the space cost of F is $O(m)$. The candidate followers of a vertex x can be extracted as follows.

Theorem 4.7. *If a vertex x is anchored in the graph G, we have $\mathcal{F}(x) \subset \bigcup_{id \in sn(x)} \mathcal{T}[id].V$.*

4.3 Reuse of Intermediate Results

After one iteration of our greedy heuristic where we choose to anchor x. For each vertex $u \neq x$, suppose we have had the follower set $F[u][id]$ for each tree node $id \in sn(u)$ before anchoring x. To reuse the follower results after anchoring x, we apply Algorithm 3 to decide, for every vertex u, whether the follower set of u on each tree node keeps the same in the next iteration.

According to Theorem 4.7, we get the affected vertex set $V_x := \bigcup_{id \in sn(x)} \mathcal{T}[id].V$ (Line 1), and initialize the reusable node set $rn(\cdot)$ for each vertex (Line 2). We remove the tree node ids from $rn(\cdot)$ where the followers cannot be reused in the next iteration (Line 3-6). Then we run core decomposition on the subgraph $CC(\mathcal{T}[x])$ with x anchored (Line 7-8) and update the subtree rooted at x (Line 9-11). The update of tree structure finds some other vertices which may be affected w.r.t x (Line 12-13). Similar to Line 3-6, we remove the tree node ids from $rn(\cdot)$ where the followers cannot be reused by above affected vertices (Line 13-16). In the implementation, for a vertex u, we can easily avoid duplicate removals in $rn(u)$ triggered by u's neighbors using tree node tags.

Algorithm 1 (Line 8) and Algorithm 2 (Line 9) both have $O(m)$ time complexity. In Line 3-6 and Line 13-16, each edge

Algorithm 3: **ResultReuse**(x, G, $\mathcal{T}$)

Input : x: the anchor vertex, G : a social network, $\mathcal{T}$: the core component tree of G,
Output : the tree node set $rn(u)$ for each vertex $u \in V(G)$, where $F[u][id]$ can be reused for each $id \in rn(u)$

```
1  V_x := ⋃_{id∈sn(x)} T[id].V;
2  rn(u) := sn(u) for each u ∈ V(G);
3  for each v ∈ V_x do
4      id := T[v].I; rn(v) := rn(v) \ {id};
5      for each id' ∈ pn(v) and each u ∈ tca[v][id'] do
6          rn(u) := rn(u) \ {id};
7  G' ← CC(T[x]); P' ← T[x].P;
8  CoreDecomp(G', {x});
9  T* ← BuildCCT(G', P');
10 T' ← T with the subtree rooted at P' replaced by T*;
11 Get tca', sn' and pn' from T';
12 V'_x := ⋃_{v∈V_x} T'[v].V;
13 for each v ∈ V'_x \ V_x do
14     id := T[v].I; rn(v) := rn(v) \ {id};
15     for each id' ∈ pn'(v) and each u ∈ tca'[v][id'] do
16         rn(u) := rn(u) \ {id};
17 return rn(u) for every vertex u ∈ V(G)
```

is accessed at most one time, respectively. So, the time complexity of Algorithm 3 is $\mathcal{O}(m)$.

Lemma 4.8. *After the anchoring of vertex x and the execution of Algorithm 3, for every non-anchor vertex $u \in V(G)$ and each $id \in rn(u)$, we have 1) $id \in sn'(u)$, 2) $\mathcal{T}'[id].K = \mathcal{T}[id].K$ and 3) $\mathcal{T}'[id].V = \mathcal{T}[id].V$.*

Theorem 4.9. *After the anchoring of vertex x and the execution of Algorithm 3, let G_x denote the graph with x anchored, considering a non-anchor vertex $u \in V(G_x)$, for each $id \in rn(u)$ and each $v \in \mathcal{T}'[id].V$, we have $v \in \mathcal{F}(u, G_x)$ iff $v \in F[u][id]$.*

After anchoring x, the search space of followers for a non-anchor vertex u is within $\bigcup_{id \in sn'(u)} \mathcal{T}'[id].V$ according to Theorem 4.7. By executing Algorithm 3, we get the $rn(u)$ so that a subset of search space $\bigcup_{id \in rn(u)} \mathcal{T}'[id].V$ does not need to be recomputed, as proven by Theorem 4.9. Essentially, we reduce the search space of follower computation from $\bigcup_{id \in sn'(u)} \mathcal{T}'[id].V$ to $\bigcup_{id \in sn'(u) \setminus rn(u)} \mathcal{T}'[id].V$.

Example 4.10. In Figure 4, we can know that, anchoring vertex **1** can make **5**, **6** and **7** the followers, which means $F[\mathbf{1}][5] = \{5, 6, 7\}$. And anchoring vertex **2** can make **3**, **4** and **7** the followers, which means $F[\mathbf{2}][2] = \{3, 4\}$ and $F[\mathbf{2}][5] = \{7\}$. Now we have $sn(\mathbf{1}) = \{5\}$ and $sn(\mathbf{2}) = \{2, 5\}$. If we choose to anchor **1**, then $V_1 := \{\mathbf{5},\mathbf{6},\mathbf{7}\}$, **5**, **6** and **7** become the followers and join the child node of their current tree node. For vertex **2**, initially we have $rn(\mathbf{2}) = sn(\mathbf{2}) = \{2, 5\}$. But V_1 makes $rn(\mathbf{2}) := rn(\mathbf{2}) \setminus \{5\}$. Obviously, $\mathcal{T}[\mathbf{7}].I = 5$ and **7** is indeed not the follower of **2** any more. And we can see **3** and **4** are still the followers of **2**, which confirms $F[\mathbf{2}][2]$ can be reused since $2 \in rn(\mathbf{2})$.

4.4 Coreness Gain Computation

In this section, we utilize the vertex deletion order in core decomposition to speed up the follower computation. Recall that we have $g(\{x\}, G) = |\mathcal{F}(x)|$ for an anchored vertex x.

Given a graph G, the **k-shell**, denoted by $H_k(G)$, is the set of vertices in G with coreness equal to k, i.e., $H_k(G) = V(C_k(G)) \setminus V(C_{k+1}(G))$. The vertices in the k-shell can be further divided to different vertex sets, named layers, according to their deletion sequence in the core decomposition (Algorithm 1). We use H_k^i to denote the i-layer of the k-shell, which is the set of vertices that are deleted in the i-th batch. Specifically, when $i = 1$, H_k^i is defined as $\{u \mid deg(u, C_k(G)) < k+1 \ \& \ u \in C_k(G)\}$. The deletion of the 1st-layer will produce the 2nd-layer. Recursively, when $i > 1$, $H_k^i = \{u \mid deg(u, G_i) < k + 1 \ \& \ u \in G_i\}$ where $G_1 = C_k(G)$ and G_i is the subgraph induced by $V(G_{i-1}) \setminus H_k^{i-1}$ on $C_k(G)$.

Shell-layer Pair. Based on the above definition, each vertex u in the graph G has a *shell-layer pair* (k, i), which means u in the i-th layer of the k-shell, i.e., $u \in H_k^i$. We record the shell-layer pair of every vertex u in $\mathcal{P}$. Specifically, for every vertex v, it is contained in the $(\mathcal{P}[v].i)$-th layer of the $(\mathcal{P}[v].k)$-shell in G. We define $\mathcal{P}[v_i] \prec \mathcal{P}[v_j]$ iff $\mathcal{P}[v_i].k < \mathcal{P}[v_j].k$ or $\mathcal{P}[v_i].k = \mathcal{P}[v_j].k \ \& \ \mathcal{P}[v_i].i < \mathcal{P}[v_j].i$.

Example 4.11. In Figure 5 (a), we can see the 2-shell contains u_1, u_2 and u_3, and the 3-shell contains u_4 and u_5. However, u_1 is the first to be deleted in core decomposition, because u_1 is the only one whose degree is less than 3 currently. After u_1 being deleted with $\mathcal{P}[u_1]$ = (2, 1), edges (u_1, u_2) and (u_1, u_4) are deleted. Then, u_2 becomes the only one with degree less than 3, so u_2 is deleted with $\mathcal{P}[u_2]$ = (2, 2). Similarly, $\mathcal{P}[u_3]$ = (2, 3). Both $\mathcal{P}[u_4]$ and $\mathcal{P}[u_5]$ are equal to (3, 1) since they contradict the degree constraint at the same time.

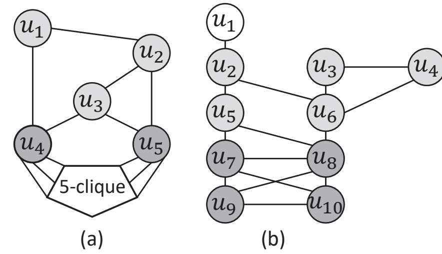


Figure 5: Example Figures

Definition 4.12. **Upstair Path**. We say there is an upstair path in G for $u \in V(G)$ w.r.t a given anchor vertex x if there is a path $x \rightsquigarrow u$ where (i) for every vertex y in the path except x, $\mathcal{P}[y].k = \mathcal{P}[u].k$; and (ii) for every two consecutive vertices v' and v'' from x to u, $(v', v'') \in E(G)$ and $\mathcal{P}[v'] \prec \mathcal{P}[v'']$.

Example 4.13. In Figure 5 (b), we can compute the shell-layer pairs of the vertices and get $\mathcal{P}[u_1]$ = (1, 1), $\mathcal{P}[u_2]$ =

Algorithm 4: **FindFollowers**(x, G, $\mathcal{T}$)

```
Input  : x : the anchor, G : a social network, T : the core
         component tree of G
Output : F[x][·] : tree node classified follower sets of x
1  x is set survived;
2  for each non-reusable tree node id ∈ sn(x) \ rn(x) do
3      H := ∅;
4      if id = i_x then
5          H.push(u) for each u ∈ tca^≥_=(x);
6      else
7          H.push(u) for each u ∈ tca[x][id];
8      while H ≠ ∅ do
9          u ← H.pop();
10         Compute d⁺(u);
11         if d⁺(u) ≥ c(u, G) + 1 then
12             u is set survived;
13             for each v ∈ tca^≥_=(u) and v ∉ H do
14                 H.push(v);
15         else
16             u is set discarded ;
17             Shrink(u);
18     F[x][id] ← survived vertices\{x} ;
19 return F[x]
```

Algorithm 5: **Shrink**(u)

```
Input  : u : the vertex for degree check
1  for each survived neighbor v with v ≠ x do
2      d⁺(v) := d⁺(v) − 1;
3      T ← v If d⁺(v) < c(v, G) + 1;
4  for each v ∈ T do
5      v is set discarded;
6      Shrink(v);
```

$\mathcal{P}[u_3]$ = $\mathcal{P}[u_4]$ = (2, 1), $\mathcal{P}[u_5]$ = $\mathcal{P}[u_6]$ = (2, 2) and $\mathcal{P}[u_7]$ = $\mathcal{P}[u_8]$ = $\mathcal{P}[u_9]$ = $\mathcal{P}[u_{10}]$ = (3, 1). The path (u_1, u_2, u_5) is an upstair path for u_5 w.r.t u_1, because $\mathcal{P}[u_1] \prec \mathcal{P}[u_2]$, $\mathcal{P}[u_2] \prec \mathcal{P}[u_5]$, and $\mathcal{P}[u_2].k = \mathcal{P}[u_5].k$. (u_2, u_5) itself can also be an upstair path for u_5 w.r.t u_2, because it does not contradict any constraint in Definition 4.12. On the contrary, (u_3, u_4, u_6) cannot be an upstair path for u_6 w.r.t u_3 because $\mathcal{P}[u_3] = \mathcal{P}[u_4]$ (contradicts (ii) of Definition 4.12), neither nor (u_3, u_6, u_8) for u_8 w.r.t. u_3 because $\mathcal{P}[u_6].k \neq \mathcal{P}[u_8].k$ which contradicts the (i) of Definition 4.12.

Theorem 4.14. *A vertex $u \in V(G)$ is a follower of the anchor x implies that there is an upstair path $x \rightsquigarrow u$ in G.*

Computing Followers. According to Theorem 4.14, the vertices without any upstair path from the anchor vertex x cannot be a follower of x. We use $CF(x)$ to denote all the candidate followers of an anchor x, i.e., the vertices that can be reached by x via upstair paths. Instead of doing core decomposition of the whole graph, we only need to explore the candidate followers $CF(x)$ to compute the follower set of x. We use $tca^{\leq}_{=}(u)$ to denote the set of u's neighbours where each neighbor v has $\mathcal{P}[v].k = \mathcal{P}[u].k$ & $\mathcal{P}[v].i \leq \mathcal{P}[u].i$. Similarly, $tca^{\geq}_{=}(u)$ contains every u's neighbour v with $\mathcal{P}[v].k = \mathcal{P}[u].k$ & $\mathcal{P}[v].i > \mathcal{P}[u].i$. For simplicity, we use i_u to denote the id of the tree node which contains the vertex u, i.e., $i_u = \mathcal{T}[u].I$. Note that, $tca^{\leq}_{=}(u)$ and $tca^{\geq}_{=}(u)$ are easily retrieved along with core decomposition.

Algorithm 4 shows the pseudo-code for computing the followers. In each iteration, we search the non-reusable tree nodes (section 4.3) in $\mathcal{T}$ to compute the followers of x in the nodes (Line 2, Algorithm 4). We maintain a min heap H to store the candidate followers $CF(x)$ which will be explored (Line 3-7 and 13-14). The key of a vertex in H is its shell-layer pair with ties broken by the vertex id. In each tree node $id \in sn(x) \setminus rn(x)$, we explore $CF(x)$ in a layer-by-layer manner: from j-th layer to $(j+1)$-th layer starting from x.

In the layer-by-layer search, a vertex is set as **unexplored** if it has never been checked with the degree constraint (Line 11). A vertex is set as **survived** if it survived the degree check (Line 12), otherwise it is set as **discarded** (Line 16). The *discarded* vertices will never be visited again, and a *survived* vertex may become *discarded* later due to the deletion cascade. The vertices that will not be visited in the search, e.g., not in any upstair path, are regarded as *discarded.*

Once a candidate follower u is discarded (Line 16), Algorithm 5 will be called to recursively delete other vertices without sufficient degree bound due to the deletion of u. After traversing all the candidate followers and deleting the candidates that cannot survive the degree check, the remaining vertices in $CF(x)$ are the true followers of x. Note that the followers are separately computed and returned for each tree node (Line 2 and Line 18 of Algorithm 4).

The time complexity of Algorithm 4 is $O(m)$, because each edge is accessesed at most three times: push neighbors into H, degree check, and compute the cascade of shrink.

Degree Check. The degree bound of a vertex $u \in CF(x)$ is denoted by $d^+(u)$. Specifically, $d^+(u) = d^+_s(u) + d^+_u(u) + d_>(u)$, in which $d^+_s(u)$ (resp. $d^+_u(u)$) is the number of *survived* (resp. *unexplored*) neighbors in $\{x\} \cup (tca^{\leq}_{=}(u) \cap H) \cup tca^{\geq}_{=}(u)$, and $d_>(u)$ is the number of neighbors in $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$. The following theorem indicates that we can exclude a candidate follower u if $d^+(u) < c(u, G) + 1$. The discard of a vertex may invoke the discard of other vertices, as shown in Algorithm 5. When the deletion cascade terminates, the tags of all the vertices affected by the discard of u will be correctly updated.

Theorem 4.15. *A vertex $u \in CF(x)$ cannot be a follower of x if $d^+(u) < c(u, G) + 1$.*

For simplicity, in the following examples, the *id* of a vertex u_i is u_i itself where $i \in [1, V(G)]$ & $i \in \mathbb{N}$. For two vertices u_i and u_j, we set $u_i < u_j$ iff $i < j$.

Example 4.16. In Figure 5 (b), we explain an example of using Algorithm 4 to compute the followers of u_1 from a single tree node. For the core component tree $\mathcal{T}$, we can see there are three tree nodes TN_1, TN_2 and TN_3, where $TN_1.V = \{u_1\}$, $TN_1.K = 1$ and $TN_1.I = u_1$; $TN_2.V = \{u_2, u_3, u_4, u_5, u_6\}$, $TN_2.K = 2$ and $TN_2.I = u_2$; $TN_3.V = \{u_7, u_8, u_9, u_{10}\}$, $TN_3.K = 3$ and $TN_3.I = u_7$. Initially, u_1 itself is set survived and we push the only adjacent vertex u_2 which is in $tca[u_1][u_2]$ into the min Heap H. Then we pop u_2 and have $d_s^+(u_2) = 1$, $d_u^+(u_2) = 2$ and $d_>(u_2) = 0$, so u_2 survives the degree check since $d^+(u_2) = c(u_2) + 1$ and we set u_2 survived. We put the vertices of $tca^{\geq}_{=}(u_2)$ into the heap so u_5 and u_6 are now in H. We first explore u_5 and have $d_s^+(u_5) = 1$, $d_u^+(u_5) = 0$, $d_>(u_5) = 2$ and $d^+(u_5) = c(u_5) + 1$, so we set u_5 survived. As $tca^{\geq}_{=}(u_5) = \emptyset$, we do not put any more vertices into H for now. Then we explore u_6 and have $d_s^+(u_6) = 1$, $d_u^+(u_6) = 0$ and $d_>(u_6) = 1$. Note that, u_3 and u_4 are unexplored neighbors of u_6 in $tca^{\leq}_{=}(u_6)$, but they will not be added into H so cannot be counted in $d_u^+(u_6)$. $d^+(u_6) < c(u_6) + 1$ so we will discard it. As illustrated in Algorithm 5, for each survived neighbor of u_6 which is u_2, we make $d^+(u_2) = d^+(u_2) - 1 = 2$ so that $d^+(u_2) < c(u_2) + 1$. So we discard u_2 and then make $d^+(u_5) = d^+(u_5) - 1 = 2$. Obviously $d^+(u_5) < c(u_5) + 1$ and gets discarded. Finally, the heap H becomes empty and anchoring u_1 has no follower.

Reusing Followers. Since we compute the followers of x regarding each tree node $id \in sn(x)$ separately, it is very simple to reuse the followers computed from the last iteration. Specifically, after anchoring each vertex x, we erase some follower results by Algorithm 3. Once a tree node *id* is visited (Line 2, Algorithm 4), we first check whether $id \in rn(x)$ or not. If $id \in rn(x)$, the follower set of x in this tree node is not erased by Algorithm 3. Thus we do not need to compute these followers (Line 3-17, Algorithm 4) again, and use the existing $F[x][id]$ instead. If $id \notin rn(x)$, we execute the Line 3-17 of Algorithm 4 to find the correct followers.

4.5 The GAC Algorithm

We first introduce an upper bound of follower number.

Upper Bound Based Pruning. We introduce an easy-to-compute upper bound to further prune unpromising candidates before the computation of followers. For a vertex x, by Equation 1, we firstly get the upper bound of followers from its own tree node $\mathcal{T}[x]$. Then for each $id \in sn(x) \setminus \{i_x\}$, we get an upper bound $UB^{>}_{id}(x)$ by Equation 2. At last we can compute the total upper bound $UB_\sigma(x)$ by Equation 3. Note that when $tca^{\geq}_{=}(u) = \emptyset$ for a vertex u, we set $UB_{i_u}(u)$ to 0.

$$UB_{i_x}(x) = \textstyle\sum_{u \in tca^{\geq}_{=}(x)} (UB_{i_u}(u) + 1) \tag{1}$$

$$UB^{>}_{id}(x) = \textstyle\sum_{u \in tca[x][id]} (UB_{i_u}(u) + 1) \tag{2}$$

$$UB_\sigma(x) = UB_{i_x}(x) + \textstyle\sum_{id \in sn(x) \setminus \{i_x\}} UB^{>}_{id}(x) \tag{3}$$

Theorem 4.17. *Given a graph G and an anchor vertex x, $|F[x][i_x]| \leq UB_{i_x}(x)$, and for each $id \in sn(x) \setminus \{i_x\}$, $|F[x][id]| \leq UB^{>}_{id}(x)$. So, $g(\{x\}, G) \leq UB_\sigma(x)$.*

About the computation of the upper bound, after getting the partial ordering (i.e., shell-layer pairs) of $V(G)$, we use *topological sorting* to construct a compatible *total ordering* of $V(G)$. Then we can accumulatively compute the upper bound of each vertex with the reverse sequence of the total ordering with a time complexity of $O(m)$.

Example 4.18. In Figure 5(a), after getting the shell-layer pair of each vertex, $\mathcal{P}[u_1] = (2, 1)$, $\mathcal{P}[u_2] = (2, 2)$, $\mathcal{P}[u_3] = (2, 3)$, and $\mathcal{P}[u_4] = \mathcal{P}[u_5] = (3, 1)$. Now in $\mathcal{T}$, we have TN_1 where $TN_1.V = \{u_1, u_2, u_3\}$, $TN_1.K = 2$ and $TN_1.I = u_1$. $TN_2.V = \{u_4\}$ where $TN_2.K = 3$ and $TN_2.I = u_4$. $TN_3.V = \{u_5\}$ where $TN_3.K = 3$ and $TN_3.I = u_5$. Then we get a total ordering of them: $u_1 \prec u_2 \prec u_3 \prec u_4 \prec u_5$. We compute their upper bounds following this order. For u_4 and u_5, $UB_{u_4}(u_4) = UB_{u_5}(u_5) = 0$ since $tca^{\geq}_{=}(u_4) = tca^{\geq}_{=}(u_5) = \emptyset$. For u_3, $tca^{\geq}_{=}(u_3) = \emptyset$ so $UB_{u_1}(u_3) = 0$. $tca[u_3][u_4] = \{u_4\}$ and $tca[u_3][u_5] = \{u_5\}$, so that $UB^{>}_{u_4}(u_3) = (UB_{u_4}(u_4) + 1) = 1$ and $UB^{>}_{u_5}(u_3) = (UB_{u_5}(u_5) + 1) = 1$. Therefore, $UB_\sigma(u_3) = UB_{u_1}(u_3) + UB^{>}_{u_4}(u_3) + UB^{>}_{u_5}(u_3) = 2$. For u_2, $tca^{\geq}_{=}(u_2) = \{u_3\}$ so $UB_{u_1}(u_2) = (UB_{u_1}(u_3) + 1) = 1$, and $tca[u_2][u_5] = \{u_5\}$ so that $UB^{>}_{u_5}(u_2) = (UB_{u_5}(u_5) + 1) = 1$. Then we have $UB_\sigma(u_2) = UB_{u_1}(u_2) + UB^{>}_{u_5}(u_2) = 2$. At last, we get $tca^{\geq}_{=}(u_1) = \{u_2\}$ and $tca[u_1][u_4] = \{u_4\}$, so we can get $UB_{u_1}(u_1) = (UB_{u_1}(u_2) + 1) = 2$, $UB^{>}_{u_4}(u_1) = (UB_{u_4}(u_4) + 1) = 1$, $UB_\sigma(u_1) = UB_{u_1}(u_1) + UB^{>}_{u_4}(u_1) = 3$.

Upper Bound Refining. After anchoring a vertex in each iteration, we can retain and update some computed upper bounds based on our tree node classified adjacency. Firstly, for each $id \in rn(u)$ of a non-anchor vertex u, $UB_{i_x}(x)$ or $UB^{>}_{id}(u)$ stays the same, so does not need to be recomputed. Secondly, if $F[u][id]$ has been computed and is not erased in Algorithm 3, it can replace $UB_{i_x}(x)$ or $UB^{>}_{id}(u)$ so that a more accurate bound is found.

Combining the Techniques. Algorithm 6 shows the detail of our final greedy algorithm which combines all the proposed techniques. We firstly apply Algorithm 1 (Line 1) to get the initial coreness of each vertex of the given graph G. Then, we apply Algorithm 2 (Line 2) to build the core component tree for the first time, followed by the computing of our upper bound of follower numbers (Line 3), which will be updated after anchoring every vertex (Line 12-13). Then, the greedy heuristic starts (Line 4). In each iteration, we use a to record the best anchor vertex found so far, and use λ to record the number of followers of the best anchor (Line

Algorithm 6: **GAC**(G, b)

```
   Input  : G : a social network, b : number of anchors
   Output : A : the set of anchor vertices
1  CoreDecomp(G, ∅);
2  𝒯 ← BuildCCT(G, root);
3  Compute upper bounds of follower numbers;
4  for i from 1 to b do
5      λ := −1; a := null;
6      for each u ∈ V(G) with decreasing order UB_σ(u) do
7          if u ∉ A and UB_σ(u) > λ then
8              F[u] := FindFollowers(u, G, 𝒯);
9              if |ℱ[u]| > λ then
10                 a := u; λ := |ℱ[u]|;
11     A := A ∪ {a}; deg(a, G) := +∞;
12     ResultReuse(a, G, 𝒯);
13     Refine upper bounds;
14 return A
```

5). We sequentially compute the followers for the vertices in decreasing order of their upper bounds (Line 6). Only if the upper bound of a vertex u is larger than λ and u is not an existing anchor (Line 7), we will continue the follower computation for u (Line 8-10). Note that we will not compute the follower number for u in the tree nodes where the numbers of followers do not change from last iteration and can be reused. After the follower computation of current iteration, the best anchor a is added to the set A, and the degree of a is set to be positive infinity. After b iterations, Algorithm 6 returns the set A of b anchor vertices (Line 14).

5 EXPERIMENTAL EVALUATION

Datasets. We use eight real-life datasets in our experiments. `Brightkite`, `Gowalla`, `Youtube` and `Livejournal` are from http://snap.stanford.edu/. The other datasets are from http://konect.uni-koblenz.de/. Table 4 shows the statistics of the datasets, listed in increasing order of edge numbers.

Algorithms. Towards effectiveness, we mainly compare 6 algorithms with our GAC algorithm, including 4 heuristics, the exact solution, and the algorithm for anchored k-core problem. Towards the efficiency, we incrementally equip the baseline with our proposed techniques to evaluate the performance. Table 5 lists all the evaluated algorithms.

Parameters. We conducted experiments by varying the budget b from 1 to 100 where the default value is 100. All the programs are implemented in C++ and compiled with G++ on Linux. The experiments are conducted on a machine with 3.4GHz Intel Xeon CPU and Redhat system.

5.1 Effectiveness

Comparison with Other Heuristics. In Figure 6, we compare the coreness gain from GAC with other heuristics (Rand, Deg, Deg-C, and SD). For SD, the **successive degree** of a vertex u is defined as $deg_{>}(u) = |\{v \mid v \in N(u,G) \;\&\; \mathcal{P}(v) > \mathcal{P}(u)\}|$ where $\mathcal{P}(u) = (k, i)$ means u is in the i-th layer of the k-shell. The details of these heuristics are in Table 5.

Table 4: Statistics of Datasets

Dataset	Nodes	Edges	d_{avg}	d_{max}	k_{max}
Brightkite	58,228	194,090	6.7	1098	52
Arxiv	34,546	421,578	24.4	846	30
Gowalla	196,591	456,830	9.2	10721	51
NotreDame	325,729	1,497,134	6.5	3812	155
Stanford	281,903	2,312,497	16.4	38626	71
YouTube	1,134,890	2,987,624	5.3	28754	51
DBLP	1,566,919	6,461,300	8.3	2023	118
LiveJournal	3,997,962	34,681,189	17.4	14815	360

Table 5: Summary of Algorithms

Algorithm	Description
Exact	identifies the optimal solution by searching all possible combinations of b anchors
Rand	randomly chooses the b anchors from $V(G)$
Deg	chooses the b anchors from $V(G)$ with the highest degree
Deg-C	chooses the b anchors with the highest value of $deg(u,G) - c(u)$ for each $u \in V(G)$
SD	chooses the b anchors with the highest successive degree $deg_{>}(\cdot)$ for every $u \in V(G)$, where $deg_{>}(u) = \|\{v \mid v \in N(u,G) \;\&\; \mathcal{P}(v) > \mathcal{P}(u)\}\|$
OLAK	the state-of-the-art algorithm for anchored k-core problem [45]
GAC	Algorithm 6
GAC-U	GAC without upper bound pruning (Section 4.5)
GAC-U-R	GAC-U without result reusing (Algorithm 3)
Baseline	GAC-U-R using core decomposition (Algorithm 1) to compute coreness gain, without Algorithm 4

As shown in Figure 6 (a), the performance of Rand is the worst as it chooses random vertices to anchor. The performance of Deg and Deg-C are better than Rand as they choose vertices with large degrees to anchor. SD has more coreness gain because the vertices with higher successive degree have more candidate followers (Theorem 4.14). Compared with the above heuristics, GAC achieves the much larger coreness gains on all the datasets. The effect of varying b is shown in Figures 6 (b) and (c). The coreness gain of GAC increases with larger b values and is better than all other four heuristics under all the settings.

Comparison with Exact Solution. We also compare the result of GAC with the Exact algorithm, which identifies the optimal b anchors by enumerating all possible combinations of b vertices. Due to the huge time cost of Exact, we extract small datasets by iteratively extracting a vertex and all its neighbours, until the number of extracted vertices reaches 100. For both Brighkite and Arxiv, we extracted 10 such subgraphs and report the average coreness gain of them. The

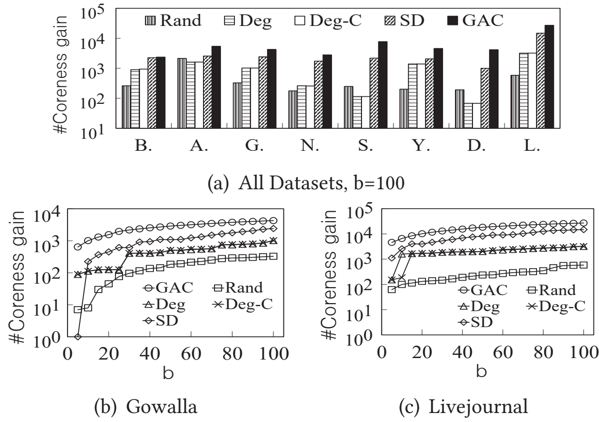


Figure 6: Coreness Gain from Different Heuristics

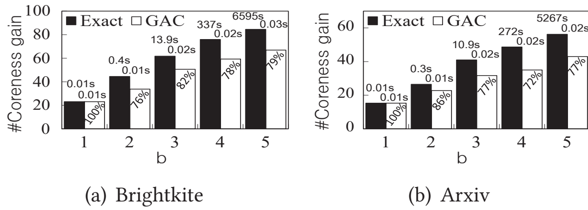


Figure 7: GAC v.s. Exact

Dataset	Deg_{avg}	Deg_{anc}	p_{Deg}	p_{CN}	p_{SD}
Brightkite	7.35	37.76	0.884	0.891	**0.893**
Arxiv	24.37	29.71	0.670	0.663	**0.678**
Gowalla	9.67	43.86	0.904	**0.919**	**0.919**
NotreDame	6.69	11.28	0.808	0.828	**0.846**
Stanford	14.14	56.09	0.745	0.763	**0.788**
YouTube	5.27	81.85	**0.985**	0.982	0.982
DBLP	8.08	27.85	0.905	0.896	**0.911**
LiveJournal	17.35	145.74	0.935	0.940	**0.943**

Table 6: Characteristics of Anchor Set

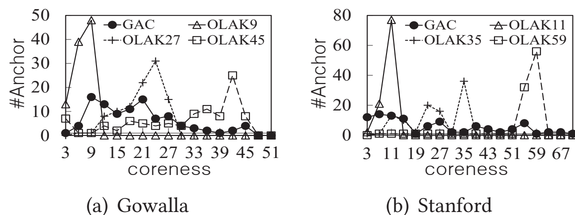


Figure 8: Distribution of Anchors on Coreness

runtimes are also reported. Figure 7 shows that the coreness gain of GAC is always at least 70% of Exact, and GAC is faster than Exact by up to 5 orders of magnitude. Note that the coreness gain percentage of GAC over Exact may increase with larger b values, e.g., from $b = 4$ to $b = 5$.

Characteristics of Anchor Set. Table 6 shows the average degree of anchors (Deg_{anc}) from GAC is much larger than the average degree of all the vertices in the graph (Deg_{avg}). Then, we investigate the average ranking of an anchor in all the vertices regarding degree, coreness, and successive degree, denoted by p_{Deg}, p_{CN}, and p_{SD}, respectively. According to Theorem 4.14, a vertex with larger successive degree has

Dataset	$Gain_{UB}$	$Gain_{DG}$	$Gain_{RD}$	J^{UB}_{DG}	J^{UB}_{RD}
Brightkite	2357	**2598**	2488	0.538	0.538
Arxiv	5426	5391	**5503**	0.739	0.681
Gowalla	**4260**	4259	4258	0.754	0.887
NotreDame	2798	**2803**	**2803**	0.653	0.681
Stanford	**7748**	7695	7727	0.695	0.739
YouTube	**4571**	4525	3782	0.361	0.370
DBLP	4159	4166	**4396**	0.802	0.695
LiveJournal	27067	**27113**	27072	0.869	0.887

Table 7: Statistics of Top-b Solutions

Dataset	B.	A.	G.	N.	S.	Y.	D.	L.
avg_{OLAK}	41%	34%	38%	4%	25%	36%	12%	21%
max_{OLAK}	61%	60%	66%	54%	70%	77%	46%	59%

Table 8: Coreness Gain, OLAK v.s. GAC

more potential followers. For each anchor $x \in A$, we get its ranking in all the vertices, denoted by O^x_{Deg}, O^x_{CN} and O^x_{SD}, in ascending order of degree, coreness and successive degree, respectively. Then $p_{Deg} = \frac{\sum_{x \in A} O^x_{Deg}}{|A||V(G)|}$, $p_{CN} = \frac{\sum_{x \in A} O^x_{CN}}{|A||V(G)|}$ and $p_{SD} = \frac{\sum_{x \in A} O^x_{SD}}{|A||V(G)|}$. Table 6 shows the rankings of anchors are higher than around 80% of the vertices in the graph, i.e., the anchors tend to be high-degree vertices while not the top vertices with extremely large degrees. Besides, for the anchors, we find that P_{SD} is slightly higher than P_{Deg} and P_{CN} on 7 of the 8 datasets. However, the backward reasoning is not effective, i.e., the vertices with large successive degree are not effective anchors, as shown by SD in Figure 6. Moreover, Figure 8 shows the distribution of 100 anchors (from GAC) on coreness is relatively uniform, i.e., the coreness values of the anchors can be either small, moderate, or large.

Analysis of Top-b Solutions In one iteration of the GAC algorithm, when there are more than one best anchor, all of which have the same largest coreness gain, we break the ties by the follower upper bound of the candidate anchors (Section 4.5). For clearness, we denote GAC by GAC-UB. Besides, we may use other criteria to break the ties in the greedy algorithm: choosing the vertex with the largest degree (denoted by GAC-DG), or randomly choosing a vertex (denoted by GAC-RD). As shown in Table 7, the coreness gains of different solutions (anchor sets) are very similar, where the values are denoted by $Gain_{UB}$, $Gain_{DG}$ and $Gain_{RD}$ accordingly, and the largest value for each dataset is marked in bold. Moreover, as shown in Table 7, there are many common anchors in different solutions, as the similarities (Jaccard Index) of the solutions are mostly over 0.5, where $J^{UB}_{DG} = \frac{|A_{UB} \cap A_{DG}|}{|A_{UB} \cup A_{DG}|}$ and $J^{UB}_{RD} = \frac{|A_{UB} \cap A_{RD}|}{|A_{UB} \cup A_{RD}|}$. In terms of running time, the three strategies are almost same, because the time cost to break the ties is dominated by other parts of the greedy algorithm.

Correlation with #Checkin We generate 19 different networks from Gowalla based on the user check-ins, where the i-th network is the induced subgraph by the users with at

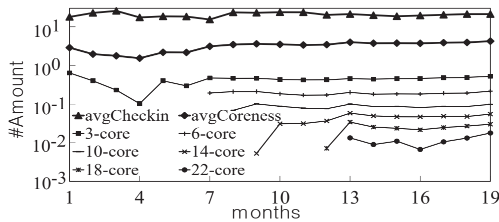
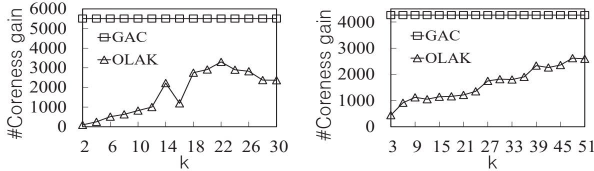
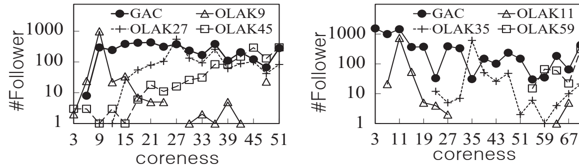


Figure 9: #Checkin, Coreness & k-Core Size



(a) Arxiv (b) Gowalla

Figure 10: Coreness Gain on Different Inputs of k



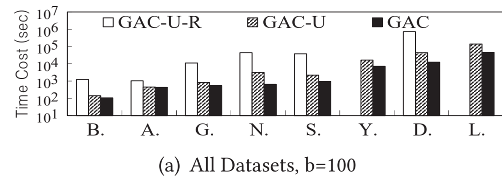
(a) Gowalla (b) Stanford

Figure 11: Distribution of Followers on Coreness

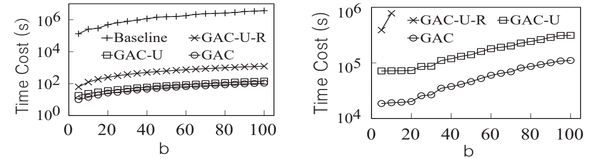
least 1 check-in during the $(i+1)$-th month, except for the first and the last months where the data is incomplete. We consider the number of user check-ins because a user with more friends may be more active in Gowalla network.

For each network, we divide the sum of #checkins, the sum of coreness, and the size of k-core, by the number of users, respectively. As shown in Figure 9, the pattern of size proportions of k-cores are more fluctuated compared to the pattern of average #checkins and average coreness, especially for large k values. However, if we choose a small k for OLAK, it generally has small coreness gain as shown in Figure 10. The pattern of average coreness over the first 7 months in Figure 9 is not similar to average #checkins, which may due to the extremely few numbers of users (less than 100) for these months. Overall, using coreness values to reinforce a social network (anchored coreness model) is more reasonable than using the size of k-core (anchored k-core model).

Comparison with OLAK. Table 8 is added to show that the largest coreness gain (denoted by max_{OLAK}) that OLAK can achieve only reaches 46%-77% of the coreness gain by GAC, on all the datasets. The largest coreness gain of OLAK is computed by running the algorithm with every possible input of k. For the anchor set A computed by OLAK, we compute the total sum of coreness gain for every coreness value and for every anchor vertex in A. Table 8 also shows that the average coreness gain (denoted by avg_{OLAK}) of OLAK for different k values is only 4%-41% of the coreness gain of GAC. Besides, Figure 10 shows that the best k for OLAK is rather different for



(a) All Datasets, b=100



(b) Brightkite (c) Livejournal

Figure 12: Time Cost of Different Algorithms

different datasets. There is no uniform preference on large, moderate, or small k values for different datasets.

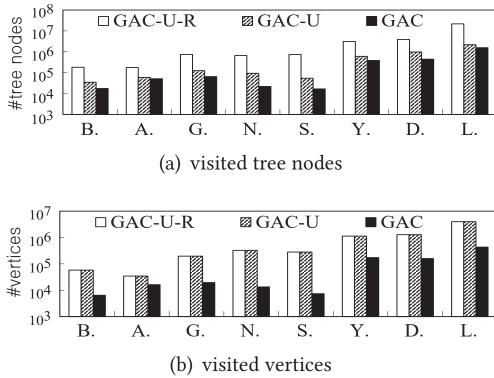
Distribution of Anchors and Followers. Figure 8 shows the distribution of 100 anchors (from GAC) on coreness is relatively uniform, compared with the anchors from OLAK, where OLAK9 denotes the anchors from OLAK with $k = 9$. Given an input k, the coreness values of the 100 anchors from OLAK can only be less than k (mostly have the coreness of $k-1$), which is consistent with the theory in [45]. Besides, Figure 11 shows the distribution of followers, which has the similar result as the distribution of anchors.

5.2 Efficiency

Overall Performance. Figure 12(a) shows the total running time of GAC, GAC-U and GAC-U-R on all the 8 datasets when $b = 100$. GAC-U-R does not return on Youtube and Livejournal after 10 days and thus the runtime is not reported. With our result-reusing mechanism (Algorithm 3), GAC-U is faster than GAC-U-R by 1 order of magnitude on average. Further benefitting from the upper bound based pruning (Section 4.5), the runtime of GAC is usually faster than GAC-U by more than 3 times. The details are as follows.

Efficient Followers Computing. Equipped with our improved algorithm for computing coreness gain of anchors (Algorithm 4), GAC-U-R is faster than Baseline by at least 1 order of magnitude on Brightkite, as shown in Figure 12(b). As it is very time-consuming to compute the coreness gain of candidate anchors using core decomposition, we can only report the runtime of Baseline on Brightkite.

Intermediate Result Reusing. By applying the core component tree (Section 4.1) and the result-reusing mechanism (Algorithm 3), GAC-U always outperforms GAC-U-R on runtime by at least 1 order of magnitude, as shown in Figure 12. Note that GAC-U-R can only find 10 anchors on Livejournal within the time limit. The scalability of GAC-U is also better than GAC-U-R in the experiments. The outperformance is because we can prune the search space by reusing the intermediate results associated with the tree nodes, when they



(a) visited tree nodes

(b) visited vertices

Figure 13: Visited Amount

keep same for one anchoring. In Figure 13(a), the number of visited tree nodes of GAC-U is around 10% of GAC-U-R.

Candidate Anchors Pruning. In Figure 12, we can see our final algorithm GAC achieves further speedup based on GAC-U when the upper bound pruning is equipped (Section 4.5). The processing time of GAC is only 20% – 30% of GAC-U because GAC reduces the search space by pruning the vertices with insufficient upper bounds of coreness gains. In Figure 13(a)-(b), the number of visited tree nodes and the number of visited vertices in GAC are much less than that in GAC-U.

6 RELATED WORK

Many cohesive subgraph models are studied in different scenarios, e.g., clique [9, 13], quasi-clique [1, 35], k-core [8, 22, 32, 36], k-truss [17, 23, 39, 42], and k-ecc [11, 50]. Among them, the k-core is widely studied with a lot of applications such as community discovery [18, 19, 28], influential spreader identification [26, 29, 30, 41], discovering protein complexes [3], recognizing hub-nodes in brain function networks [7], analyzing the structure of Internet [10], understanding software networks and its functional consequences [48], predicting structural collapse in ecosystems [34], and graph visualization [2, 49].

An in-memory algorithm for core decomposition is introduced in [4] with a time complexity of $O(m+n)$. External algorithms are proposed to handle graphs that cannot reside in the memory [12]. An I/O efficient algorithm is introduced in [43] which assumes the memory can maintain a small constant amount of data. In addition, a distributed algorithm is developed in [33] for core decomposition. Core decomposition is investigated in [25] using different frameworks to compare the performance on a single PC.

The engagement dynamics in social networks has attracted increasing attention, e.g., [6, 16, 31, 46]. The k-core model is widely applied, as its degeneration property well models the dynamic of user engagement [31]. Besides, the k-truss is also used in the study of user engagement [47], which may be feasible for particular communities.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose and study the anchored coreness problem aiming to anchor a set of vertices such that the coreness gain from all the vertices is maximized. We prove the problem is NP-hard and APX-hard. An efficient greedy algorithm is proposed with a novel tree based result reusing mechanism. We also propose effective pruning techniques to reduce the search space. Extensive experiments on 8 real-life networks demonstrate the effectiveness of our model and the efficiency of our algorithm. The reusing mechanism sheds light on the computings for other problems on hierarchical decomposition, e.g., truss decomposition. It implies that the computation can be divided into independent units and the reuse of intermediate results is feasible. The data locality and independency in the tree structure of decompositions may also inspire efficient parallel and distributed solutions.

8 PROOFS OF THEOREMS

Proof of Theorem 4.6: We prove it by contradiction. Suppose there is a non-anchor vertex $u \in V(G)$ with coreness increasing from k' to k^* after anchoring x and $k^* > k'+1$. Let M be the k^*-core after x is anchored, we have $u \in M$ and $deg(v, M) \geq k^*$ for every vertex $v \in M$. If we delete x and its corresponding edges from M, we have $deg(v, M \setminus \{x \cup E(x)\}) \geq k^* - 1$ for every $v \in M$ because at most one edge is removed for each vertex $v \in M$. Thus, $M \setminus \{x \cup E(x)\} \subseteq C_{k^*-1}(G)$. As $u \in M$ and $u \neq x$, we have $u \in C_{k^*-1}(G)$ and thus $k' \geq k^* - 1$ which contradicts with $k^* > k' + 1$. ■

Proof of Theorem 4.7: Let O denote a vertex deletion order of core decomposition on G *without* the anchoring of x. Note that the deletion order may be different when there are some vertices with same degree in the deletion procedure, while it is proved in [45] that any order following Algorithm 1 leads to the same coreness result. We denote the graph *after* anchoring x by G_x. After the anchoring of x, for every vertex $u \in V(G_x)$ with $c(u,G) < c(x,G)$, we can follow the deletion order O of G in the core decomposition of G_x, and then $c^x(u, G_x) = c(u, G)$ because the degree of u in the order keeps same when u is visited and to be deleted. Let $k' = c(x, G)$, we have $C_{k'}(G_x) = C_{k'}(G)$. Let C denote the k'-core component containing x, for every vertex $u \in \{C_{k'}(G_x) - C\}$, we have $c^x(u, G_x) = c(u, G)$ because u and x are not in the same connected component of $C_{k'}(G_x)$.

Consider a tree node TN in $\mathcal{T}$ of G with $TN.I \notin sn(x)$ and $TN.K \geq c(x, G)$. The anchoring of x may make a vertex set V_+ (from $TN.P$) increase coreness and enter $CC(TN)$. However, for each $v \in V_+$, $v \notin C_{(TN.K)+1}(G_x)$ because, 1) the coreness of a vertex can increase by at most 1 for one anchor, according to Theorem 4.6; 2) $x \notin V_+$ otherwise $TN.I \in sn(x)$ which contradicts the assumption. Thus, if we delete the vertices in V_+ before $TN.V$ in core decomposition, each vertex

$u \in TN.V$ has the same degree as in O when u is visited and to be deleted, i.e., $c^x(u, G_x) = c(u, G)$. Thus, only the vertices in $\bigcup_{id \in sn(x)} \mathcal{T}[id].V$ may be the followers of x. ■

Proof of Lemma 4.8: We prove it by contradiction. To prove 1), suppose an $id \in rn(u)$ has $id \notin sn'(u)$. That means ⓐ $\mathcal{T}'[id].V$ does not contain any neighbor of u or ⓑ $\mathcal{T}'[id].V$ contains the neighbors of u but also contains another vertex whose $id' < id$ so $\mathcal{T}'[id].I = id'$ and $id' \in sn'(u)$.

For ⓐ, if vertex id itself did not increase its coreness, then the neighbors of u in $\mathcal{T}[id].V$ must have increased their coreness and left $\mathcal{T}[id].V$. So these neighbors belong to V_x (Line 1 of Algorithm 3) and they are used to erase id at Line 3-6, which contradicts with $id \in rn(u)$; If id increased its coreness, id would make all the vertices in $\mathcal{T}[id].V$ belong to V_x (Line 1), and then id is erased from $rn(u)$ at Line 3-6 which contradicts with $id \in rn(u)$. For ⓑ, if vertex id did not increase its coreness, it means there is a vertex v with coreness increased and then joined $\mathcal{T}'[id].V$. For such v, $v \in V_x$ which makes the neighbors of u in $\mathcal{T}'[id].V$ be included in V'_x (Line 12). So, id is erased from $rn(u)$ at Line 13-16 which contradicts with $id \in rn(u)$; If id increased its coreness, it contradicts with $id \in rn(u)$ because of the same reason when id increased its coreness in ⓐ.

To prove 2), suppose there is an $id \in rn(u)$ having $\mathcal{T}'[id].K \neq \mathcal{T}[id].K$, that means vertex id must have increased its coreness. So, all the vertices of $\mathcal{T}[id].V$ belong to V_x (Line 1) which erased id from $rn(u)$ at Line 3-6 and contradicts with $id \in rn(u)$.

To prove 3), suppose there is an $id \in rn(u)$ having $\mathcal{T}'[id].V \neq \mathcal{T}[id].V$. We already proved that $id \in sn'(u)$ and $\mathcal{T}'[id].K = \mathcal{T}[id].K$. Thus, there must be ⓒ a vertex $v \in \mathcal{T}[id].V$ increased $c(v)$ and then left $\mathcal{T}[id].V$, or ⓓ a vertex v joined in $\mathcal{T}'[id].V$ because its coreness increased.

For ⓒ, v can make all vertices of $\mathcal{T}[id].V$ belong to V_x (Line 1) then erase id (Line 3-6). For ⓓ, v can make u's neighbors in $\mathcal{T}'[id].V$ belong to V'_x (Line 12) and can erase id (Line 13-16). Both ⓒ and ⓓ contradict with $id \in rn(u)$. ■

Proof of Theorem 4.9: Let O denote a vertex deletion order of core decomposition on G *without* anchoring x. Similar to the proof of Theorem 4.7, we follow the deletion order O in the core decomposition of G_x. Let $k^* = \mathcal{T}'[id].K$. The anchoring of x may make a vertex set V_+ (from $\mathcal{T}[id].P$) increase coreness so enter $CC(\mathcal{T}'[id])$, but for each $v \in V_+$, $v \notin C_{k^*+1}(G_x)$ since the coreness of a vertex can increase by at most 1 for one anchor according to Theorem 4.6. Also, we have $\mathcal{T}'[id].K = \mathcal{T}[id].K$ and $\mathcal{T}'[id].V = \mathcal{T}[id].V$ from Lemma 4.8. Thus, $V+ = \emptyset$. Now we conclude each vertex $u \in \mathcal{T}'[id].V$ has the same degree as in O when u is visited and to be deleted in core decomposition of G_x, i.e., $c^x(u, G_x) = c(u, G)$. So the followers of x at node id keeps same after anchoring x. ■

Proof of Theorem 4.14: Before the anchoring of x in G, let $k = c(u, G)$, all the neighbors of u in G are classified into three sets: N_u^0 contains every neighbor v with $\mathcal{P}[v].k < \mathcal{P}[u].k$, i.e., $c(v, G) < c(u, G)$; N_u^1 contains every neighbor v with $\mathcal{P}[v].k = \mathcal{P}[u].k$ and $\mathcal{P}[v].i < \mathcal{P}[u].i$; and N_u^2 contains the other neighbors of u. (i) Suppose $x \in N_u^0 \cup N_u^1$, (x, u) itself is an upstair path from x to u. (ii) Suppose $x \in N_u^2$, let O denote a vertex deletion order of core decomposition on G without any anchors (Algorithm 1). We denote the graph after anchoring x by G_x. For every vertex $v \in V(G_x)$ with $\mathcal{P}[v] \prec \mathcal{P}[x]$, we can follow the same deletion order O in the core decomposition of G_x, and then $c^x(v, G_x) = c(v, G)$ because the degree of v in the order keeps same when v is visited and to be deleted. Thus, $c^x(u, G_x) = c(u, G)$ and u is not a follower of x if $x \in N_u^2$. So $x \notin N_u^2$. (iii) Suppose $x \notin N_u^0 \cup N_u^1 \cup N_u^2$, u must have a neighbor $v_0 \in N_u^1 \cap C_{k+1}(G_x)$; otherwise, $c^x(u, G_x) = c(u, G)$ as in case (ii) following the deletion order O. Thus, if a vertex $v_i \in C_{k+1}(G_x) \setminus C_{k+1}(G)$, v_i must have a neighbor $v_{i+1} \in N_{v_i}^1 \cap C_{k+1}(G_x)$ or $v_{i+1} = x$. Recursively, $u \in \mathcal{F}(x)$ implies there is a path $(x, ..., u)$ which is an upstair path from x to u where each vertex in the path is a follower of x except x itself. ■

Proof of Theorem 4.15: We denote the graph after anchoring x by G_x, and let $k^+ = c(u, G) + 1$. We show if $d^+(u) < c(u, G) + 1$, then $deg(u, C_{k^+}(G_x)) < k^+$, so u cannot be a follower of x. u's neighous can be divided into those in $\bigcup_{id \in pn(u)} tca[u][id]$, $tca_=^{\leq}(u) \cup tca_=^{\geq}(u)$ and $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$, respectively. Obviously the neighbors of $\bigcup_{id \in pn(u)} tca[u][id]$ are not in $C_{k^+}(G_x)$, because they cannot increase the coreness by 2 according to Theorem 4.6. For the neighbors in $tca_=^{\leq}(u) \cup tca_=^{\geq}(u)$, they are all considered in $d_s^+(u)$ or $d_u^+(u)$, unless they are *discarded* or never pushed to H, both of which mean they are not in $C_{k^+}(G_x)$. At last, for the neighbors in $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$, they satisfy $|\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]| = d_>(u)$. Since $d^+(u)$ considers all the neighbors of u which are possible to be in $C_{k^+}(G_x)$, $d^+(u)$ is a degree bound of $deg(u, C_{k^+}(G_x))$. ■

Proof of Theorem 4.17: According to the Equation 1 and 2, all the vertices of $\bigcup_{id \in sn(x)} \mathcal{T}[id].V$ which are reachable by x via upstair paths are counted at least once in the equations. Therefore, based on Theorem 4.14, we can prove that $|F[x][i_x]| \leq UB_{i_x}(x)$ and $|F[x][id]| \leq UB_{id}^{>}(x)$ for each $id \in sn(x) \setminus \{i_x\}$. Then, based on Equation 3 and Theorem 4.7, we can conclude that $g(\{x\}, G) \leq UB_\sigma(x)$. ■

ACKNOWLEDGMENTS

Fan Zhang is supported by RQ2020090. Xuemin Lin is supported by 2018YFB1003504, NSFC61232006, ARC DP180103096 and DP170101628. Wenjie Zhang is supported by ARC DP180103096. Ying Zhang is supported by ARC DP180103096 and FT170100128.

REFERENCES

[1] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. 2002. Massive Quasi-Clique Detection. In *LATIN*. 598–612.

[2] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NeurIPS*. 41–50.

[3] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4 (2003), 2.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[5] Kshipra Bhawalkar, Jon M. Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. 2012. Preventing Unraveling in Social Networks: The Anchored k-Core Problem. In *ICALP*. 440–451.

[6] Kshipra Bhawalkar, Jon M. Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. 2015. Preventing Unraveling in Social Networks: The Anchored k-Core Problem. *SIAM J. Discrete Math.* 29, 3 (2015), 1452–1475.

[7] Michal Bola and Bernhard A. Sabel. 2015. Dynamic reorganization of brain functional networks during cognition. *NeuroImage* 114 (2015), 398–413.

[8] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.

[9] Coenraad Bron and Joep Kerbosch. 1973. Finding All Cliques of an Undirected Graph (Algorithm 457). *Commun. ACM* 16, 9 (1973), 575–576.

[10] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. 2007. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences* 104, 27 (2007), 11150–11154.

[11] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216.

[12] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*. 51–62.

[13] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2010. Finding maximal cliques in massive networks by H*-graph. In *SIGMOD*. 447–458.

[14] Rajesh Chitnis, Fedor V. Fomin, and Petr A. Golovach. 2016. Parameterized complexity of the anchored k-core problem for directed graphs. *Inf. Comput.* 247 (2016), 11–22.

[15] Rajesh Hemant Chitnis, Fedor V. Fomin, and Petr A. Golovach. 2013. Preventing Unraveling in Social Networks Gets Harder. In *AAAI*.

[16] M. S.-Y. Chwe. 2000. Communication and coordination in social networks. *The Review of Economic Studies* 67, 1 (2000), 1–16.

[17] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–1.

[18] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and classification of dense implicit communities in the Web graph. *TWEB* 3, 2 (2009), 7:1–7:36.

[19] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *PVLDB* 10, 6 (2017), 709–720.

[20] Uriel Feige. 1998. A Threshold of ln n for Approximating Set Cover. *J. ACM* 45, 4 (1998), 634–652.

[21] David García, Pavlin Mavrodiev, and Frank Schweitzer. 2013. Social resilience in online communities: the autopsy of friendster. In *Conference on Online Social Networks*. 39–50.

[22] Christos Giatsidis, Fragkiskos D. Malliaros, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2014. CoreCluster: A Degeneracy Based Graph Clustering Framework. In *AAAI*. 44–50.

[23] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[24] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*. 85–103.

[25] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *PVLDB* 9, 1 (2015), 13–23.

[26] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888.

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. 457–472.

[29] Jian-Hong Lin, Qiang Guo, Wen-Zhao Dong, Li-Ying Tang, and Jian-Guo Liu. 2014. Identifying the node spreading influence with largest k-core values. *Physics Letters A* 378, 45 (2014), 3279–3284.

[30] Fragkiskos D Malliaros, Maria-Evgenia G Rossi, and Michalis Vazirgiannis. 2016. Locating influential nodes in complex networks. *Scientific reports* 6 (2016), 19307.

[31] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*. 469–478.

[32] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[33] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *IEEE Trans. Parallel Distrib. Syst.* 24, 2 (2013), 288–300.

[34] Flaviano Morone, Gino Del Ferraro, and Hernán A Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics* 15, 1 (2019), 95.

[35] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *SIGKDD*. 228–238.

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

[37] Kazunori Seki and Masataka Nakamura. 2016. The collapse of the Friendster network started from the center of the core. In *ASONAM*. 477–484.

[38] Kazunori Seki and Masataka Nakamura. 2017. The mechanism of collapse of the Friendster network: What can we learn from the core structure of Friendster? *Social Netw. Analys. Mining* 7, 1 (2017), 10:1–10:21.

[39] Yingxia Shao, Lei Chen, and Bin Cui. 2014. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*. 613–624.

[40] Babak Tootoonchi, Venkatesh Srinivasan, and Alex Thomo. 2017. Efficient Implementation of Anchored 2-core Algorithm. In *ASONAM*. 1009–1016.

[41] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon M. Kleinberg. 2012. Structural diversity in social contagion. *Proc. Natl. Acad. Sci. U.S.A.* 109, 16 (2012), 5962–5966.

[42] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.

[43] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient Core Graph Decomposition at web scale. In *ICDE*. 133–144.

[44] Shaomei Wu, Atish Das Sarma, Alex Fabrikant, Silvio Lattanzi, and Andrew Tomkins. 2013. Arrival and departure dynamics in social networks. In *WSDM*. 233–242.

[45] Fan Zhang, Wenjie Zhang, Ying Zhang, Lu Qin, and Xuemin Lin. 2017. OLAK: An Efficient Algorithm to Prevent Unraveling in Social Networks. *PVLDB* 10, 6 (2017), 649–660.
[46] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. Finding Critical Users for Social Network Engagement: The Collapsed k-Core Problem. In *AAAI*. 245–251.
[47] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2018. Efficiently Reinforcing Social Networks over User Engagement and Tie Strength. In *ICDE*. 557–568.
[48] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing* 53, 2 (2010), 352–369.
[49] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *PVLDB* 6, 2 (2012), 85–96.
[50] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*. 480–491.