The VLDB Journal
https://doi.org/10.1007/s00778-021-00673-6

SPECIAL ISSUE PAPER

Anchored coreness: efficient reinforcement of social networks

Qingyuan Linghu[1] · Fan Zhang[2] · Xuemin Lin[1] · Wenjie Zhang[1] · Ying Zhang[3]

Received: 24 August 2020 / Revised: 15 February 2021 / Accepted: 24 April 2021

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 greedy algorithm is proposed with novel techniques on pruning search space and reusing the intermediate results. The algorithm is also extended to distributed environment with a novel graph partition strategy to ensure the computing independency of each machine. Extensive experiments on real-life data demonstrate that our model is effective for reinforcing social networks and our algorithms are efficient.

Keywords Social network · User engagement · Network stability · Core decomposition · Distributed algorithm

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 [46]. 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 [31,54].

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 [9]. The k-core is defined as the maximal subgraph in which every vertex has at least k neighbors in the subgraph [48,52]. 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., [46,53,58].

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* [9,10]: 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

✉ Fan Zhang
fanzhang.cs@gmail.com

Qingyuan Linghu
q.linghu@unsw.edu.au

Xuemin Lin
lxue@cse.unsw.edu.au

Wenjie Zhang
zhangw@cse.unsw.edu.au

Ying Zhang
ying.zhang@uts.edu.au

1 University of New South Wales, Sydney, Australia

2 Guangzhou University, Guangzhou, China

3 Centre for AI, University of Technology Sydney, Sydney, Australia

Published online: 31 May 2021

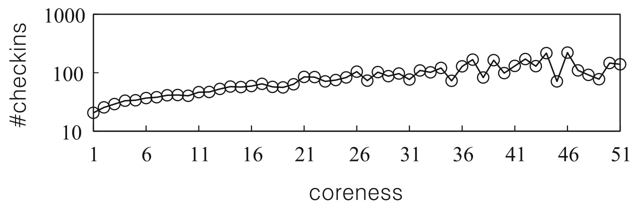


Fig. 1 Check-in number versus coreness value

in the network and support the engagement of other users [10]. The anchored k-core problem has been further studied on different aspects, e.g., the theoretical side [20,21], the experimental evaluation [31,62] and the efficient solutions [57,65].

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 [65], 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 [65]. Moreover, it is very hard to determine a good input value of k for the AK problem.

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) [53] or the outside of center cores [31], i.e., the collapse happens in a “global” way. As shown in [46], a user's coreness 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 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 equal to k. As shown in Fig. 1, the coreness value and check-in number in Gowalla [41] are in a positive correlation, except for the disturbance on the center cores due to the small sample. So 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 (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 itself.

Example 1 Figure 2 shows a graph G of 13 vertices and their connections. The coreness of each vertex is marked 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

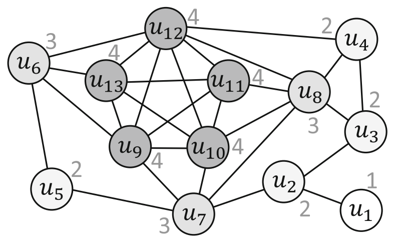


Fig. 2 A toy example

Table 1 Anchored k-core versus 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

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 [8], 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 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 [65], 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 [65] 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 mechanism (Sect. 5.3) 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 $\mathcal{T}$ (Sect. 5.1) of core decomposition [8] 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 (Sect. 5.4) 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* (Sect. 5.5) of coreness gain to further prune candidate anchors and well match the technique with the reusing mechanism to improve efficiency. Combining all these techniques, we propose the serial greedy algorithm GAC (Sect. 5.5) which is conducted in single-machine computing environment.

We then extend GAC to DGAC which is conducted in distributed computing environment. In order to reduce the communication cost among the machines, we propose a graph partition strategy where the graph can be divided into *shell component partitions* (partitions) induced by the subgraphs of *k-shell component* in the structure $\mathcal{SP}$ (Sect. 6.1). Therefore, for anchoring a vertex x, the followers of x are divided into different partitions in $\mathcal{SP}$. We prove x's followers from different partitions are not overlapped and the computation of followers from different partitions can be conducted concurrently and independently. We also show the upper bound proposed in GAC is a reasonable estimate of the time cost of computing x's followers. Based on these, we propose a *computing resource scheduling* (Sect. 6.3) to make the machines evenly and independently have computing tasks. Similar to the reuse mechanism of GAC, the shell component partitions become the units of deciding whether the associated computed results are reused.

Contributions In the paper, we overcome all the challenges with above solutions. The preliminary version is published in [44]. Our main 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.
- We propose a serial greedy algorithm for single-machine environment with novel techniques. With the tree of core decomposition, we introduce a 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.
- We propose a distributed greedy algorithm for anchored coreness problem in distributed computing environment. With the graph partition strategy based on k-shell component, all the machines can independently and concurrently conduct computations, i.e., the coreness gain computations of vertices are divided into independent units regarding k-shell components. Our computing resource scheduling strategy ensures the communication cost across machines is limited and computing tasks are evenly distributed. The techniques of reuse mechanism, computation and upper bound of coreness gain in the serial algorithm are specifically designed for our distributed algorithm.
- Comprehensive experiments are conducted on 8 real-life datasets to show that (1) the proposed serial algorithm GAC is more effective than 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 for GAC largely improve the algorithm efficiency. (5) The proposed distributed algorithm DGAC is significantly more efficient than GAC, and the time cost is inversely proportional to the number of machines in general.

2 Related work

Many cohesive subgraph models are studied in different scenarios, e.g., clique [14,19], quasi-clique [3,51], k-core [13,32,48,52], k-truss [23,34,55,59], k-plexes [24,25,71], and k-ecc [17,70]. Among them, the k-core is widely studied with a lot of applications such as community discovery [27,29,42], influential spreader identification [40,43,45,58], discovering protein complexes [7], recognizing hub-nodes in brain function networks [12], analyzing the structure of Internet [15], understanding software networks and its functional consequences [68], predicting structural collapse in ecosystems [50], and graph visualization [5,69].

User engagement study in social networks has attracted increasing attention, e.g., [10,22,46,64,66]. The k-core model is widely applied, as its degeneration property well captures the dynamic of user engagement [44,46,65,72]. Besides, the k-truss is also investigated in user engagement study focusing on denser subgraphs [67].

An in-memory algorithm for core decomposition is introduced in [8] with a time complexity of $\mathcal{O}(m+n)$. External algorithms are proposed to handle graphs that cannot reside in the memory [18]. An I/O efficient algorithm is introduced in [61] which assumes the memory can maintain a small constant amount of data. In addition, a distributed algorithm is developed in [49] for core decomposition. Core decomposition is investigated in [39] using different frameworks to compare the performance on a single PC.

Some works about core decomposition in parallel environments have been studied recently [16,28,36,47,49]. An algorithm for core decomposition on multicore platforms is introduced in [36]. In [16], an distributed $2(1+\epsilon)$ approximate algorithm of core decomposition is proposed. Based on the distributed framework Spark [63], Mandal and Hasan [47] uses the think-like-a-vertex paradigm to conduct core decomposition. Toward the incremental core decomposition, so-called core maintenance, some distributed or parallel algorithms are also proposed [4,6,33,35]. Hua et al. [33] proposes a structure called *joint edge set* to parallelize inserting/deleting a set of edges, which is based on the idea of *matching* in [35].

There are many works doing parallel computation of other cohesive subgraph models such as clique, k-plexes and k-truss. An algorithm implemented on shared-memory multicore machine is introduced for the maximal clique enumeration (MCE) problem [26]. Wang et al. [60] proposes an approach for maximal clique and k-plexes enumeration at the same time, which identifies dense subgraphs by binary graph partitioning, and it is implemented on MapReduce. In [24], a shared-nothing distributed algorithm for k-plexes enumeration is proposed, but only limited to $k = 2$. [25] presents D2K, which exploits the fact that large enough k-plexes have diameter 2, so that the distributed implementation can handle very large graphs. For k-truss model, [37] and [56] develop the parallel algorithms for k-truss decomposition on multicore (shared-memory) system. In [11], a performance exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU is presented.

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 neighbor vertex set 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 ith 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 $\|\mathcal{F}(x)\|$

3 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 vertex 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 1 ***k*-core** [48,52]. 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.

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

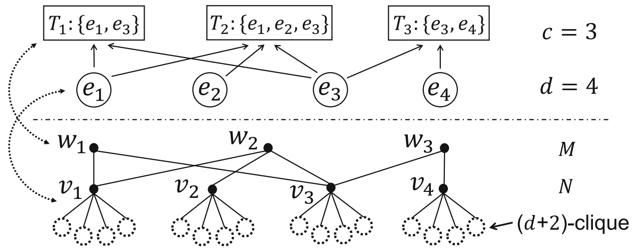


Fig. 3 Construction example for hardness proofs

Definition 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 3 **core decomposition**. Given a graph G, core decomposition of G is to compute the coreness of every vertex in $V(G)$.

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 [8], 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 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 $V(G)\backslash A$, i.e., $g(A, G) = \sum_{u \in V(G)\backslash A}(c^A(u) - c(u))$.

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.

4 Problem analysis

Theorem 1 *Given a graph G, the anchored coreness problem is NP-hard.*

Proof We reduce the maximum coverage (MC) problem [38], 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 an example of 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.

Assume each element in H is contained by at least 1 set, for each $w_i \in M$ and $v_j \in (V(G)\backslash M)$, 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 $k = 1, 2, \ldots$ 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 the 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 the 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 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 1. For any $\epsilon > 0$, the MC problem cannot be approximated in polynomial time within a ratio of $(1 - 1/e + \epsilon)$, unless $P = NP$ [30]. 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 *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 $B = \{v_6\}$, $g(A)+g(B) = 0 < g(A \cup B)+g(A \cap B) = 4$. □

5 A greedy 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 (Sect. 5.1), and the theorems of finding the candidate followers which may increase the coreness due to the anchoring (Sect. 5.2). Based on the tree and the theorems, we propose a mechanism to reuse the intermediate results across iterations (Sect. 5.3), and the algorithm to compute the coreness gain of one anchor by partially exploring the tree (Sect. 5.4). Combining the above with an upper bound technique for candidate anchors pruning, our final GAC algorithm is presented (Sect. 5.5).

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

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

5.1 Core component tree

Definition 5 **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.

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 all the vertices in $TN.V$ have coreness equal to $TN.K$. We assume each vertex in $V(G)$ has a positive integer id as its unique identifier, i.e., $id \in [1, V(G)] \wedge id \in \mathbb{N}$. Let $TN.I$ denote the smallest vertex id from the vertices in $TN.V$. We use $TN.P$ to denote the

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 $c(v) \geq c(u) \ \wedge \ \mathcal{T}[v].I = id$
$pn(u)$	The tree node id set where $id \in pn(u)$ iff $\exists v \in N(u)$ having $c(v) < c(u) \ \wedge \ \mathcal{T}[v].I = id$
$F[x][id]$	The follower set of x at tree node id, i.e., $v \in F[x][id]$ iff $v \in \mathcal{F}(x) \ \wedge \ \mathcal{T}[v].I = id$

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 [48]. 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 6 **tree node classified adjacency** (tca)**.** For a given graph G, we scan the neighbor vertex set 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 7 **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 $c(v) \geq c(u) \ \wedge \ \mathcal{T}[v].I = id$.

Definition 8 **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 $c(v) < c(u) \ \wedge \ \mathcal{T}[v].I = id$.

Example 2 In Fig. 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 $\mathcal{T}$.

5.2 Restriction of candidate followers

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

Theorem 4 *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 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 $\deg(v, M \backslash \{x \cup E(x)\}) \geq k^* - 1$ for every $v \in M$ because at most one edge is removed for each vertex $v \in M$. Thus, $M \backslash \{x \cup E(x)\} \subseteq C_{k^*-1}(G)$. 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 $k^* > k' + 1$. □

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, $g(\{x\}) = |\mathcal{F}(x)|$. 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 $v \in \mathcal{F}(x) \wedge \mathcal{T}[v].I = id$.

A fast method to compute the followers will be introduced in Sect. 5.4. 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 $\mathcal{O}(m)$. The candidate followers of a vertex x can be extracted as follows.

Theorem 5 *If a vertex x is anchored in the graph G, we have $\mathcal{F}(x) \subset \bigcup_{id \in sn(x)} \mathcal{T}[id].V$.*

Proof 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 [65] 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 $c^x(u, G_x) = c(u, G)$ since 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; (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. □

5.3 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 5, 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 $\mathcal{T}$ can find other vertices which may be affected w.r.t x (Line 12–13). Similar to Lines 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 easily avoid duplicate removals in $rn(u)$ triggered by u's neighbors using tree node tags.

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

Algorithm 1 (Line 8) and Algorithm 2 (Line 9) both have $\mathcal{O}(m)$ time complexity. In Lines 3–6 and Lines 13–16, each edge is accessed at most one time, respectively. So, the time complexity of Algorithm 3 is $\mathcal{O}(m)$.

Lemma 1 *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) $\mathcal{T}'[id].V = \mathcal{T}[id].V$.*

Proof 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 Lines 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 Lines 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

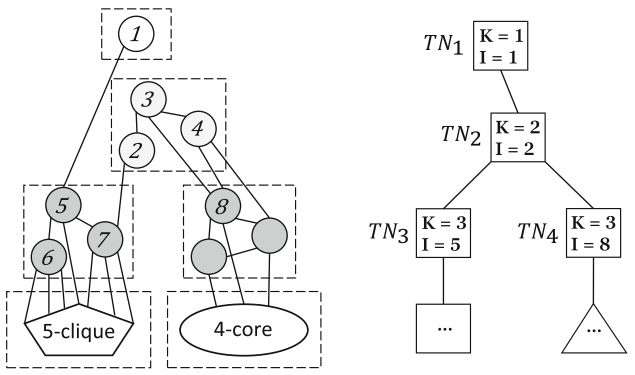


Fig. 4 Core component tree

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 Lines 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 $id \in rn(u)$. □

Theorem 6 *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 $v \in F[u][id]$.*

Proof Let O denote a vertex deletion order of core decomposition on G *without* anchoring x. Similar to the proof of Theorem 5, 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. Also, we have $\mathcal{T}'[id].K = \mathcal{T}[id].K$ and $\mathcal{T}'[id].V = \mathcal{T}[id].V$ from Lemma 1. 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 the same after anchoring x. □

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 5. 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 6. 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 3 In Fig. 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 $2 \in rn(\mathbf{2})$.

5.4 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 ith batch. Specifically, when $i = 1$, H_k^i is defined as $\{u \mid \deg(u, C_k(G)) < k + 1 \wedge u \in C_k(G)\}$. The deletion of the 1st-layer will produce the 2nd-layer. Recursively, when $i > 1$, $H_k^i = \{u \mid \deg(u, G_i) < k + 1 \wedge u \in G_i\}$ 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 ith 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 $\mathcal{P}[v_i].k = \mathcal{P}[v_j].k \wedge \mathcal{P}[v_i].i < \mathcal{P}[v_j].i$.

Example 4 In Fig. 5a, 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.

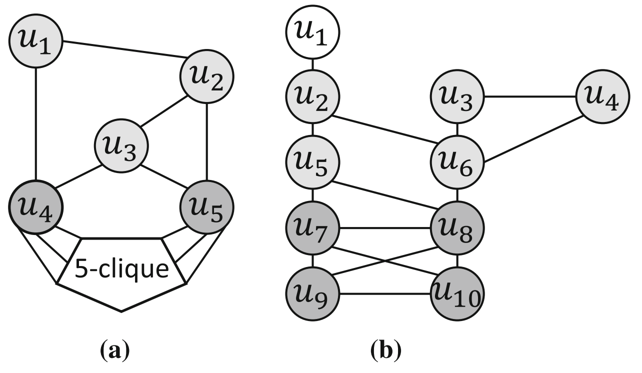


Fig. 5 Figures for Examples 4, 5, 6 and 7

Definition 9 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 5 In Fig. 5b, we can compute the shell-layer pairs of the vertices and get $\mathcal{P}[u_1] = (1, 1)$, $\mathcal{P}[u_2] = \mathcal{P}[u_3] = \mathcal{P}[u_4] = (2, 1)$, $\mathcal{P}[u_5] = \mathcal{P}[u_6] = (2, 2)$ 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 9. 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 9), 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 9.

Theorem 7 *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.*

Proof 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 $v_i \in C_{k+1}(G_x) \backslash C_{k+1}(G)$, 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 $(x, \ldots, u)$ which is an upstair path from x to u where each vertex in the path is a follower of x except x itself. □

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);
```

Computing Followers According to Theorem 7, 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 $tca^{\leqq}(u)$ to denote the set of u's neighbors where each neighbor v has $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i \leq \mathcal{P}[u].i$. Similarly, $tca^{\geqq}(u)$ contains every u's neighbor v with $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i > \mathcal{P}[u].i$. 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, $tca^{\leqq}(u)$ and $tca^{\geqq}(u)$ 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 (Sect. 5.3) 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 (Lines 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 $id \in sn(x) \backslash rn(x)$, we explore $CF(x)$ in a layer-by-layer manner: from jth 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 not be visited again, and a *survived* vertex may become *discarded* later due to the deletion cascade. The vertices that are 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 $\mathcal{O}(m)$, because each edge is accessed 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 $\{x\} \cup (tca_{\equiv}^{\leq}(u) \cap H) \cup tca_{\equiv}^{\geq}(u)$, and $d_>(u)$ is the number of neighbors in $\bigcup_{id \in sn(u) \backslash \{i_u\}} tca[u][id]$. 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 8 *A vertex $u \in CF(x)$ cannot be a follower of x if $d^+(u) < c(u, G) + 1$.*

Proof 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 neighbors can be divided into those in $\bigcup_{id \in pn(u)} tca[u][id]$, $tca_{\equiv}^{\leq}(u) \cup tca_{\equiv}^{\geq}(u)$ and $\bigcup_{id \in sn(u) \backslash \{i_u\}} tca[u][id]$, 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. For the neighbors in $tca_{\equiv}^{\leq}(u) \cup tca_{\equiv}^{\geq}(u)$, 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 $\bigcup_{id \in sn(u) \backslash \{i_u\}} tca[u][id]$, they satisfy $|\bigcup_{id \in sn(u) \backslash \{i_u\}} tca[u][id]| = d_>(u)$. 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 $\deg(u, C_{k^+}(G_x))$. □

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

Example 6 In Fig. 5b, 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 $tca_{\equiv}^{\geq}(u_2)$ 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 $tca_{\equiv}^{\geq}(u_5) = \emptyset$, 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 $tca_{\equiv}^{\leq}(u_6)$, 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 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 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 (Lines 3–17, Algorithm 4) again, and use the existing $F[x][id]$ instead. If $id \notin rn(x)$, we execute the Lines 3–17 of Algorithm 4 to find the correct followers.

5.5 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

Eq. 1, we firstly get the upper bound of followers from its own tree node $\mathcal{T}[x]$. Then for each $id \in sn(x)\backslash\{i_x\}$, we get an upper bound $UB^{>}_{id}(x)$ by Eq. 2. At last we can compute the total upper bound $UB_\sigma(x)$ by Eq. 3. When $tca^{\geq}_{=}(u) = \emptyset$ for a vertex u, we set $UB_{i_u}(u)$ to 0.

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

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

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

Theorem 9 *Given a graph G and an anchor vertex x, $|F[x][i_x]| \leq UB_{i_x}(x)$, and for each $id \in sn(x)\backslash\{i_x\}$, $|F[x][id]| \leq UB^{>}_{id}(x)$. So, $g(\{x\}, G) \leq UB_\sigma(x)$.*

Proof According to Eqs. 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 7, we can prove that $|F[x][i_x]| \leq UB_{i_x}(x)$ and $|F[x][id]| \leq UB^{>}_{id}(x)$ for each $id \in sn(x)\backslash\{i_x\}$. Then, based on Eq. 3 and Theorem 5, we can conclude that $g(\{x\}, G) \leq UB_\sigma(x)$. □

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 $\mathcal{O}(m)$.

Example 7 In Fig. 5a, 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 $UB_{u_1}(u_1) = (UB_{u_1}(u_2) + 1) = 2$, $UB^{>}_{u_4}(u_1) = (UB_{u_4}(u_4) + 1) = 1$, $UB_\sigma(u_1) = UB_{u_1}(u_1) + UB^{>}_{u_4}(u_1) = 3$.

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

```
Input  : $G$ : a social network, $b$ : number of anchors
Output : $A$ : the set of anchor vertices
1  CoreDecomp($G, \emptyset$);
2  $\mathcal{T} \leftarrow$ BuildCCT($G, root$);
3  Compute upper bounds of follower numbers;
4  for $i$ from 1 to $b$ do
5      $\lambda := -1$; $a := null$;
6      for each $u \in V(G)$ with decreasing order $UB_\sigma(u)$ do
7          if $u \notin A$ and $UB_\sigma(u) > \lambda$ then
8              $F[u]$ := FindFollowers($u, G, \mathcal{T}$);
9              if $|\mathcal{F}[u]| > \lambda$ then
10                 $a := u$; $\lambda := |\mathcal{F}[u]|$;
11     $A := A \cup \{a\}$; $deg(a, G) := +\infty$;
12     ResultReuse($a, G, \mathcal{T}$);
13     Refine upper bounds;
14 return $A$
```

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 (Lines 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 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 (Lines 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).

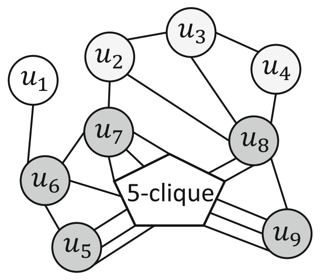


Fig. 6 k-shell component

6 Distributed greedy algorithm

We introduce a distributed greedy anchored coreness algorithm, DGAC, in which a master machine is responsible for resource scheduling and multiple slave machines are responsible for specific computing tasks. DGAC can further parallelize the computation via the multithreads of each machine.

6.1 Shell component partition

In this subsection, we introduce the graph partition algorithm and its maintenance algorithm after each iteration of the greedy strategy. We firstly define k-shell component, followed by the definitions regarding *shell component partition* which can divide the data graph into fine-grained units, thus helps balance the computation among all the machines in distributed setting. Examples 8 and 9 are the instances to illustrate k-shell component and shell component partition, respectively. Based on these, we introduce Algorithm 7 (graph partition algorithm) and Algorithm 9 (partition maintenance algorithm) and then prove their correctness.

Definition 10 ***k*-shell component**. Given a graph G and the k-shell $H_k(G)$, a subgraph S_k^i is the ith k-shell component of $H_k(G)$, if S_k^i is a maximal induced connected component of $H_k(G)$.

Example 8 In Fig. 6, we have $H_1(G) = \{u_1\}$, $H_2(G) = \{u_2, u_3, u_4\}$ and $H_3(G) = \{u_5, u_6, u_7, u_8, u_9\}$. Within $H_1(G)$, S_1^1 is the only k-shell component with $V(S_1^1) = \{u_1\}$. Within $H_2(G)$, S_2^1 is the only k-shell component with $V(S_2^1) = \{u_2, u_3, u_4\}$. But within $H_3(G)$, we have two k-shell components S_3^1 and S_3^2, in which $V(S_3^1) = \{u_8, u_9\}$ and $V(S_3^2) = \{u_5, u_6, u_7\}$.

Shell Component Partition $(SC, SF, \mathcal{SP})$ We use SC to denote a *shell component partition* affiliated to one k-shell component S_k^i, then SC has the following domains:

(1) $SC.V$, having $u \in SC.V$ iff. $u \in V(S_k^i)$;
(2) $SC.V^-$, having $u \in SC.V^-$ iff. $c(u, G) < k$ and $\exists (u, v) \in E(G) \wedge v \in SC.V$;
(3) $SC.E$, having $(u, v) \in SC.E$ iff. $u \in SC.V \wedge v \in SC.V \cup SC.V^- \wedge (u, v) \in E(G)$;
(4) $SC.h$, having for each $u \in SC.V$, $SC.h[u] = |\{v|(u, v) \in E(G) \wedge c(v, G) > c(u, G)\}|$;
(5) $SC.C$, the candidate anchor set in SC, which will be explained in Sect. 6.3 in detail.
(6) $\mathcal{SP}[u]$, the set of all the shell component partitions having u, i.e., $\mathcal{SP}[u] = \{SC \mid u \in SC.V \cup SC.V^-\}$.
(7) We use $SF[u][SC]$ to denote the follower set of u in SC, i.e., $SF[u][SC] = \{v \mid v \in \mathcal{F}(u) \wedge v \in SC.V\}$.

All the notations regarding shell component partition are summarized in Table 4, and other notations for DGAC are summarized in Table 5.

Table 4 Summary of notations for $\mathcal{SP}$, SC

Notation	Definition
SC	A shell component partition affiliated to S_k^i
$SC.V$	The set of vertices with coreness equal to k of S_k^i
$SC.V^-$	The set of vertices with coreness less than k of S_k^i
$SC.E$	The set of edges in SC
$SC.h[u]$	Number of u's neighbors having higher coreness
$SC.C$	The anchor candidates set in SC
$SF[u][SC]$	The follower set of u in SC
$\mathcal{SP}[u]$	The set of shell component partitions containing u

Table 5 Other notations for DGAC

Notation	Definition
S_k^i	The ith k-shell component
V_x	$(\bigcup_{SC\in\mathcal{SP}[x]} SC.V)\backslash\{x\}$
G_x	The graph G with x anchored
G_{SC}	The subgraph formed by $SC.V$, $SC.V^-$, $SC.E$
$N^{\le}(u, SC)$; $N^{>}(u, SC)$	Set of such u's neighbor v in SC with $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i \le \mathcal{P}[u].i$ or $\mathcal{P}[v].k < \mathcal{P}[u].k$ (resp. $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i > \mathcal{P}[u].i$ or $\mathcal{P}[v].k > \mathcal{P}[u].k$)
$d_P^+(u, SC)$	The degree bound of u in partition SC
$U_{SC}(u)$	The upper bound of u's followers in SC
N_S	The number of slave machines
N_T	The number of threads within each slave machine
Master	Refer to the master machine
Slave$_i$	Refer to the ith slave machine
$LB(u)$	Equation 4
$\mathcal{LB}$	Equation 5

Example 9 In Fig. 7, we have 5 shell component partitions SC_1, SC_2, SC_3, SC_4 and SC_5, which are affiliated

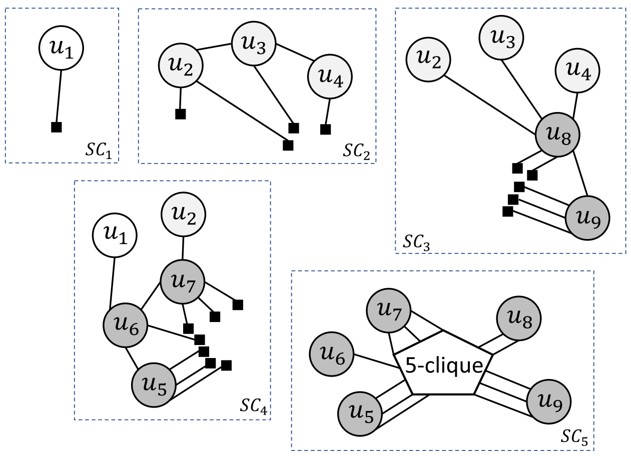


Fig. 7 Shell component partition

to 5 k-shell components S_1^1, S_2^1, S_3^1, S_3^2 and S_4^1 from the example in Fig. 6. For instance, SC_4 is affiliated to S_3^2. $V(S_3^2) = \{u_5, u_6, u_7\}$ and $E(S_3^2) = \{(u_5, u_6), (u_6, u_7)\}$. $SC_4.V = \{u_5, u_6, u_7\}$, $SC_4.V^- = \{u_1, u_2\}$ and $SC_4.E = \{(u_1, u_6), (u_2, u_7), (u_5, u_6), (u_6, u_7)\}$. For the edges between $SC_4.V$ and the 5-clique, we do not store those specific edges in SC_4, but only record $SC_4.h[u_5] = 3$, $SC_4.h[u_6] = 1$ and $SC_4.h[u_7] = 3$.

Algorithm 7 partitions the data graph G based on shell component partitions. Firstly, we need to conduct core decomposition (Line 1) on G so that we can get the coreness of each vertex. We traverse all the vertices with ascending order of coreness (Line 2). Each vertex is marked *unassigned* to shell component partition as default. Each time meeting an *unassigned* vertex u in Algorithm 7 (Line 3), we create a new SC for u, set the related domains of SC and set u as *assigned* (Lines 4–7). Then, we call Algorithm 8 (details following) to recursively collect all the vertices which are supposed to be in SC (Line 8), followed by adding SC to $\mathcal{SP}[u]$. When all the vertices are set *assigned* (in Algorithm 7 or Algorithm 8), we get the complete $\mathcal{SP}$.

In Algorithm 8, for the vertex u, its neighbors $N(u, G)$ are classified into 3 categories. Lines 2–5 include such $v \in N(u, G)$ with $c(v) < c(u)$ to SC. Lines 14–15 add such $v \in N(u, G)$ with $c(v) > c(u)$ to $SC.h[u]$. For such $v \in N(u, G)$ with $c(v) = c(u)$, apart from including v to SC, we also need to recursively call Algorithm 8 for v, because $v \in SC.V$. (Lines 6–13)

After choosing an anchor vertex in each iteration, we use Algorithm 9 to maintain the partition for the next iteration. We prove the maintenance by Algorithm 9 is correct. We firstly set all the vertices as *unassigned* (Lines 1–2). For the anchoring of x, we define $V_x = (\bigcup_{SC \in \mathcal{SP}[x]} SC.V) \backslash \{x\}$. Then, in the post-anchor graph G_x with anchoring x, after updating the coreness of the anchor vertex and its followers (Lines 3–5), any k-shell component S_k^i having $\exists v \in V_x$ s.t.

Algorithm 7: **ShellPartition**(G)

```
Input  : $G$ : the graph
Output : $\mathcal{SP}$ : the shell component partition of $G$
1  CoreDecomp($G$, $\emptyset$);
2  for each $u \in V(G)$ in ascending $c(u)$ order do
3      if $u$ is $unassigned$ then
4          $SC \leftarrow$ an empty shell component partition;
5          $SC.V := SC.V \cup \{u\}$;
6          $SC.h[u] := 0$;
7          $u$ is set $assigned$;
8          ShellConnect($u$, $G$, $SC$);
9          $\mathcal{SP}[u] := \mathcal{SP}[u] \cup \{SC\}$;
10 return $\mathcal{SP}$
```

Algorithm 8: **ShellConnect**(u, G, SC)

```
Input : $u$ : a vertex, $G$ : the graph, $SC$ : the shell component
        partition containing $u$
1  for each $v \in N(u, G)$ do
2      if $c(v) < c(u)$ then
3          $SC.E := SC.E \cup \{(u, v)\}$;
4          $SC.V^- := SC.V^- \cup \{v\}$;
5          $\mathcal{SP}[v] := \mathcal{SP}[v] \cup \{SC\}$;
6      else if $c(v) = c(u)$ then
7          $SC.E := SC.E \cup \{(u, v)\}$;
8          if $v$ is $unassigned$ then
9              $SC.V := SC.V \cup \{v\}$;
10             $SC.h[v] := 0$;
11             $v$ is set $assigned$;
12             ShellConnect($v$, $G$, $SC$);
13             $\mathcal{SP}[v] := \mathcal{SP}[v] \cup \{SC\}$;
14     else if $c(v) > c(u)$ then
15         $SC.h[u]$++;
16     else
```

$v \in V(S_k^i)$ and its affiliated shell component partition are updated by Lines 6–13. The following lemmas and theorems prove the other shell component partitions remain the same.

Lemma 2 *For each non-anchor vertex $u \in V(G) \backslash A$, there is only one shell component partition SC having $u \in SC.V$.*

Proof We prove it by contradiction. Assume $u \in SC.V$ and $u \in SC'.V$. Then the S_k^i that SC is affiliated to and the $S_{k'}^{i'}$ that SC' is affiliated to satisfy (1) $k = k' = c(u)$; and (2) $V(S_k^i)$ and $V(S_{k'}^{i'})$ belong to one connected component since they are both connected to u. Thus, SC and SC' would be one shell component partition, which contradicts our assumption. Proof completes. □

For each $u \in SC.V$, we define $\deg(u, SC)$ as the degree of u in shell component partition SC. $\deg(u, SC) = |\{v \mid (u, v) \in SC.E\}| + SC.h[u]$. With this definition, we can get the coreness for $u \in SC.V$, denoted by $c(u, SC)$.

Algorithm 9: **MaintainSP**(x, G)

```
Input   : $x$ : the anchor vertex, $G$ : the graph
1  for each $u \in V(G)$ do
2      $u$ is set as $unassigned$;
3  $c(x) := +\infty$;
4  for each $u \in \mathcal{F}[x]$ do
5      $c(u)$++;
6  for each $SC \in \mathcal{SP}[x]$ do
7      for each $unassigned$ $u \in SC.V$ with $u \neq x$ do
8          $SC'$ ← an empty shell component partition;
9          $SC'.V := SC'.V \cup \{u\}$;
10         $SC'.h[u] := 0$;
11         $u$ is set $assigned$;
12         ShellConnect($u, G, SC'$);
13         $\mathcal{SP}[u] := \mathcal{SP}[u] \cup \{SC'\}$;
14 $D$ is the set of expired $SC$;
15 for each $assigned$ node $u$ and each $SC \in \mathcal{SP}[u]$ do
16     if $u \in SC.V$ and $SC$ is not new added then
17         $D := D \cup \{SC\}$;
18 for each $SC \in D$ and each $u \in SC.V^- \cup SC.V$ do
19     $\mathcal{SP}[u] := \mathcal{SP}[u] \backslash \{SC\}$;
```

Lemma 3 *For a shell component partition SC and a vertex $u \in SC.V$, the coreness of u in SC is the same as the coreness of u in G, i.e., $c(u, SC) = c(u, G)$.*

Proof Let $\mathcal{O}$ denote a vertex deletion order of core decomposition on G. And $\mathcal{O}$ deletes all the vertices which can be deleted before each vertex in $SC.V$. After the deletion, for each $u \in SC.V$, we denote the degree of u in G as $\deg(u, \mathcal{O})$. When doing core decomposition on the subgraph formed by $SC.V^-$, $SC.V$ and $SC.E$, we can delete all the vertices in $SC.V^-$ before $SC.V$, because their degrees are all 1, and the remaining subgraph is denoted by SC'. Then, for each $u \in SC.V$, $\deg(u, SC') = \deg(u, \mathcal{O})$. Now we can follow the same order to delete the vertices in $SC.V$ as they follow in $\mathcal{O}$. Thus, $c(u, SC) = c(u, G)$ for each $u \in SC.V$. □

Theorem 10 *If a vertex x is anchored in the graph G, we have $\mathcal{F}(x) \subset V_x$.*

Proof Consider a non-anchor vertex $u \notin V_x$. According to Lemma 2, there exists one SC, in which $u \in SC.V$, and $x \notin SC.V \cup SC.V^-$. According to Lemma 3, $c(u, G) = c(u, SC)$. Because $x \notin SC.V \cup SC.V^-$, when x is anchored, $c(u, SC)$ remains the same, so $u \notin \mathcal{F}(x)$. Thus, $\mathcal{F}(x) \subset V_x$. □

Theorem 11 *For a shell component partition SC in G, if for each $u \in SC.V$, there does not exist a $v \in V_x$ and a S_k^i in G_x such that $v \in V(S_k^i) \wedge u \in V(S_k^i)$, SC remains the same in G_x.*

Proof We prove it by contradiction. If SC is different in G_x, it can either be (1) $SC.V^-$ is different, (2) $SC.V$ is different, (3) $SC.E$ is different, or (4) $SC.h$ is different.

For (1), let us assume $SC.V^-$ is different (SC becomes SC' in G_x). S_k^i is the k-shell component that SC' is affiliated to in G_x. This means $\exists u \in SC.V^-$ s.t. $c(u, G_x) > c(u, G) \wedge c(u, G_x) \geq k$. Because $c(u, G) < k$ and u can increase its coreness at most 1 based on Theorem 4, we have $c(u, G_x) = k$, which means $u \in V_x \wedge u \in V(S_k^i)$. This contradicts the condition of the theorem.

For (2), let us assume $SC.V$ is different. This means $\exists u \in SC.V$ s.t. $c(u, G_x) > c(u, G)$. Based on Theorem 10, $u \in V_x$, so for each $v \in SC.V$, $v \in V_x$. This contradicts the condition of the theorem.

For (3), when both $SC.V^-$ and $SC.V$ remain the same in G_x, $SC.E$ must be the same.

For (4), let us assume $SC.h$ is different. Then we have SC' in G_x where $SC'.V^- = SC.V^-$, $SC'.V = SC.V$ and $SC'.E = SC.E$, but $\exists u \in SC'.V$ s.t. $SC'.h[u] > SC.h[u]$ (none vertex's coreness would decrease so $SC'.h[u]$ cannot be less). Because $N(u, G)$ are from $SC.V^-$, $SC.V$ or the neighbor vertices counted in $SC.h[u]$, there must be $v \in N(u, G) \cap (SC.V^- \cup SC.V)$ increasing its coreness. This contradicts our assumption. □

We have proved all the shell component partitions are correctly updated by Algorithm 9; now we clarify that for each non-anchor vertex u, the set of shell component partitions containing u, $\mathcal{SP}[u]$, is updated correctly in Algorithm 9. Lines 6–13 ensure all the new created shell component partitions have been inserted into $\mathcal{SP}$. Theorem 11 proves part of the shell component partitions remain the same, and Lines 14–17 collect all other expired shell component partitions. Then, Lines 18–19 erase them from $\mathcal{SP}$.

Partition Complexity The space complexity of our partition is $\mathcal{O}(2 \cdot m + n)$. The extra storage $\mathcal{O}(n)$ is from $SC.h$ where we do not need to store the specific edges but only record a number $SC.h[u]$ for each u. For time complexity, Algorithms 7, 8 and 9 are all $\mathcal{O}(m)$, because Algorithms 7 and 9 are both dominated by the subcall of Algorithm 8, and in Algorithm 8, each neighbor $v \in N(u, G)$ of each vertex u is accessed once.

6.2 Independency and reuse

In this subsection, we introduce how each parallel unit (a slave machine, a thread of a machine) can independently and concurrently compute the followers in each shell component partition and how our partition strategy makes part of the computed followers reusable in the next iteration.

Theorem 12 *For each vertex $u \in V(G)$, $|\mathcal{F}(u, G)| = \sum_{SC \in \mathcal{SP}[u]} |SF[u][SC]|$.*

Proof Based on Theorem 10, for each $SC \notin \mathcal{SP}[u]$, SC does not have any follower of u. Based on Lemma 2, for each SC and SC' with $SC \in \mathcal{SP}[u] \wedge SC' \in \mathcal{SP}[u]$, $SF[u][SC]$ and

$SF[u][SC']$ do not have any overlap. Based on Lemma 3, $SF[u][SC]$ can be correctly computed within each $SC \in \mathcal{SP}[u]$. Therefore, $|\mathcal{F}(u, G)| = \sum_{SC \in \mathcal{SP}[u]} |SF[u][SC]|$. □

By Theorem 12, we know that each $SF[u][SC]$ can be computed independently and concurrently. In our parallel algorithm, we distribute each shell component partition SC to one or more machines and use $SC.C$ to distribute the anchor candidates, so that for each $u \in V(G)$ and each $SC \in \mathcal{SP}[u]$, $SF[u][SC]$ can be computed in the only machine having $u \in SC.C$. The specific distributing strategy will be explained in Sect. 6.3. Algorithm 10 presents how one machine computes the followers of one shell component partition SC. As each single machine has multiple threads, we can parallelly compute $SF[u][SC]$ for each $u \in SC.C$ (Line 2). The candidates in $SC.C$ are randomly and evenly allocated to each thread of a machine. For the subgraph formed by $SC.V^-$, $SC.V$ and $SC.E$, denoted by G_{SC}, we can get the *shell-layer pair* (Sect. 5.4) of each $u \in SC.V^- \cup SC.V$. Then, we have $N^{\leq}(u, SC)$ to denote the neighbors in $N(u, G_{SC})$ where each neighbor v has $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i \leq \mathcal{P}[u].i$ or $\mathcal{P}[v].k < \mathcal{P}[u].k$. And we have $N^{>}(u, SC)$ to denote the neighbors in $N(u, G_{SC})$ where each neighbor v has $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i > \mathcal{P}[u].i$ or $\mathcal{P}[v].k > \mathcal{P}[u].k$. With $N^{\leq}(u, SC)$ and $N^{>}(u, SC)$ of each $u \in SC.V$ (Line 1), Lines 3–15 can compute the followers. Similar to the *Degree Check* in Sect. 5.4, we develop *Degree Check in Partition*, and Theorem 13 ensures the correctness of Algorithm 10. For computing the followers of each single $u \in SC.C$, the time complexity of Algorithm 10 is the same as Algorithm 4, $\mathcal{O}(m)$. Considering the number of threads N_T and the number of anchor candidates n in the worst case, the time complexity of Algorithm 10 is $\mathcal{O}(\frac{n \cdot m}{N_T})$.

Degree Check in Partition For a vertex $u \in SC.V$, the *partition degree bound* of u in SC is denoted by $d_P^+(u, SC)$. Specifically, $d_P^+(u, SC) = d_s^+(u, SC) + d_u^+(u, SC) + d_>(u, SC)$, in which $d_s^+(u, SC)$ (resp. $d_u^+(u, SC)$) is the number of $survived$ (resp. $unexplored$) neighbors in $\{x\} \cup (N^{\leq}(u, SC) \cap H) \cup N^{>}(u, SC)$, and $d_>(u, SC)$ is equal to $SC.h[u]$. The following theorem indicates that, for a shell component partition SC, we can exclude a candidate follower $u \in SC.V$ if $d_P^+(u, SC) < c(u, G) + 1$. The discard of a vertex may invoke the discard of other vertices in $SC.V$. When the deletion cascade terminates, the tags of all the vertices affected by the discard of u will be correctly updated.

Theorem 13 *A vertex $u \in SC.V$ cannot be a follower if $d_P^+(u, SC) < c(u, G) + 1$.*

Proof According to the definitions of partition degree bound and the degree bound in Sect. 5.4, for a vertex $u \in SC.V$, $d_P^+(u, SC) = d^+(u, G)$. And based on the proof of Theorem 8, this theorem also holds. □

Algorithm 10: FindFollowers(SC)

```
Input  : $SC$ : the shell component partition
Output : $SF[\cdot][SC]$ : the computed followers in $SC$
1  Get $N^{\leq}(u, SC)$ and $N^{>}(u, SC)$ for each $u \in SC.V$;
2  for each $x \in SC.C$ by parallel multi-threads do
3      $H := \emptyset$;
4      $H.push(x)$;
5      while $H \neq \emptyset$ do
6          $u \leftarrow H.pop()$;
7          if $d_P^+(u, SC) \geq c(u, G) + 1$ or $u = x$ then
8              $u$ is set *survived*;
9              for each $v \in N^{>}(u, SC)$ and $v \notin H$ do
10                 $H.push(v)$;
11         else
12             $u$ is set *discarded* ;
13             Shrink($u$) (Algorithm 5);
14     for each *survived* vertex $v$ except for $x$ do
15         $SF[x][SC] := SF[x][SC] \cup \{v\}$;
16 return $SF[\cdot][SC]$
```

Followers Reuse In Algorithm 9, except those new added shell component partitions, other shell component partitions in $\mathcal{SP}$ remain the same. For these unchanged ones, due to the independency of followers computation across shell component partitions, the computed followers in last iteration can be reused in the next iteration.

6.3 Computing resource scheduling

In order to evenly utilize our computing resource (multiple machines and multithreads) and limit the communication cost, we propose a scheduling strategy in this subsection. Firstly, we propose a reasonable estimation of the computational amount of finding the followers of a vertex in a shell component partition. We adapt the upper bound of followers in Sect. 5.5 to *upper bound in partition* and then explain that it is a reasonable estimation of computational amount.

Upper Bound in Partition. For a candidate anchor vertex $x \in SC.V^- \cup SC.V$, the number of followers of x in SC, $SF[x][SC]$, has an upper bound, denoted by $UB_{SC}(x)$. We have $UB_{SC}(x) = \sum_{u \in N^{>}(x,SC)} (UB_{SC}(u) + 1)$ if $|N^{>}(x, SC)| > 0$ and $UB_{SC}(x) = 0$ if $|N^{>}(x, SC)| = 0$. We can accumulatively compute the upper bound in partition as the way of Sect. 5.5. Based on Theorem 9, it is easy to prove that $SF[x][SC] \leq UB_{SC}(x)$.

Distributing Strategy When computing $SF[u][SC]$ of $u \in SC.C$, the computational amount is dominated by the heap traverse during Lines 6–13 of Algorithm 10. And the number of vertices that can be added into the heap H is correlated to $UB_{SC}(u)$, so we adopt $UB_{SC}(u)$ as the estimated computational amount of $SF[u][SC]$. Along with the followers reuse strategy mentioned in the last sec-

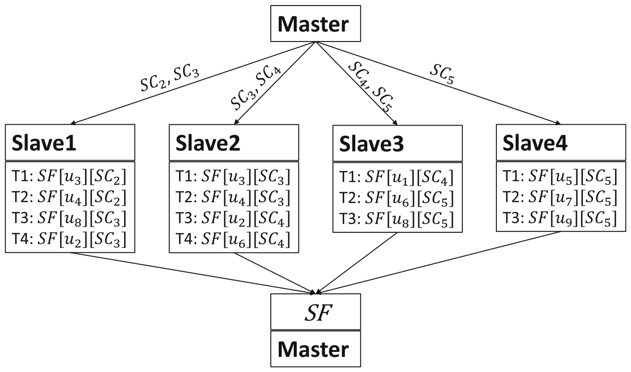


Fig. 8 Computing schedule

tion, in each iteration, we only distribute such computing task of $SF[u][SC]$ which cannot be reused. We firstly get the estimated average computational amount $C_{avg} = (\sum_{u\in V(G)}\sum_{SC\in\mathcal{SP}[u]} UB_{SC}(u))/N_S$, where N_s is the number of *Slave* machines. With a sequence $\mathcal{S}$ of all the computing tasks, in which $\mathcal{S}(i) = SF[u_i][SC_i]$, and the computational amount C_j (initialized as 0) of each machine $Slave_j$, we distribute the tasks as following. Starting from $Slave_1$ and $\mathcal{S}(1)$, if $UB_{SC_1}(u_1) > 0, C_1 = C_1 + UB_{SC_1}(u_1)$, we add u_1 to $SC_1.C$ and send SC_1 to $Slave_1$, so that $\mathcal{S}(1)$ is distributed to $Slave_1$, until the jth task $\mathcal{S}(j)$ is distributed to $Slave_1$ satisfying $\sum_{i=1}^{j} UB_{SC_i}(u_i) \le C_{avg}$ and $\sum_{i=1}^{j+1} UB_{SC_i}(u_i) > C_{avg}$. Then we can start distributing tasks for the next machine. For machine $Slave_j$ with $j \in [1, N_s - 1]$, we distribute tasks as the above way, and the last machine $Slave_{N_s}$ have all the remaining tasks. Note that it is easy to avoid sending one shell component partition multiple times to one machine.

Example 10 Figure 8 gives a scheduling example in the 5-machine distributed environment and the example is about the graph G in Fig. 6. Firstly, the *Master* machine decomposes G into the five shell component partitions as in Fig. 7. Then, we compute all the upper bounds in partition. In SC_1, $UB_{SC_1}(u_1) = 0$. In SC_2, $UB_{SC_2}(u_2) = 0$, $UB_{SC_2}(u_3) = 1$ and $UB_{SC_2}(u_4) = 2$. In SC_3, $UB_{SC_3}(u_9) = 0$, $UB_{SC_3}(u_8) = 1$ and $UB_{SC_3}(u_2) = UB_{SC_3}(u_3) = UB_{SC_3}(u_4) = 2$. In SC_4, $UB_{SC_4}(u_5) = UB_{SC_4}(u_7) = 0$, $UB_{SC_4}(u_6) = 2$, $UB_{SC_4}(u_2) = 1$ and $UB_{SC_4}(u_1) = 3$. In SC_5, for the vertices in the 5-clique, the upper bounds in partition are 0. $UB_{SC_5}(u_6) = 1$, $UB_{SC_5}(u_8) = 2$, and $UB_{SC_5}(u_5) = UB_{SC_5}(u_7) = UB_{SC_5}(u_9) = 3$. After using our distributing strategy, the shell component partitions and the corresponding candidate anchors are distributed to 4 *Slave* machines. When each machine has 4 threads, it can parallelly compute the followers of candidate anchors within one machine. For instance, in *Slave*1, thread 1 (T1) computes $SF[u_3][SC_2]$, thread 2 (T2) computes $SF[u_4][SC_2]$, thread 3 (T3) computes $SF[u_8][SC_3]$ and thread 4 (T4) computes $SF[u_2][SC_3]$. Each *Slave* sends the followers to the *Master*, and *Master* collects them then has the complete SF.

6.4 The DGAC algorithm

In this section, we firstly introduce our *lower bound based pruning* technique to further accelerate our algorithm, then we combine all the elements and introduce the distributed algorithm DGAC. In each iteration of the greedy strategy, the time cost of DGAC is dominated by the parts of Sects. 6.1–6.3. Thus, for an anchoring budget b, the time complexity of DGAC is $O(\frac{b\cdot n\cdot m}{N_S\cdot N_T})$.

Lower Bound Based Pruning In each iteration, we compute a lower bound of followers $LB(u)$ of each non-anchor vertex $u \in V(G)\backslash A$ as Equation 4. Note that, for a shell component partition $SC \in \mathcal{SP}[u]$, if $SF[u][SC]$ has not been computed in last iteration or cannot be reused, we let $SF[u][SC] = \emptyset$. Then, we can compute a lower bound of followers of this iteration $\mathcal{LB}$ as Eq. 5. Theorem 14 shows how the lower bound manages to prune some anchor candidates in each iteration.

$$LB(u) = \sum_{SC\in\mathcal{SP}[u]\wedge SF[u][SC]\neq\emptyset} |SF[u][SC]| \tag{4}$$

$$\mathcal{LB} = \max\{LB(u) \mid u \in V(G)\backslash A\} \tag{5}$$

Theorem 14 *Given $\mathcal{LB}$ in an iteration, for a non-anchor vertex $u \in V(G)\backslash A$, $UB_\sigma(u) = \sum_{SC\in\mathcal{SP}[u]}(UB_{SC}(u))$. If $UB_\sigma(u) < \mathcal{LB}$, u is not the best anchor of the iteration.*

Proof According to Eq. 5, there exists a u' with $LB(u') = \mathcal{LB}$. We have $|\mathcal{F}(u')| \ge LB(u')$ from Eq. 4 and have $|\mathcal{F}(u)| \le UB_\sigma(u)$ from the definition of $UB_\sigma(u)$. Given $UB_\sigma(u) < \mathcal{LB}$, we can conclude $|\mathcal{F}(u)| < |\mathcal{F}(u')|$, so that u does not have the maximum followers and cannot be the best anchor. □

Based on the lower bound-based pruning, Algorithm 11 is called after Algorithm 7 finishes the graph partitioning or Algorithm 9 maintains the partition, and before the *Master* machine distributes specific computing tasks. Firstly, we only compute the upper bound in partition of the vertices in the new added shell component partitions (Line 1). In Lines 3–10, we compute the upper bound $UB_\sigma(u)$ of the total number of followers for each $u \in V(G)\backslash A$. Note that, for the reusable part of computed followers, we use them for a tighter bound (Line 7). By only adding the reusable followers for a vertex u, we can get the lower bound $LB(u)$ (Line 8). The $\mathcal{LB}$ (Line 2, Lines 11–12) is to record the maximum $LB(u)$ among $u \in V(G)\backslash A$. By $\mathcal{LB}$, we can prune such vertex u with $UB_\sigma(u) \le \mathcal{LB}$ (Lines 13–16). Only such

Algorithm 11: **PruneCandidates**(G)

```
Input  : $G$ : a social network with all the above definitions as
         global variables
1 Compute $UB_{SC}(\cdot)$ only for new added $SC$;
2 $\mathcal{LB} := 0$;
3 for each $u \in V(G)\backslash A$ do
4     $UB_\sigma(u) := 0$; $LB(u) := 0$;
5     for each $SC \in \mathcal{SP}[u]$ do
6         if $SF[u][SC] \neq \emptyset$ then
7             $UB_\sigma(u) := UB_\sigma(u) + |SF[u][SC]|$;
8             $LB(u) := LB(u) + |SF[u][SC]|$;
9         else
10            $UB_\sigma(u) := UB_\sigma(u) + UB_{SC}(u)$;
11    if $LB(u) > \mathcal{LB}$ then
12        $\mathcal{LB} := LB(u)$;
13 for each $u \in V(G)\backslash A$ do
14    if $UB_\sigma(u) > \mathcal{LB}$ then
15        for each $SC \in \mathcal{SP}[u]$ with $SF[u][SC] = \emptyset$ do
16            $SC.C := SC.C \cup \{u\}$;
```

Algorithm 12: **DecideAnchor**(G)

```
Input  : $G$ : a social network with all the above definitions as
         global variables
Output : $a$ : the anchor vertex of current iteration
1 $\lambda := -1$; $a := null$;
2 for each $u \in V(G)\backslash A$ do
3     if $\exists SC \in \mathcal{SP}[u]$ with $SF[u][SC] = \emptyset$ then
4         Continue;
5     else
6         $\mathcal{F}(u, G) := \bigcup_{SC \in \mathcal{SP}[u]} SF[u][SC]$;
7         if $|\mathcal{F}(u, G)| > \lambda$ then
8             $a := u$; $\lambda := |\mathcal{F}(u, G)|$;
9 return $a$
```

vertex u with $UB_\sigma(u) > \mathcal{LB}$ (Line 14) needs to be added into $SC.C$ for $SC \in \mathcal{SP}[u]$ where $SF[u][SC]$ has not been computed in last iteration (Line 15). The *Master* empties $SC.C$ for each shell component partition SC in $\mathcal{SP}$ before each iteration starts, and only considers the updated $SC.C$ when distributing the computing tasks.

Algorithm 12 is called after the *Master* machine collects the computed followers result. Note that, for a vertex $u \in V(G)\backslash A$, if $\exists SC \in \mathcal{SP}[u]$ with $SF[u][SC] = \emptyset$ (Line 3), that means u is pruned by Algorithm 11, so does not need to be considered as the anchor in the current iteration. We traverse all the non-anchor vertices (Line 2) and keep updating the best anchor so far (Lines 6–8). Finally, we get the best anchor in this iteration (Line 9).

Combining all the elements, the abstract pseudo-code of DGAC is summarized as Algorithm 13.

Algorithm 13: **DGAC**(G, b)

```
Input  : $G$ : a social network, $b$ : number of anchors
Output : $A$ : the set of anchor vertices
1 $\mathcal{SP}$ := ShellPartition($G$) [Master calling];
2 for $iteration$ from 1 to $b$ do
3     PruneCandidates($G$) [Master calling];
4     Use scheduling strategy (Sect. 6.3) [Master calling];
5     for each $Slave_i$ with $i \in [1, N_s]$ in parallel do
6         for each received $SC_i^j$ of $Slave_i$ do
7             $SF[\cdot][SC_i^j]$ := FindFollowers($SC_i^j$);
8             Send $SF[\cdot][SC_i^j]$ to Master;
9     Collect all the computed followers [Master calling];
10    $a$ := DecideAnchor($G$) [Master calling];
11    $A := A \cup \{a\}$; $deg(a, G) := +\infty$;
12    MaintainSP($a, G$) [Master calling];
13 return $A$
```

DGAC for Single Machine The algorithm DGAC can be adapted to a parallel algorithm in a single machine with multiple threads. The main changes are explained as follows:

(1) We regard each available thread as one independent slave machine when using our distributing strategy.
(2) Instead of transferring data among machines, in one machine, we only need to assign different shell component partitions to each thread.
(3) Each thread independently calls Algorithm 10 to conduct computation. When it comes to Line 2, each thread simply serializes the computation of followers for the anchor candidates in its $SC.C$.

7 Experimental evaluation

Datasets. We use eight real-life datasets in our experiments. Brightkite, Gowalla, Youtube and Livejournal are from http://snap.stanford.edu/. Arxiv, NotreDame, Stanford and DBLP are from http://konect.uni-koblenz.de/. Due to the Space limitation, we abbreviate the dataset names as their capital first letters when necessary. Table 6 shows the statistics of the datasets, listed in increasing order of edge numbers.

Parameters All the programs are implemented in C++ and compiled with G++ on Linux. The experiments are conducted on a cluster with 9 machines having 1 Gbps network. They all have 3.4 GHz Intel Xeon CPU with 4 cores (8 threads available) and Redhat system. We use MPICH [1] to transfer data among the machines and use OpenMP [2] to utilize the multithreads within one machine.

Algorithms Toward effectiveness, we mainly compare 6 algorithms with our GAC algorithm, including 4 heuristics, the exact solution, and the algorithm for anchored k-core prob-

Table 6 Statistics of datasets

Dataset	Nodes	Edges	d_{avg}	$d_{\max}$	$k_{\max}$
`B.`	58,228	194,090	6.7	1098	52
`A.`	34,546	421,578	24.4	846	30
`G.`	196,591	456,830	9.2	10,721	51
`N.`	325,729	1,497,134	6.5	3812	155
`S.`	281,903	2,312,497	16.4	38,626	71
`Y.`	1,134,890	2,987,624	5.3	28,754	51
`D.`	1,566,919	6,461,300	8.3	2023	118
`L.`	3,997,962	34,681,189	17.4	14,815	360

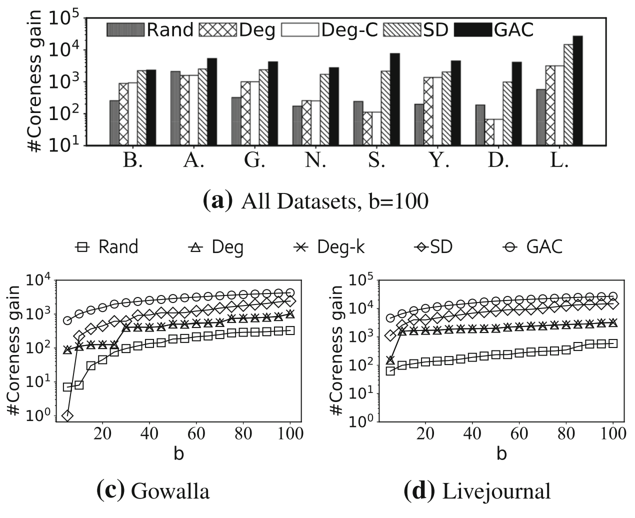
Table 7 Summary of algorithms

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 $\deg_{\succ}(\cdot)$ for every $u \in V(G)$, where $\deg_{\succ}(u) = \lvert\{v \mid v \in N(u, G)\ \&\ \mathcal{P}(v) \succ \mathcal{P}(u)\}\rvert$
`OLAK`	The state-of-the-art algorithm for anchored k-core problem [65]
`GAC`	Algorithm 6
`DGAC`	Algorithm 13
`DGAC(-C)`	`DGAC` without the time of data communication between machines.
`GAC-U`	`GAC` without upper bound pruning (Sect. 5.5)
`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

lem. Toward the efficiency of our serial algorithm `GAC`, we incrementally equip the baseline with our proposed techniques to evaluate the performance. Toward the efficiency of our distributed algorithm `DGAC`, we compare its time cost (1 master + 8 slaves each with 8 threads as default) to `GAC` and then vary the number of slave machines and the number of threads to test its scalability. We conducted experiments by varying the budget b from 1 to 100 where the default value is 100. Table 7 lists all the evaluated algorithms.

7.1 Effectiveness

Comparison with Other Heuristics In Fig. 9, we compare the coreness gain from `GAC` with other heuristics (`Rand`, `Deg`, `Deg-C`, and `SD`). The motivation of the basic methods



(a) All Datasets, b=100

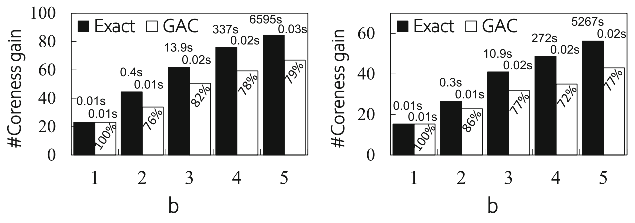
(c) Gowalla **(d)** Livejournal

Fig. 9 Coreness gain from different heuristics

is as follows. For `Deg`, a vertex with larger degree means the anchoring of it can potentially influence more vertices through its large number of neighbor vertices. For `Deg-C`, according to Theorem 7, anchoring a vertex u can only increase the coreness of vertices with coreness higher than $c(u)$. Thus, anchoring a vertex with relatively higher degree and lower coreness may be more effective. For `SD`, the **successive degree** of a vertex u is defined as $\deg_{\succ}(u) = \lvert\{v \mid v \in N(u, G)\ \&\ \mathcal{P}(v) \succ \mathcal{P}(u)\}\rvert$ where $\mathcal{P}(u) = (k, i)$ means u is in the ith layer of the k-shell. According to Theorem 7, for a vertex u, only the vertices in $N(u, G)$ that contribute to $\deg_{\succ}(u)$ is reached by u via upstair paths. Thus, anchoring a vertex with large successive degree may be effective. Table 7 shows the details.

As shown in Fig. 9a, the performance of `Rand` is the worst as it chooses random vertices to anchor. The performance of `Deg` and `Deg-C` is 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 7). Compared with the above heuristics, `GAC` achieves the much larger coreness gains on all the datasets. The effect of varying b is shown in Fig. 9c, d. 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 neighbors, 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 runtimes are also reported. Figure 10 shows that the coreness gain of `GAC` is always at least 70% of `Exact`, and `GAC`

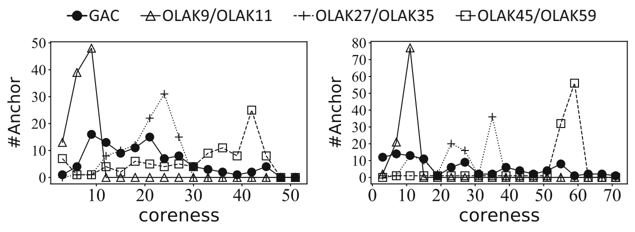


(a) Brightkite **(b)** Arxiv

Fig. 10 GAC versus Exact

Table 8 Characteristics of anchor set

Dataset	$\mathrm{Deg}_{\mathrm{avg}}$	$\mathrm{Deg}_{\mathrm{anc}}$	p_{Deg}	p_{CN}	p_{SD}
B.	7.35	37.76	0.884	0.891	0.893
A.	24.37	29.71	0.670	0.663	0.678
G.	9.67	43.86	0.904	0.919	0.919
N.	6.69	11.28	0.808	0.828	0.846
S.	14.14	56.09	0.745	0.763	0.788
Y.	5.27	81.85	0.985	0.982	0.982
D.	8.08	27.85	0.905	0.896	0.911
L.	17.35	145.74	0.935	0.940	0.943



(b) Gowalla **(c)** Stanford

Fig. 11 Distribution of anchors on coreness

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 8 shows the average degree of anchors ($\mathrm{Deg}_{\mathrm{anc}}$) from GAC is much larger than the average degree of all the vertices in the graph ($\mathrm{Deg}_{\mathrm{avg}}$). 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 7, a vertex with larger successive degree has more potential followers. For each anchor $x \in A$, we get its ranking in all the vertices, denoted by $\mathcal{O}^x_{\mathrm{Deg}}$, $\mathcal{O}^x_{\mathrm{CN}}$ and $\mathcal{O}^x_{\mathrm{SD}}$, in ascending order of degree, coreness and successive degree, respectively. Then, $p_{Deg} = \frac{\sum_{x \in A} \mathcal{O}^x_{\mathrm{Deg}}}{|A||V(G)|}$, $p_{\mathrm{CN}} = \frac{\sum_{x \in A} \mathcal{O}^x_{\mathrm{CN}}}{|A||V(G)|}$ and $p_{\mathrm{SD}} = \frac{\sum_{x \in A} \mathcal{O}^x_{\mathrm{SD}}}{|A||V(G)|}$. Table 8 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 Fig. 9. Moreover, Fig. 11 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.

Table 9 Statistics of top-b solutions

Dataset	Gain_{UB}	Gain_{DG}	Gain_{RD}	J^{UB}_{DG}	J^{UB}_{RD}
B.	2357	**2598**	2488	0.538	0.538
A.	5426	5391	**5503**	0.739	0.681
G.	**4260**	4259	4258	0.754	0.887
N.	2798	**2803**	**2803**	0.653	0.681
S.	**7748**	7695	7727	0.695	0.739
Y.	**4571**	4525	3782	0.361	0.370
D.	4159	4166	**4396**	0.802	0.695
L.	27067	**27113**	27072	0.869	0.887

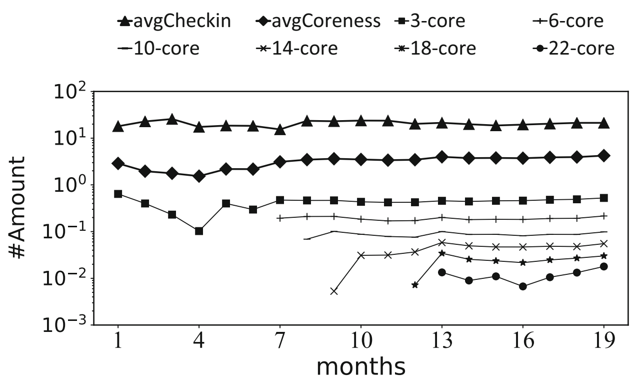


Fig. 12 #Checkin, coreness and k-core size

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 (Sect. 5.5). 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 9, the coreness gains of different solutions (anchor sets) are very similar, where the values are denoted by $\mathrm{Gain}_{\mathrm{UB}}$, $\mathrm{Gain}_{\mathrm{DG}}$ and $\mathrm{Gain}_{\mathrm{RD}}$ accordingly, and the largest value for each dataset is marked in bold. Moreover, as shown in Table 9, 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 the 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 ith network is the induced subgraph by the users with at least 1 check-in during the $(i+1)$th month, except for the first and the last months where the data are 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 Fig. 12, the pattern of size proportions of k-cores is 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 Fig. 14. The pattern of average coreness over the first 7 months in Fig. 12 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) is more reasonable than using the size of k-core (anchored k-core).

Comparison with OLAK For each dataset, we run OLAK with every possible input of k and record every time cost. In Fig. 13, we use OLAK(avg) to denote the average time of all the inputs k for each dataset, and use OLAK(all) to denote the total time of all the inputs k. We compare OLAK(avg) and OLAK(all) with GAC and DGAC. Note that the anchor budget $b = 100$ for all the above algorithms. In Fig. 13, we can see OLAK(avg) is always the fastest; this is because for one single input k, the number of anchor candidates is $\mathcal{O}(k_{\max})$ less than that of GAC. Even further, for each anchor candidate, the search space of followers in OLAK is also $\mathcal{O}(k_{\max})$ less than that of GAC. Therefore, GAC is theoretically $\mathcal{O}(k_{\max}^2)$ (resp. $\mathcal{O}(k_{\max})$) slower than OLAK(avg) (resp. OLAK(all)). Considering the value of $k_{\max}$ of each dataset in Table 6, our algorithm GAC (resp. DGAC) runs efficiently, faster than OLAK(all) and around two orders of magnitude (resp. one order of magnitude) slower than OLAK(avg).

Table 10 is added to show that the largest coreness gain (denoted by $\max_{\text{OLAK}}$) that OLAK can achieve only reaches 46–77% of the coreness gain by GAC, on all the datasets. For the anchor set A_k computed by OLAK with each possible input k, we compute the total sum of coreness gain from all the vertices and all the coreness value with the anchoring of A_k. Then, we can compute the largest and average coreness gain (denoted by $\max_{\text{OLAK}}$ and avg_{OLAK}) for different k values on each dataset. Table 10 also shows that avg_{OLAK} is only 4–41% of the coreness gain of GAC. Besides, Fig. 14 shows the best k for OLAK is rather different for different datasets. There is no uniform preference on large, moderate, or small k values for different datasets.

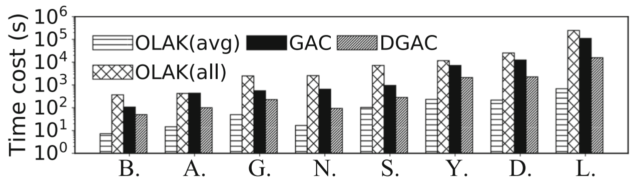


Fig. 13 Time cost, OLAK, GAC and DGAC

Figure 11 shows the distribution of 100 anchors from GAC and OLAK on coreness value. Figure 15 shows the distribution of followers from GAC and OLAK on coreness value. For each coreness value x_i on the horizontal ordinate of Fig. 11 (resp. Fig. 15), its value on the vertical ordinate is the number of anchors (resp. followers) with original coreness value within $(x_{i-1}, x_i]$. We can see that the distribution of anchors from GAC 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 [65]. Besides, Fig. 15 shows the distribution of followers, which has the similar result as the distribution of anchors. We then explore the overlap of anchoring results of GAC and OLAK by computing the Jaccard Index between their anchor sets, which is shown in Table 11. Specifically, for each dataset, GAC has the only anchor set A_{GAC}. For each k value input to OLAK, we have the anchor set A^k_{OLAK}, so we can compute the Jaccard Index $J^k = \frac{|A_{GAC} \cap A^k_{\text{OLAK}}|}{|A_{GAC} \cup A^k_{\text{OLAK}}|}$. Then, we compute $\text{avg}_{\text{Jaccard}}$ (resp. $\max_{\text{Jaccard}}$) which is the average (resp. maximum) value of J^k of all the inputs k. From Table 11, we can find the $\text{avg}_{\text{Jaccard}}$ and $\max_{\text{Jaccard}}$ of all the datasets are quite small, which means the anchor set of OLAK and GAC have little overlap.

Finally, we present a case study on DBLP dataset which is shown in Fig. 16. We choose the anchor budget $b = 5$ for GAC and OLAK (with inputs $k = 10, 20, 30$). As the number of vertices and edges are huge for DBLP, we only visualize the anchors and followers and the edges induced by them. Note that the followers of OLAK are the ones improving their coreness from any original coreness value. All the vertices are drawn within a number of concentric circles. The followers are drawn by grey-color nodes, and the ones having the same coreness value before the anchoring are put in the same circle. The anchors are drawn by bold black-color nodes. Note that, when we input $k = 30$ for OLAK, the 5th anchor has no follower, so there are 4 anchors in Fig. 16d. We can find the followers of GAC are much more than OLAK no matter which inputs. Also, there are 16, 6, 4 and 2 circles in (a), (b), (c) and (d), respectively, in Fig. 16, which means the followers of GAC are more diverse and this is consistent with the result of Fig. 15.

Table 10 Coreness gain, OLAK versus GAC

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 11 Overlap of followers set, OLAK versus GAC

Dataset	B.	A.	G.	N.	S.	Y.	D.	L.
$\text{avg}_{\text{Jaccard}}$	0.021	0.006	0.004	0.002	0.019	0.005	0.001	0.004
$\text{max}_{\text{Jaccard}}$	0.058	0.02	0.02	0.053	0.064	0.02	0.02	0.031

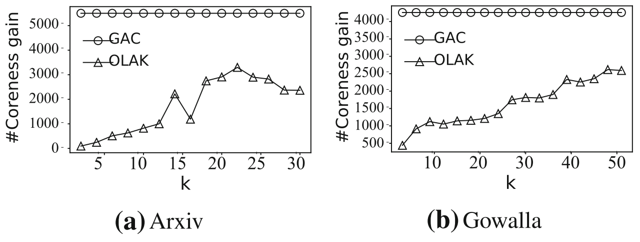


Fig. 14 Coreness gain on different inputs of k

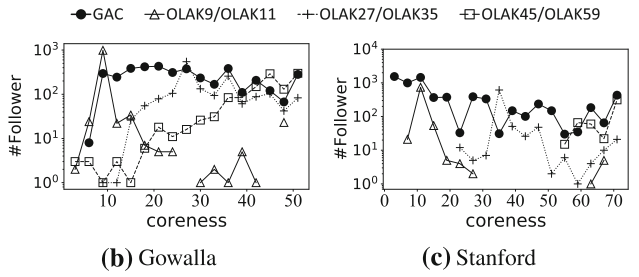


Fig. 15 Distribution of followers on coreness

Comparison with Variations of OLAK We reasonably adjust OLAK aiming to globally reinforce social networks, i.e., to maximize the coreness gain from all the vertices but not limited to the vertices in $(k-1)$-shell as in the original OLAK with a fixed input k. We develop two variation algorithms OLAK-v1 and OLAK-v2. For both of them, on each dataset, we firstly input every possible k value to the original OLAK with $b = 100$, to get the anchor set A_k for each $k \in [2, k_{\max}]$. The union of all such A_k is the candidate anchors pool A_p of OLAK-v1 and OLAK-v2. Then OLAK-v1 works as follows: (1) Compute the coreness gain $g(u)$ for each $u \in A_p$; (2) In each of the 100 iterations, we choose a non-anchor vertex in A_p that has the largest coreness gain on G as the new anchor. OLAK-v2 works as follows: In each of the 100 iterations, we choose a random $k \in [2, k_{\max}]$ in A_k, and we update the coreness gain of each non-anchor vertex (using the reuse mechanism of GAC). Then, we choose the vertex with the largest coreness gain on current graph (with existing anchors) as the new anchor. Figure 17a shows the coreness gain of OLAK-v1 and OLAK-v2 comparing with GAC. We find that, the coreness gain of GAC is always larger than OLAK-v1 and OLAK-v2. In larger datasets, the coreness gain gap is even larger. Figure 17b shows the time cost of OLAK-v1 and OLAK-v2 comparing with GAC and DGAC, we find either OLAK-v1 or OLAK-v2 is always slower than both GAC and DGAC.

7.2 Efficiency of GAC

Overall Performance Figure 18a shows the total running time of GAC, GAC-U and GAC-U-R on all the 8 datasets

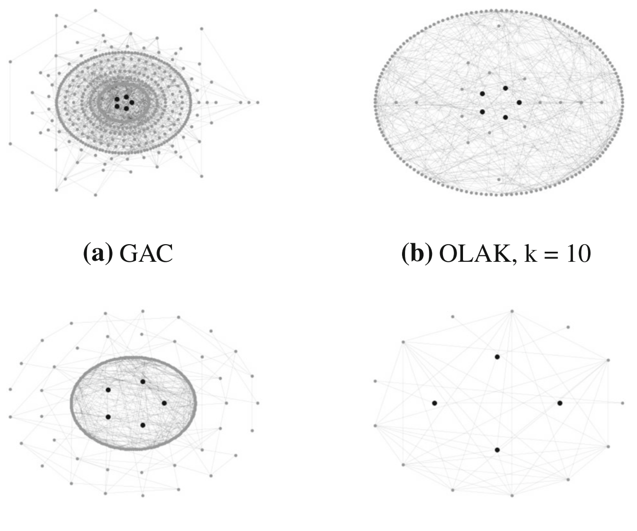
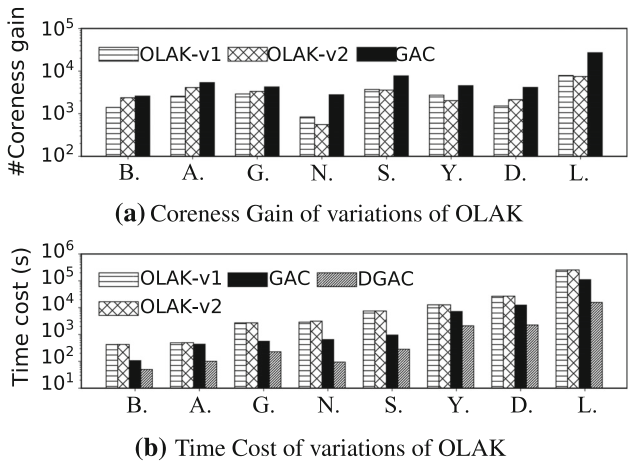


Fig. 16 Case study on DBLP, $b = 5$

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 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 (Sect. 5.5), the runtime of GAC is usually faster than GAC-U by more than 3 times. The details are as follows.

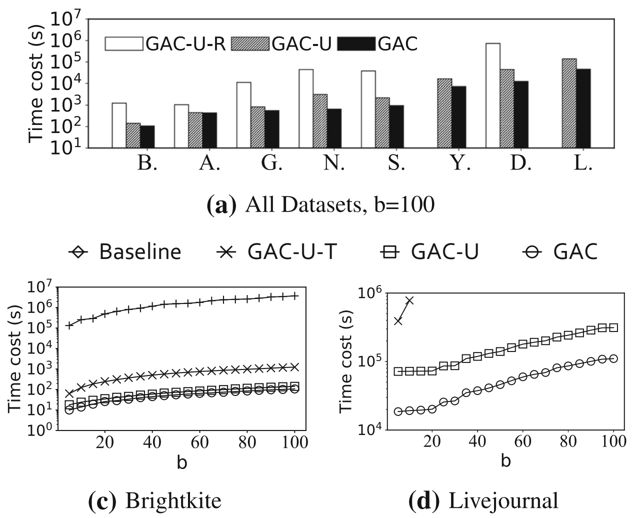
Efficient Followers Computing Equipped with Algorithm 4, the efficient followers computing of the anchors, GAC-U-R is faster than Baseline by at least 1 order of magnitude



(a) Coreness Gain of variations of OLAK

(b) Time Cost of variations of OLAK

Fig. 17 Variations of OLAK versus GAC and DGAC



(a) All Datasets, b=100

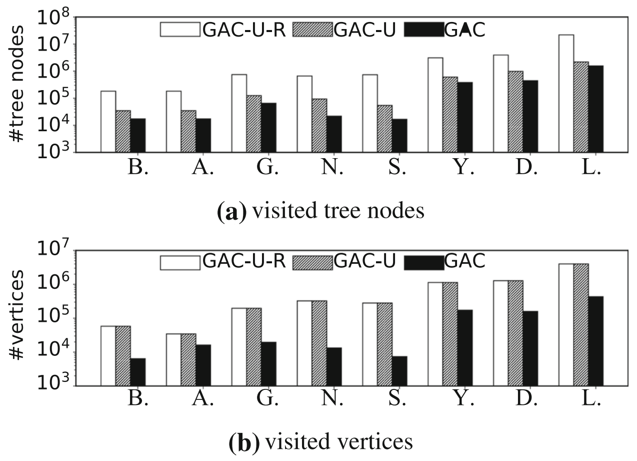
(c) Brightkite

(d) Livejournal

Fig. 18 Time cost of different algorithms

on Brightkite, as shown in Fig. 18c. 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 (Sect. 5.1) 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 Fig. 18. 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 keep the same for one anchoring. In Fig. 19a, the number of visited tree nodes of GAC-U is around 10% of GAC-U-R.



(a) visited tree nodes

(b) visited vertices

Fig. 19 Visited amount

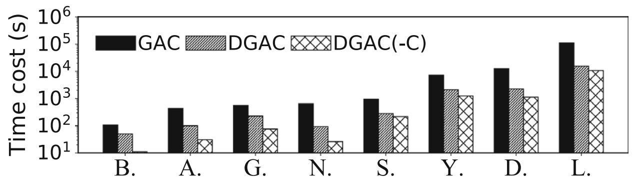
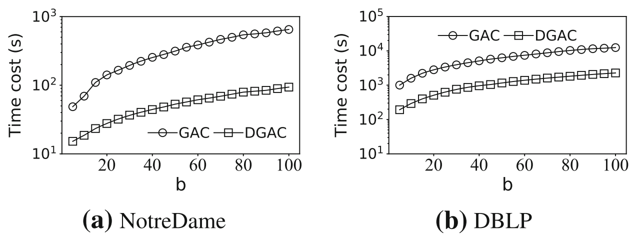


Fig. 20 Time cost, GAC, DGAC and ideal DGAC



(a) NotreDame

(b) DBLP

Fig. 21 Time cost varying *b*, GAC versus DGAC

Candidate Anchors Pruning In Fig. 18, we can see our final algorithm GAC achieves further speedup based on GAC-U when the upper bound pruning is equipped (Sect. 5.5). 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 Fig. 19a, b, the number of visited tree nodes and the number of visited vertices in GAC are much less than that in GAC-U.

7.3 Efficiency of DGAC

We compare the efficiency of DGAC to GAC on all the 8 datasets. In this section, DGAC is conducted on 9 machines (1 master + 8 slaves). As our cluster only has 1Gbps network, the data communications between the master machine and slave machines occupy a large proportion of running time. In Fig. 20, we use DGAC(-C) to denote the total running time minus communication time of DGAC, to also show the

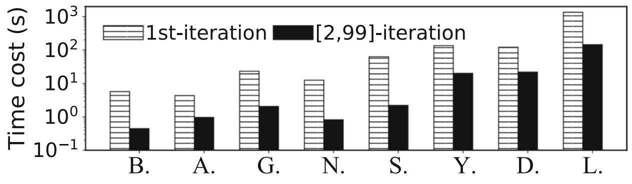


Fig. 22 Time cost, first iteration versus average of [2, 99]-iteration

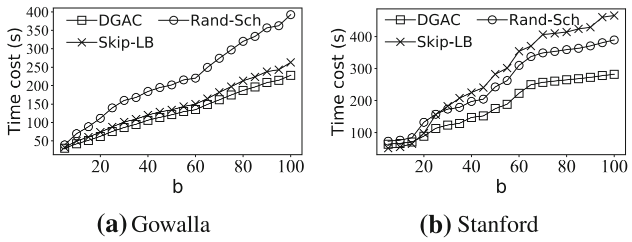


Fig. 23 Time cost of DGAC (skipping individual components)

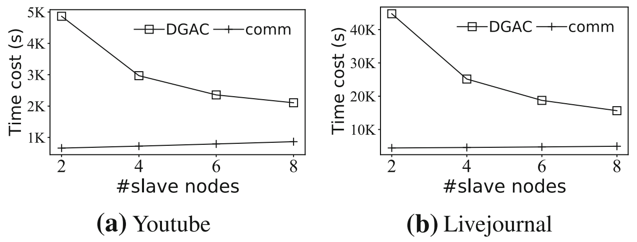


Fig. 24 Time cost, varying the number of machines (8 threads)

ideal algorithm execution time. We can see that, with 8 slave machines having 8 available threads each, DGAC is faster than GAC on all the datasets. On larger datasets such as DBLP and Livejournal, the gap of time cost is more significant, i.e., DGAC is nearly one order of magnitude faster than GAC. For smaller datasets such as Brightkite and Arxiv, we find the data communication time takes more than half of the running time of DGAC, in which the time of DGAC(-C) reflects the successful parallelization of DGAC. Figure 21a, b shows the details of running time of GAC and DGAC with *b* from 1 to 100. We can see that the running time ratio of GAC and DGAC keeps stable with different *b* values, which means DGAC keeps having the advantage of parallelization with different input of anchor budget *b*.

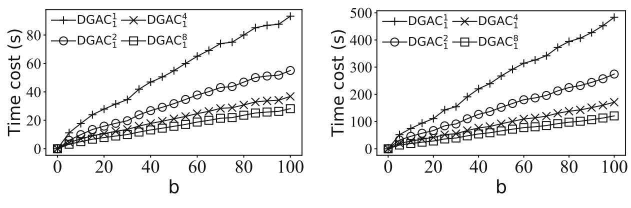
We find that the time cost gaps of GAC and DGAC are different on different datasets. In our resource scheduling strategy, an estimation of computation cost of anchoring a vertex is used to assign the vertices for computation to each slave machine. The accuracy of cost estimation is different on different datasets, while our proposed upper bound for estimation is much more effective than other methods, e.g., degree, coreness and partition size, in our preliminary experiments. As different anchor candidates of one shell component partition can be assigned to different slave machines, the partition is relevant to both the cost estimation and the data communication cost, while it is not the deterministic factor. The time cost from each machine is more relevant to the assigned vertex set for computation.

Validation of Individual techniques of DGAC We validate three techniques of DGAC individually which are reuse mechanism (Sect. 6.2), computing resource scheduling (Sect. 6.3) and lower bound based pruning (Sect. 6.4). Running DGAC without equipping the reuse mechanism is cost-prohibitive especially for large *b*. To show the effectiveness of the reuse mechanism, we report the time cost of the first iteration and the average time cost of 2–99 iterations in the greedy algorithm. The results are shown in Fig. 22. We can find the time cost of the first iteration is around one order of magnitude more than the average time cost of 2–99 iterations. This is because DGAC does not have any computed followers result to reuse at the first iteration, and our reuse mechanism can decide a large part of computed followers to reuse in later iterations. DGAC skipping the lower bound pruning is denoted by Skip-LB in Fig. 23. And we run DGAC using a random resource scheduling strategy (each shell component partition is randomly sent to a slave machine which computes the followers of all the candidate vertices in this partition), which is shown by Rand-Sch in Fig. 23. We can find that DGAC is always the fastest. Skip-LB or Rand-Sch has close performance to DGAC in some cases, while it fails on other cases. Thus, both our computing resource scheduling strategy and lower bound pruning are effective for accelerating DGAC.

7.4 Scalability of DGAC

Varying the Number of Machines We show the scalability of DGAC, varying the number of slave machines. In Fig. 24, we use comm to separately show the time of data communication with different number of slave machines. The time cost of DGAC is roughly inversely proportional to the number of slaves machines (2, 4, 6 and 8) on most of datasets. This is because we use the upper bound of followers (Sect. 6.3) as the estimation of computational amount of finding followers. The time cost of data communication (comm in Fig. 24a, b) slightly increases with the number of slave machines increasing. This is because the total data amount that needs to be transferred is close with different number of machines, but more machines involved cause a slightly more cost of data dividing and scheduling.

Varying the Number of Threads We also vary the number of threads within one single machine and test the cost of our algorithm in Fig. 25. As Sect. 6.4 illustrates, our DGAC algorithm is easily adapted to a parallel algorithm on one machine with multithreads, in which we simply treat each thread as one slave machine having only one thread in DGAC to share the followers computing tasks. DGAC_1^1, DGAC_1^2, DGAC_1^4 and



(a) Brightkite **(b)** Arxiv

Fig. 25 Time cost, varying the number of threads (1 machine)

$\mathtt{DGAC}_1^8$ are conducted on 1 single machine with 1, 2, 4 and 8 threads, respectively. Figure 25 shows the trend of time cost of these 4 algorithms with b from 1 to 100. We find that the more threads we use, the less time the algorithm costs. The most significant time cost gap is between $\mathtt{DGAC}_1^1$ and $\mathtt{DGAC}_1^2$, and with the number of threads becoming larger, the time cost gap becomes less. This is because with more threads involved, the last finished thread becomes the bottleneck of the whole algorithm. We can also find that the trends of the 4 lines are always similar, even though sometimes they do not grow smoothly. This is because the effect of our reuse mechanism varies from different chosen anchors in different iteration, but is regardless of the number of threads.

8 Conclusion

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. A serial greedy algorithm is proposed to be conducted in single-machine environment with a novel tree-based result reusing mechanism. We also propose effective pruning techniques to reduce the search space. The preliminary version is published in [44]. Then, we extend our algorithm to distributed computing environment with a novel graph partition strategy to ensure the computing independency of each machine. Extensive experiments on 8 real-life networks demonstrate the effectiveness of our model and the efficiency of our algorithms. The reusing mechanism and graph partition strategy shed light on the computations of other problems on hierarchical decomposition, e.g., truss decomposition. It shows that the computation can be divided into independent units and the reuse of intermediate results is feasible.

Acknowledgements Fan Zhang is supported by NSFC62002073. Xuemin Lin is supported by ARC DP200101338. Wenjie Zhang is supported by ARC DP210101393 and ARC DP200101116. Ying Zhang is supported by FT170100128 and ARC DP180103096.

References

1. MPICH. https://www.mpich.org/
2. OpenMP. https://www.openmp.org/
3. Abello, J., Resende, M.G.C., Sudarsky, S.: Massive quasi-clique detection. In: LATIN, pp. 598–612 (2002)
4. Aksu, H., Canim, M., Chang, Y., Korpeoglu, I., Ulusoy, Ö.: Distributed k -core view materialization and maintenance for large dynamic graphs. IEEE Trans. Knowl. Data Eng. **26**(10), 2439–2452 (2014)
5. Alvarez-Hamelin, J.I., Dall'Asta, L., Barrat, A., Vespignani, A.: Large scale networks fingerprinting and visualization using the k-core decomposition. In: NeurIPS, pp. 41–50 (2005)
6. Aridhi, S., Brugnara, M., Montresor, A., Velegrakis, Y.: Distributed k-core decomposition and maintenance in large dynamic graphs. In: DEBS, pp. 161–168. ACM (2016)
7. Bader, G.D., Hogue, C.W.V.: An automated method for finding molecular complexes in large protein interaction networks. BMC Bioinform. **4**, 2 (2003)
8. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. arXiv:cs.DS/0310049 (2003)
9. Bhawalkar, K., Kleinberg, J.M., Lewi, K., Roughgarden, T., Sharma, A.: Preventing unraveling in social networks: the anchored k-core problem. In: ICALP, pp. 440–451 (2012)
10. Bhawalkar, K., Kleinberg, J.M., Lewi, K., Roughgarden, T., Sharma, A.: Preventing unraveling in social networks: the anchored k-core problem. SIAM J. Discrete Math. **29**(3), 1452–1475 (2015)
11. Blanco, M.P., Low, T.M., Kim, K.: Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In: HPEC, pp. 1–7. IEEE (2019)
12. Bola, M., Sabel, B.A.: Dynamic reorganization of brain functional networks during cognition. NeuroImage **114**, 398–413 (2015)
13. Bonchi, F., Khan, A., Severini, L.: Distance-generalized core decomposition. In: SIGMOD, pp. 1006–1023 (2019)
14. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
15. Carmi, S., Havlin, S., Kirkpatrick, S., Shavitt, Y., Shir, E.: A model of internet topology using k-shell decomposition. Proc. Natl. Acad. Sci. **104**(27), 11150–11154 (2007)
16. Chan, T.H., Sozio, M., Sun, B.: Distributed approximate k-core decomposition and min-max edge orientation: breaking the diameter barrier. In: IPDPS, pp. 345–354. IEEE (2019)
17. Chang, L., Yu, J.X., Qin, L., Lin, X., Liu, C., Liang, W.: Efficiently computing k-edge connected components via graph decomposition. In: SIGMOD, pp. 205–216 (2013)
18. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: ICDE, pp. 51–62 (2011)
19. Cheng, J., Ke, Y., Fu, A.W., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks by h*-graph. In: SIGMOD, pp. 447–458 (2010)
20. Chitnis, R., Fomin, F.V., Golovach, P.A.: Parameterized complexity of the anchored k-core problem for directed graphs. Inf. Comput. **247**, 11–22 (2016)
21. Chitnis, R.H., Fomin, F.V., Golovach, P.A.: Preventing unraveling in social networks gets harder. In: AAAI (2013)
22. Chwe, M.S.-Y.: Communication and coordination in social networks. Rev. Econ. Stud. **67**(1), 1–16 (2000)
23. Cohen, J.: Trusses: cohesive subgraphs for social network analysis. Natl. Secur. Agency Tech. Rep. **16**, 3.1 (2008)
24. Conte, A., Firmani, D., Patrignani, M., Torlone, R.: Shared-nothing distributed enumeration of 2-plexes. In: CIKM, pp. 2469–2472. ACM (2019)
25. Conte, A., Matteis, T.D., Sensi, D.D., Grossi, R., Marino, A., Versari, L.: D2K: scalable community detection in massive networks via small-diameter k-plexes. In: SIGKDD, pp. 1272–1281. ACM (2018)
26. Das, A., Sanei-Mehri, S., Tirthapura, S.: Shared-memory parallel maximal clique enumeration from static and dynamic graphs. ACM Trans. Parallel Comput. **7**(1), 5:1-5:28 (2020)

27. Dourisboure, Y., Geraci, F., Pellegrini, M.: Extraction and classification of dense implicit communities in the web graph. TWEB **3**(2), 7:1-7:36 (2009)
28. Esfandiari, H., Lattanzi, S., Mirrokni, V.S.: Parallel and streaming algorithms for k-core decomposition. In: ICML, Volume 80 of Proceedings of Machine Learning Research, pp. 1396–1405. PMLR (2018)
29. Fang, Y., Cheng, R., Li, X., Luo, S., Hu, J.: Effective community search over large spatial graphs. PVLDB **10**(6), 709–720 (2017)
30. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**(4), 634–652 (1998)
31. García, D., Mavrodiev, P., Schweitzer, F.: Social resilience in online communities: the autopsy of friendster. In: Conference on online social networks, pp. 39–50 (2013)
32. Giatsidis, C., Malliaros, F.D., Thilikos, D.M., Vazirgiannis, M.: Corecluster: A degeneracy based graph clustering framework. In: AAAI, pp. 44–50 (2014)
33. Hua, Q., Shi, Y., Yu, D., Jin, H., Yu, J., Cai, Z., Cheng, X., Chen, H.: Faster parallel core maintenance algorithms in dynamic graphs. IEEE Trans. Parallel Distrib. Syst. **31**(6), 1287–1300 (2020)
34. Huang, X., Cheng, H., Qin, L., Tian, W., Yu, J.X.: Querying k-truss community in large and dynamic graphs. In: SIGMOD, pp. 1311–1322 (2014)
35. Jin, H., Wang, N., Yu, D., Hua, Q., Shi, X., Xie, X.: Core maintenance in dynamic graphs: a parallel approach based on matching. IEEE Trans. Parallel Distrib. Syst. **29**(11), 2416–2428 (2018)
36. Kabir, H., Madduri, K.: Parallel k-core decomposition on multicore platforms. In: IPDPS workshops, pp. 1482–1491. IEEE Computer Society (2017)
37. Kabir, H., Madduri, K.: Parallel k-truss decomposition on multicore systems. In: HPEC, pp. 1–7. IEEE (2017)
38. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations, pp. 85–103 (1972)
39. Khaouid, W., Barsky, M., Venkatesh, S., Thomo, A.: K-core decomposition of large networks on a single PC. PVLDB **9**(1), 13–23 (2015)
40. Kitsak, M., Gallos, L.K., Havlin, S., Liljeros, F., Muchnik, L., Stanley, H.E., Makse, H.A.: Identification of influential spreaders in complex networks. Nat. Phys. **6**(11), 888 (2010)
41. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)
42. Li, R., Qin, L., Ye, F., Yu, J.X., Xiao, X., Xiao, N., Zheng, Z.: Skyline community search in multi-valued networks. In: SIGMOD, pp. 457–472 (2018)
43. Lin, J.-H., Guo, Q., Dong, W.-Z., Tang, L.-Y., Liu, J.-G.: Identifying the node spreading influence with largest k-core values. Phys. Lett. A **378**(45), 3279–3284 (2014)
44. Linghu, Q., Zhang, F., Lin, X., Zhang, W., Zhang, Y.: Global reinforcement of social networks: the anchored coreness problem. In: SIGMOD, pp. 2211–2226. ACM (2020)
45. Malliaros, F.D., Rossi, M.-E.G., Vazirgiannis, M.: Locating influential nodes in complex networks. Sci. Rep. **6**, 19307 (2016)
46. Malliaros, F.D., Vazirgiannis, M.: To stay or not to stay: modeling engagement dynamics in social graphs. In: CIKM, pp. 469–478 (2013)
47. Mandal, A., Hasan, M.A.: A distributed k-core decomposition algorithm on spark. In: BigData, pp. 976–981. IEEE Computer Society (2017)
48. Matula, D.W., Beck, L.L.: Smallest-last ordering and clustering and graph coloring algorithms. J. ACM **30**(3), 417–427 (1983)
49. Montresor, A., Pellegrini, F.D., Miorandi, D.: Distributed k-core decomposition. IEEE Trans. Parallel Distrib. Syst. **24**(2), 288–300 (2013)
50. Morone, F., Del Ferraro, G., Makse, H.A.: The k-core as a predictor of structural collapse in mutualistic ecosystems. Nat. Phys. **15**(1), 95 (2019)
51. Pei, J., Jiang, D., Zhang, A.: On mining cross-graph quasi-cliques. In: SIGKDD, pp. 228–238 (2005)
52. Seidman, S.B.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
53. Seki, K., Nakamura, M.: The collapse of the friendster network started from the center of the core. In: ASONAM, pp. 477–484 (2016)
54. Seki, K., Nakamura, M.: The mechanism of collapse of the Friendster network: what can we learn from the core structure of Friendster? Soc. Netw. Anal. Min. **7**(1), 10:1-10:21 (2017)
55. Shao, Y., Chen, L., Cui, B.: Efficient cohesive subgraphs detection in parallel. In: SIGMOD, pp. 613–624 (2014)
56. Smith, S., Liu, X., Ahmed, N.K., Tom, A.S., Petrini, F., Karypis, G.: Truss decomposition on shared-memory parallel systems. In: HPEC, pp. 1–6. IEEE (2017)
57. Tootoonchi, B., Srinivasan, V., Thomo, A.: Efficient implementation of anchored 2-core algorithm. In: ASONAM, pp. 1009–1016 (2017)
58. Ugander, J., Backstrom, L., Marlow, C., Kleinberg, J.M.: Structural diversity in social contagion. Proc. Natl. Acad. Sci. U.S.A. **109**(16), 5962–5966 (2012)
59. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
60. Wang, Z., Chen, Q., Hou, B., Suo, B., Li, Z., Pan, W., Ives, Z.G.: Parallelizing maximal clique and k-plex enumeration over graph data. J. Parallel Distrib. Comput. **106**, 79–91 (2017)
61. Wen, D., Qin, L., Zhang, Y., Lin, X., Yu, J.X.: I/O efficient core graph decomposition at web scale. In: ICDE, pp. 133–144 (2016)
62. Wu, S., Sarma, A.D., Fabrikant, A., Lattanzi, S., Tomkins, A.: Arrival and departure dynamics in social networks. In: WSDM, pp. 233–242 (2013)
63. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud. USENIX Association (2010)
64. Zhang, F., Yuan, L., Zhang, Y., Qin, L., Lin, X., Zhou, A.: Discovering strong communities with user engagement and tie strength. In: DASFAA, pp. 425–441 (2018)
65. Zhang, F., Zhang, W., Zhang, Y., Qin, L., Lin, X.: OLAK: an efficient algorithm to prevent unraveling in social networks. PVLDB **10**(6), 649–660 (2017)
66. Zhang, F., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Finding critical users for social network engagement: the collapsed k-core problem. In: AAAI, pp. 245–251 (2017)
67. Zhang, F., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Efficiently reinforcing social networks over user engagement and tie strength. In: ICDE, pp. 557–568 (2018)
68. Zhang, H., Zhao, H., Cai, W., Liu, J., Zhou, W.: Using the k-core decomposition to analyze the static structure of large-scale software systems. J. Supercomput. **53**(2), 352–369 (2010)
69. Zhao, F., Tung, A.K.H.: Large scale cohesive subgraphs discovery for social network visual analysis. PVLDB **6**(2), 85–96 (2012)
70. Zhou, R., Liu, C., Yu, J.X., Liang, W., Chen, B., Li, J.: Finding maximal k-edge-connected subgraphs from a large graph. In: EDBT, pp. 480–491 (2012)
71. Zhou, Y., Xu, J., Guo, Z., Xiao, M., Jin, Y.: Enumerating maximal k-plexes with worst-case time guarantee. In: AAAI, pp. 2442–2449. AAAI Press (2020)
72. Zhou, Z., Zhang, F., Lin, X., Zhang, W., Chen, C.: K-core maximization: an edge addition approach. In: IJCAI, pp. 4867–4873 (2019)