Towards Structural Stability of Social Networks

by

Qingyuan Linghu

B.E. ZHEJIANG UNIVERSITY, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE SCHOOL

OF

COMPUTER SCIENCE AND ENGINEERING

UNSW
AUSTRALIA

May 13, 2022

🎓 Welcome to the Research Alumni Portal, Qingyuan Linghu!

You will be able to download the finalised version of all thesis submissions that were processed in GRIS here.

Please ensure to include the **completed declaration** (from the Declarations tab), your **completed Inclusion of Publications Statement** (from the Inclusion of Publications Statement tab) in the final version of your thesis that you submit to the Library.

Information on how to submit the final copies of your thesis to the Library is available in the completion email sent to you by the GRS.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | **Declarations** | **Inclusion of Publications Statement**

Corrected Thesis and Responses

ORIGINALITY STATEMENT

☑ I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

COPYRIGHT STATEMENT

☑ I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.

AUTHENTICITY STATEMENT

☑ I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

🎓 Welcome to the Research Alumni Portal, Qingyuan Linghu!

You will be able to download the finalised version of all thesis submissions that were processed in GRIS here.

Please ensure to include the **completed declaration** (from the Declarations tab), your **completed Inclusion of Publications Statement** (from the Inclusion of Publications Statement tab) in the final version of your thesis that you submit to the Library.

Information on how to submit the final copies of your thesis to the Library is available in the completion email sent to you by the GRS.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | **Declarations** | **Inclusion of Publications Statement**

Corrected Thesis and Responses

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☑ The candidate has declared that **their thesis has publications - either published or submitted for publication - incorporated into it in lieu of a Chapter/s. Details of these publications are provided below.**.

Publication Details #1

Full Title:	Global reinforcement of social networks: The anchored coreness problem
Authors:	Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang
Journal or Book Name:	Proceedings of the 2020 ACM SIGMOD International Conference of Management of Data
Volume/Page Numbers:	pages 2211-2226
Date Accepted/Published:	June 2020

Status:	published
The Candidate's Contribution to the Work:	The candidate is the first author of this paper and did the problem formulation, algorithm designing, experimenting and paper writing.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	The paper is in Chapter 3 (3.1-3.4 & 3.6-3.7) of the thesis.

Publication Details #2

Full Title:	Anchored coreness: efficient reinforcement of social networks
Authors:	Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang
Journal or Book Name:	The VLDB Journal
Volume/Page Numbers:	pages 1-26
Date Accepted/Published:	31 May 2021
Status:	published
The Candidate's Contribution to the Work:	The candidate is the first author of this paper and did the problem formulation, algorithm designing, experimenting and paper writing.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	The paper is in Chapter 3 (3.5-3.7) of the thesis.

Publication Details #3

Full Title:	Towards User Engagement Dynamics in Social Networks
Authors:	Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang
Journal or Book Name:	Arxiv preprint
Volume/Page Numbers:	2110.12193
Date Accepted/Published:	
Status:	submitted
The Candidate's Contribution to the Work:	The candidate is the first author of this paper and did the problem formulation, algorithm designing, experimenting and paper writing. This paper has been preprinted at arXiv:2110.12193.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	The content of the preprinted paper is included in Chapter 4 of the thesis.

Publication Details #4

Full Title:	Quantifying Node Importance over Network Structural Stability
Authors:	Fan Zhang, Qingyuan Linghu, Jiadong Xie, Kai Wang, Xuemin Lin, Wenjie Zhang
Journal or Book Name:	Very Large Data Base Endowment Inc. (VLDB Endowment)
Volume/Page Numbers:	
Date Accepted/Published:	
Status:	submitted
The Candidate's Contribution to the Work:	The candidate is the second author of this paper and did the problem formulation, algorithm designing, part of the experimenting and paper writing. This paper is going to be submitted on 1 June 2022 to the VLDB Endowment.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	The paper is adapted from the arxiv preprinted paper 'Towards User Engagement Dynamics in Social Networks' so that is incorporated together in Chapter 4 of the thesis.

Candidate's Declaration

I confirm that where I have used a publication in lieu of a chapter, the listed publication(s) above meet(s) the requirements to be included in the thesis. I also declare that I have complied with the Thesis Examination Procedure.

Abstract

The structural stability of a social network indicates the ability of the network to maintain a sustainable service, which is important for both the network holders and the participants. Graphs are widely used to model social networks, where the coreness of a vertex (node) has been validated as the "best practice" for capturing a user's engagement. Based on this argument, we study the following problems: 1) reinforcing the network structural stability by detecting critical users, with its efficient solution in distributed computation environment; 2) monitoring each user's influence on the network structural stability.

Firstly, we aim to reinforce a social network in a global manner, instead of focusing on a local view as existing works, e.g., the anchored k-core problem aims to enlarge the size of k-core with a fixed input k. We propose a new model so-called the anchored coreness problem: anchoring a small number of users to maximize the coreness gain (the total increment of coreness) of all the users 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.

Secondly, although the static engagement of a user is well estimated by its core-

ness, each user's influence on other users is not well monitored when its engagement is weakened or strengthened. Besides, the dynamic of user engagement has not been well captured for evolving networks. We systematically study the network dynamic against the engagement change of each user. The influence of a user is monitored via two novel concepts: the collapsed power to measure the effect of user weakening, and the anchored power to measure the effect of user strengthening. The two concepts can be naturally integrated such that a unified offline algorithm is proposed to compute both the collapsed and anchored followers for each user. When the network structure evolves, online techniques are designed to maintain the users' followers, which is faster than redoing the offline algorithm by around 3 orders of magnitude.

Publications

- **Qingyuan Linghu**, Fan Zhang, Xuemin Lin, Wenjie Zhang and Ying Zhang. Global reinforcement of social networks: The anchored coreness problem. In Proceedings of the 2020 ACM SIGMOD International Conference of Management of Data, pages 2211-2226. (Chapter 3)

- **Qingyuan Linghu**, Fan Zhang, Xuemin Lin, Wenjie Zhang and Ying Zhang. Anchored coreness: efficient reinforcement of social networks. The VLDB Journal, pages 1-26, 2021. (Chapter 3)

- **Qingyuan Linghu**, Fan Zhang, Xuemin Lin, Wenjie Zhang and Ying Zhang. Towards User Engagement Dynamics in Social Networks. arXiv:2110.12193. (Chapter 4)

- Fan Zhang, **Qingyuan Linghu**, Jiadong Xie, Kai Wang, Xuemin Lin, Wenjie Zhang. Quantifying Node Importance over Network Structural Stability. VLDB 2023 (being submitted) (Chapter 4)

Dedication

my families

my supervisors

my friends

For their love and support

Acknowledgements

I would like to begin by expressing my heart-felt gratitude to my supervisor, Prof. Xuemin Lin, for his guidance and support, without which I would not be able to achieve my present achievements. His insight and knowledge into the subject matter steered me through this research. He gave me selfless guidance and helped me revise and polish my research works. He never failed to motivate me during my doctoral studies even when I was struggling to move forward with research. I am very lucky to have him as my advisor and never expected to have such a good supervisor for the PhD program as well. I am also thankful to my joint-supervisor Prof. Wenjie Zhang and Prof. Fan Zhang for their helpful discussions and insightful suggestions for the work in this thesis.

Besides, special thanks to all the people I have met in this wonderful research group: Prof. Ying Zhang, Dr. Xin Cao, Dr. Lu Qin, Dr. Longbin Lai, Dr. Xing Feng, Dr. Yang Yang, Dr. Haida Zhang, Dr. Dian Ouyang, Dr. Xubo Wang, Dr. Wei Li, Dr. Dong Wen, Dr. Chen Zhang, Dr. Kai Wang, Dr. You Peng, Dr. Boge Liu, Dr. Hanchen Wang, Dr. Xuefeng Chen, Dr. Xiaoshuang Chen, Dr. Conggai Li, Dr. Maryam Ghafouri, Ms. Danyang Wang, Ms. Yuanyuan Xu, Mr. Yu Hao, Mr. Yuren Mao, Mr. Yixing Yang, Mr. Michael Ruisi Yu, Mr. Chenji Huang, Mr. Kongzhang Hao, Mr. Zhengyi Yang, Mr. Peilun Yang, Mr. Yuanhang Yu, Mr. Gengda Zhao, Mr. Shunyang Li, Mr. Qingshuai Feng, Mr. Yizhang He. The time we spent together will be memorized forever.

Last but not least, I would like to thank my families and friends for their love and support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

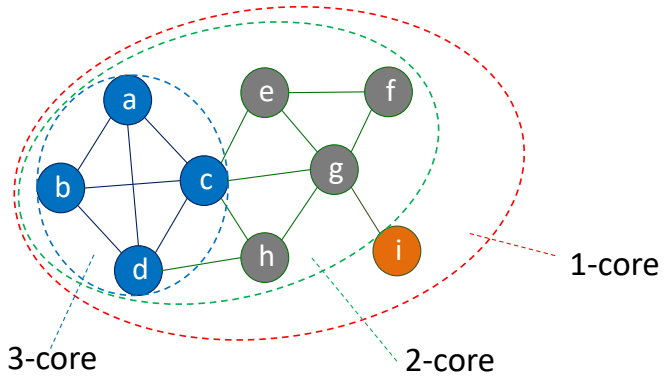
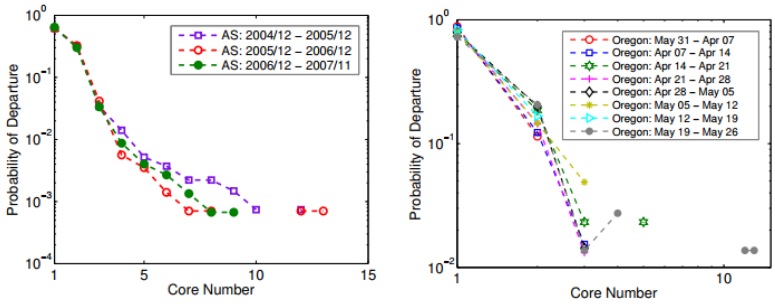


Figure 1.1: The k-Core Model

The analysis of social networks has been shown powerful in many applications with the growing capacity and activity networking sites, e.g., TikTok, Instagram and Facebook etc. Maintaining the structural stability of a social network is one of the significant applications because it is an indicator for a sustainable service which is important for both the network holders and the participants. Social networks are often modeled as graphs when studied, where the users are represented by vertices and the relationship

between users are represented by edges. In graph theory, The k-core [85, 76] model is widely applied because its degeneration property well captures the engagement dynamics in real-life social networks. Given a graph, the k-core is defined as the maximal subgraph where each vertex has at least k neighbors (degree) in the subgraph. For example, in Figure 1.1, the subgraph in red circle is the 1-core satisfying the degree of each vertex is not less than 1. Similarly, the subgraph in green (resp. blue) circle is the 2-core (resp. 3-core) of the graph. The procedure k-core decomposition iteratively removes every vertex with degree less than k. Then every vertex in the graph has a unique *coreness* value, aka. core number, that is, the largest k s.t. the k-core contains the vertex. For the example in Figure 1.1, the coreness of vertex `i` is 1, the coreness of vertex `e`, `f`, `g` & `h` is 2, and the coreness of vertex `a`, `b`, `c` & `d` is 3.



(a) CAIDA (b) OREGON

Figure 1.2: Coreness v.s. Engagement

Existing works have well studied the effect of coreness on capturing the user engagement. In [74], the coreness of a user is demonstrated as the "best practice" for its engagement. Figure 1.2 is from [74] which provides some empirical observations regarding the departure of users and the correlation with their coreness. Because the social networks where the users explicitly define their departure time are difficult to access, they examined two snapshots of the Internet topology (`CAIDA` and `OREGON` Autonomous Systems) with dropout information. Figure 1.2 visualizes the coreness of each

user and its probability of departure, which shows the users with smaller coreness are more possible to leave the network. This demonstrates the positive correlation between a user's engagement and its coreness value.

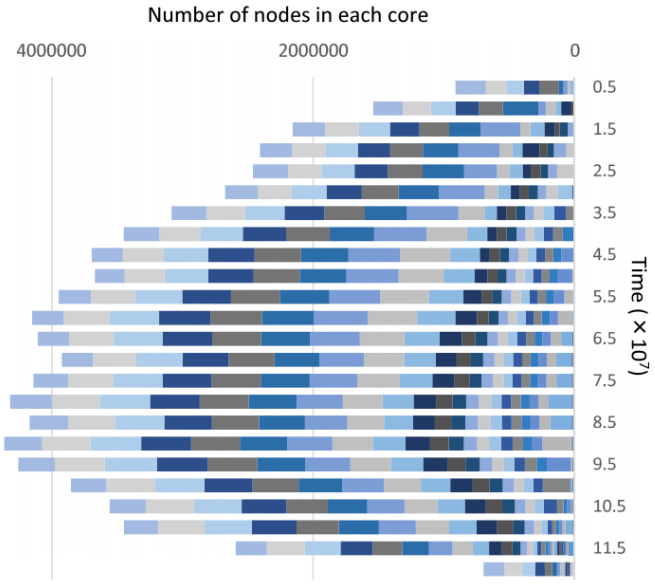


Figure 1.3: The Collapse of Friendster

Seki and Nakamura [86] further use the coreness of users to explain the collapse of a real-life social network, i.e., Friendster, which was once popular in Asia and America in the early 21st century. They reasonably explain the evolution of the number of active users, which firstly increased then decreased with time going by. Figure 1.3 is from [86], where each band shows the core structure of the network at a timestamp. Within a band, the leftmost fragment presents the number of users with coreness less than 5, and the next fragment presents the number of users with coreness between 5 and 9, and so on. It is shown that the users with higher coreness values started collapsing at around the time 8.5×10^7, however, the total number of active users reached the peak at around the time 9×10^7. This means the collapse of Friendster started from the center of k-core structure, i.e., the users with higher engagement collapse before the total number of active users start decreasing.

1.2 Motivation

1.2.1 Global Reinforcement of Social Networks

The leave of users in a social network may cause negative influence to the engagement level of their friends, and thus these friends may also choose to leave [74]. The cascade of user departures can significantly bring down the overall user engagement (structural stability) of a social network. Due to the degeneracy property as illustrated above, k-core decomposition naturally models the process of user departure in a network. In [12], Bhawalker and Kleinburg *e*t al.assume that each user incurs an integer cost of $k > 0$ to remain engaged and obtains a benefit of 1 from each of its neighbors who are engaged, the natural equilibrium of this model corresponds to the k-core of the social network, i.e., users with less than k engaged friends leave the network until the remained engaged users form the k-core.

As the size of k-core is a feasible indicator of network stability, the *anchored k-core* (AK) problem is proposed in [12]: given a graph G, and 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 (number of neighbors) of an anchored vertex is considered as positive infinity, that is, an anchored vertex will stay in the k-core regardless of its original degree. It is meaningful to reinforce a network by giving incentives to some users, e.g., anchored vertices, such that they will keep engaged in the network and support the engagement of other users [12].

Nevertheless, the 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. Given a fixed k, the AK problem can only increase the coreness of a partial set of vertices, i.e., the vertices with coreness $k - 1$, proven in [102]. Besides, the valid vertices for anchoring are from a small set of vertices, and anchoring other vertices are invalid for enlarging the

size of k-core. Furthermore, determining a proper input value of k for the AK problem is not straightforward.

As analyzed in the study of Friendster, the collapse may start from the users with higher coreness values, instead of the assumption of the AK problem, i.e., users with coreness less than k leave first. Thus, it is more promising to reinforce a social network in a "global" manner: considering the coreness increment of every user. Motivated by the above fact, we propose a new model, *anchored coreness* (AC) problem: given a graph G and a budget b, anchor a set of b vertices in the graph s.t. the coreness gain (the total increment of coreness) of all the vertices is maximized. The *followers* of an anchor vertex x are the vertices with coreness increased after anchoring x, except x.

1.2.2 User Influence Monitoring for Network Stability

In the study of network structural stability, it is essential to monitor the engagement of the users in a network, and motivate or protect the critical users. The weakening and strengthening of users are the two natural engagement dynamics of users in a social network. Specifically, in [103] (resp. [12]), when weakening (resp. strengthening) a user, we say it is *collapsed* (resp. *anchored*) such that its degree is regarded as 0 (resp. $+\infty$). When a user x is collapsed (resp. anchored), all the other users which decrease (resp. increase) the coreness values due to collapsing (resp. anchoring) x are called x's *collapsed followers* (resp. *anchored followers*). Then, the influence of a user can be monitored by two aspects: the *collapsed power* by computing its collapsed followers to measure the effect of user weakening, and the *anchored power* by computing its anchored followers to measure the effect of user strengthening.

The application of the novel user monitoring formulation can be summarized into two aspects: 1) the monitored user, and 2) the follower users. For 1), it is important to know how each monitored user can influence the network structural stability, which is

reflected by the collapsed power and anchored power. That is, if a user is found to have a large number of collapsed followers, it needs to be protected from leaving the network as it can cause a cascade of many users leaving; If a user is found to have a large number of anchored followers, it has potential to target at to promote a new service. For 2), it is also important to know the identities of those specific follower users which change their engagement levels due to the collapsing or anchoring of a monitored user. That is, once a user has been found to leave the network, its collapsed followers need to be paid attention to because they are in the risk of engagement decrease; once a user has been anchored, observing its anchored followers can help track the outcome of service promotion.

Motivated by the above promising applications, we aim to study the integration and the efficient computation towards collapsed and anchored followers. Besides, real-life social networks are always evolving, i.e., new signed-up users, new added friends of a user etc. After a network evolves, the change of vertices' collapsed followers and anchored followers can be significant. Thus, we also aim to efficiently maintain the correct collapsed and anchored followers for each user against edge insertions and deletions of the network.

1.3 Contribution

1.3.1 Global Reinforcement of Social Networks

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 $\mathcal{O}(m)$ time by core decomposition [10], 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 func-

tion is non-submodular. Although it is unpromising to estimate or predict the coreness gain of multiple anchors, we observe that the change of coreness is relatively restricted for one anchored vertex. Thus, we adopt a greedy heuristic to find the best anchor in each iteration, while the candidate anchor set is still very large and a straightforward implementation is still very time consuming.

Due to the huge number of candidate anchors, a well-designed reusing strategy is necessary for a greedy heuristic which aims to exhaustively reuse the intermediate results from the executed iterations. To do so, we apply the tree structure $\mathcal{T}$ of core decomposition [10] 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.

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

1.3.2 Efficient Parallel Solution of Network Reinforcement

We then extend GAC to DGAC (**D**istributed) 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 $\mathcal{SP}$. 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* to make the machines evenly and independently share the 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.

1.3.3 User Influence Monitoring for Network Stability

To the best of our knowledge, no existing work studies each user's influence for network structural stability by systematically integrating the collapsed and anchored powers of each user. The naive core decomposition [10] is a straightforward method to compute the collapsed and anchored follower sets for each vertex, simply by regarding the degree of each vertex as 0 and $+\infty$. However, it is certainly cost-prohibitive due to the massive search space. Core maintenance [105] is only to update the coreness value of each vertex itself, after an edge is inserted into or removed from the graph. However, it cannot compute or update the collapsed and anchored follower sets of each vertex. Because of the novelty on the first study, we design and propose our new solution.

Firstly, we propose an offline algorithm to efficiently compute the collapsed and anchored follower sets of each vertex in parallel. Specifically, the graph is divided into

multiple *shell components* induced by all the maximal connected subgraphs where each vertex has the same coreness. Therefore, for either collapse or anchor a vertex x, the follower sets are formed by some vertex subsets from the shell components. We prove those vertex subsets are not overlapped and can be independently computed within each shell component, so that any parallel architecture can be utilized to concurrently compute the collapsed and anchored follower sets. Secondly, we propose an online algorithm to efficiently update the collapsed and anchored follower sets of each vertex. When an edge is inserted into or removed from the graph, we first adopt the state-of-the-art core maintenance algorithm 'k-order' [105] to correctly update each vertex's coreness. Then some new induced shell components are efficiently collected, and the collapsed and anchored follower sets of any vertex can be updated only based on the new shell components. Because the scale of new shell components against one inserted or removed edge is constant, our online maintenance algorithm can achieve around 3 orders of magnitude faster than redoing the offline algorithm.

1.4 Organization

This thesis is organized as follows:

- Chapter 2 introduces the related works on user engagement, cohesive subgraphs and influence maximization.
- Chapter 3 presents our new proposed model, i.e., anchored coreness, and its corresponding algorithm `GAC`, followed by its parallel extension algorithm `DGAC`. The experimental studies for `GAC` and `DGAC` can also be found in the chapter.
- Chapter 4 presents our new model of monitoring users' influence on network structural stability, with its corresponding algorithms and experimental studies.

- Chapter 5 summarizes our research and discusses several possible directions for future work.

Chapter 2

Literature Review

2.1 User Engagement

In this thesis, the objective is to consider the structural stability of social networks, in which the engagements of users play an critical role to improve the user stickiness and avoid the collapse of network. The k-core model is widely applied because its degeneration property well captures the engagement dynamics in real-life social networks, which is verified by Malliaros and Vazirgiannis *e*t al.in [74]. Garcia *e*t al.[42] further use k-core decomposition to explain the decline of Friendster which was once a popular social network in Asia and America in the early 21st century. It is claimed that the collapse of Friendster is stem from the engagements of users (determined by their coreness values) steadily drop down. Under the assertion, their model reasonably explain the evolution of the number of active users, which firstly increased then decreased with time going by. Seki and Nakamura also explain the collapse of Friendster [86], in which they adopt an individual-level model in [18] and conclude that the collapse started from the center of k-core structure. Ugander *e*t al.[91] emphasize that the neighborhood structure hypothesis has formed the underpinning of essentially all the current models for social contagion.

They argue that the probability of contagion is tightly controlled by the number of friends in current subgraph, e.g., k-core, instead of by the actual number of friends in the whole graph. Bhawalkar and Kleinberg *e*t al.[12] propose the anchored k-core model to prevent network unraveling based on game theory, in which the unraveling process terminates when the remaining engaged users correspond to the k-core in the network. Specifically, given a graph G, anchored k-core model aims to anchor b vertices to increase the largest possible number of vertices in the k-core of G. They prove the problem is NP-hard. Then an efficient approximate algorithm for the model is proposed by Zhang *e*t al.in [102]. Zhang *e*t al.further propose the collapsed k-core model [103] to find the users who can significant break the current k-core. Specifically, given a graph G, collapsed k-core aims to remove b vertices in current k-core to decrease the largest possible number of vertices in the k-core of G. This problem is also proved to be NP-hard.

2.2 Cohesive Subgraphs

Many cohesive subgraph models are studied in different scenarios, e.g., clique [16, 25], quasi-clique [3, 82], k-core [76, 85, 15, 43], k-truss [29, 50, 93, 88], k-plexes [108, 30, 31], and k-ecc [22, 107]. In this section, we focus on the three widely applied models, i.e., k-core, k-truss and clique, having incremental cohesiveness. Then we also study the parallel computation for cohesive subgraphs.

2.2.1 k-Core

The concept of k-core and its computing algorithm are first introduced in [85] and [76], where k-core is a the maximal subgraph where each vertex has at least k neighbors in the subgraph. The coreness of a vertex is the largest value of k such that the k-core of the graph contains the vertex. Core decomposition is the procedure for computing the

coreness of each vertex in a graph [10].

An $\mathcal{O}(m+n)$ in-memory algorithm for core decomposition is provided in [10], which iteratively deletes each vertex with the smallest number of neighbors in the remaining graph. For the graphs which are so massive that cannot fit in the main memory of a machine, Wen *e*t al.[97] and Cheng *e*t al.[24] propose I/O efficient algorithms of core decomposition. Coreness of vertices can also be estimated in a local computing way [32, 79]. In [56], Khaouid *e*t al.concludes it is affordable to perform core decomposition for large graphs in a consumer PC. Core maintenance is the procedure of maintaining the coreness of each vertex after inserting or deleting en edge in a dynamic graph. The state-of-the-art core maintenance algorithm is proposed in [105], and other core maintenance approaches include [5, 65, 84].

From the application perspective, the k-core is widely studied in different scenarios such as community discovery [36, 39, 64], finding critical users [103, 101], influential spreader identification [58, 91, 68, 73], discovering protein complexes [9], recognizing hub-nodes in brain function networks [14], analyzing the structure of Internet [19], understanding software networks and its functional consequences [104], predicting structural collapse in ecosystems [78], and graph visualization [7, 106]. One of the reasons why k-core enjoys much popularity in the theory is that it can be used as a subroutine for computing more cohesive subgraphs [56], such as k-truss and clique with size k. The k-core can also provide approximations for the densest subgraph problem and the densest at-least-k subgraph problem [60].

2.2.2 k-Truss

Motivated by k-core, Cohen [29] proposes the model of k-truss, which is a maximal subgraph where each edges has a support (number of common neighbors for its two end vertices) of at least $k-2$ in the subgraph. We also say each edge exists in at least $k-2$

triangles in the subgraph. Each k-truss of the graph is a subgraph of the $(k-1)$-core of the graph. As the k-truss model has requirements for the edges, not only does it capture the users' engagements, but also it ensures the tie strength of relationship between each pair of users. We say each edge in a graph has a trussness which is the largest k such that the k-truss of the graph contains the edge. Then the trussness of each vertex can be computed by the truss decomposition procedure. The first truss decomposition algorithm [29] iteratively deletes the edge with the smallest support in the current graph. It has a time complexity of $\mathcal{O}(\sum_{v \in V(G)}(deg(v)^2))$. Wang *e*t al.[93] then reduce its time complexity to $\mathcal{O}(m^{1.5})$ by detecting the edges that decrease their supports after removing each edge. In [93], an I/O efficient algorithm is also studied when the graph cannot completely fit in the main memory for truss decomposition.

In [106], Zhao *e*t al.introduce the concept k-mutual-friend based on k-truss to capture the cohesion in social interactions, with an I/O efficient algorithm to discover the corresponding cohesive subgraphs. They further integrate these approaches to a community visualizing system where the active social groups and interactions can be quickly located. For probabilistic graphs, Huang *e*t al.[50] extend the k-truss model such that each edge needs to have at least γ (a given threshold) probability to be contained in at least $k-2$ triangles. Also in [50], a k-truss based community model is proposed to further requires the edge connectivity inside a community. Then a corresponding community search algorithm based on a compact tree-shape index is proposed. Furthermore, in dynamic graphs where edges are streamingly inserted and deleted, Huang *e*t al.identify the affected scope in the graph to conduct incremental k-truss community search.

2.2.3 Clique

The clique, being the subgraph model that has the most cohesiveness, has a long history of being studied. Given a graph, a clique is defined as a subgraph in which each pair

of vertices has an edge, which means every vertex is adjacent to each other vertex in this subgraph. If there exists no supergraph of a clique S, S is a maximal clique; If there exists no other clique in the graph having larger number of vertices than S, S is a maximum clique. A clique can ensure the complete reachability and perfect familiarity among its vertex set, but the definition is too strict for widespread applications, i.e., every two vertices are connected is not necessary in many real-life subgraphs. Therefore, some models are proposed to relax the restrictions of clique and are applied in different situations.

In [81], Pattillo *e*t al.review the differences among the classic clique relaxation models. Six properties of clique are investigated including: distance, degree, domination, density, diameter and connectivity, based on which a taxonomy of clique relaxation models is introduced. They also provide some insights for choosing a specific relaxation model over another. Luce [72] proposes the s-clique model that restricts the distance between each two pair of vertices to be less or equal to s within the subgraph. However, the intermediate vertices on a shortest path between two vertices of an s-clique may not be included in the s-clique. s-club [6] is proposed to solve this issue, by requiring all the intermediate vertices on a shortest path to be also included in the s-club, besides all the restrictions of s-clique. Apart from focusing on the diameter of a subgraph as s-clique and s-club, Abello *e*t al.[3] proposes the quasi-clique model which requires the edge density not to be less than a fixed threshold in a subgraph S. Specifically, the edge density of S is the ratio of the actual number of edges in S to the potential number of edges if each pair of vertices in S has an edge.

Maximal clique enumeration is a classic NP-hard graph problem, whose algorithms are mostly based on backtracking search, e.g., [16]. Eppstein *e*t al.[37] further accelerate the maximal clique enumeration algorithm by heuristically selecting the pivots which can potentially reduce the backtracking searching space. The overlaps among different

cliques in a graph are also utilized to speedup the maximal clique enumeration algorithm [94]. A variety of other clique computation algorithms in special circumstances are widely studied, e.g., [26, 21], to name a few.

2.2.4 Parallel Computation for Cohesive Subgraphs

Some works about core decomposition in parallel environments have been studied recently [52, 77, 20, 75, 38]. An algorithm for core decomposition on multicore platforms is introduced in [52]. In [20], a distributed $2(1+\epsilon)$ approximate algorithm of core decomposition is proposed. Based on the distributed framework Spark [100], Mandal *e*t al.[75] use the think-like-a-vertex paradigm to conduct core decomposition. Towards core maintenance, some distributed or parallel algorithms are also proposed [8, 4, 49, 51]. In [49], Hua *e*t al.propose a structure called 'joint edge set' to parallelize inserting/deleting a set of edges, which is based on the idea of 'matching' in [51].

There are also many works studying the 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 problem [34]. In [96], Wang *e*t al.propose an approach for maximal clique and k-plexes enumeration at the same time, which identifies the dense subgraphs by binary graph partitioning, and it is implemented on MapReduce. In [30], a shared-nothing distributed algorithm for k-plexes enumeration is proposed, but only limited to $k = 2$. In [31], Conte *e*t al.present 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, Kabir *e*t al.[53] and Smith *e*t al.[89] develop the parallel algorithms for k-truss decomposition on multicore (shared-memory) system. In [13], a performance exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU is presented.

2.3 Influence Maximization

Apart from studying a user's influence regarding its role in network structural stability, another classic problem which investigates the node influence in networks is *influence maximization* (IM), which focuses on a user's role in information diffusion instead. The IM problem is first presented in [55]. As the analysis of information diffusion has been shown powerful in many applications including the adoption of political standpoints and the commercial value in marketing, the IM problem is being extensively studied in recent years, e.g., [23, 66, 41, 44, 45, 61, 67, 71, 80, 95, 46, 83], to name some. Specifically, IM aims to select a set of k users in a network, aka. seed set with the maximum influence spread, i.e., the expected number of influenced users via the seed set in information diffusion is maximized. The most typical application of IM is viral marketing [35], in which a company hopes to spread the purchase of a product from some initially selected buyers, based on the social relationship between the users. Besides, other typical applications include rumor control [17, 48], network monitoring [62] and social recommendation [99].

The IM problem induces enormous research challenges despite its widespread application potential. Firstly, how to model the information diffusion process in a social network can significantly affect the influence spread of a seed set. Secondly, IM is proven to be NP-hard for obtaining an optimal solution under most of the settings [23, 55, 70]. Moreover, the stochastic nature of information diffusion makes even the evaluation of influence spread of any seed set is cost-prohibitive. It is shown in the above theoretical results that it is very challenging to retrieve a optimal or near optimal seed set from a massive social network. Thirdly, nowadays social networks are equipped with meta information, e.g., topical analysis, streaming content and location-based services etc. The meta information brings new opportunities to improve the effectiveness of IM, by combining IM with different contexts such as locations, timestamps and topic information

etc. Then more technical challenges naturally arise when solving such context-aware IM problems.

Chapter 3

Global Reinforcement of Social Networks

3.1 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 [74]. 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 [42, 87].

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 [11]. The k-core is defined as the maximal subgraph in which every vertex has at least k neighbors in the subgraph [76, 85]. 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., [74, 91, 86].

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* [11, 12]: given a graph G, an integer k and a budget b, anchoring a set of b vertices in the graph s.t. the number of vertices in the k-core is maximized. The degree (the number of neighbors) of an anchored vertex is considered as positive infinity, namely, an anchored vertex will stay in the k-core regardless of its original degree. It is promising to reinforce a network by giving incentives to some users (e.g., anchored vertices) such that they will keep engaged in the network and support the engagement of other users [12]. The anchored k-core problem has been further studied on different aspects, e.g., the theoretical side [28, 27], the experimental evaluation [42, 98] and the efficient solutions [102, 90].

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 [102], 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 [102]. 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) [86] or the outside of center cores [42], i.e, the collapse happens in a "global" way. As shown in [74], 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

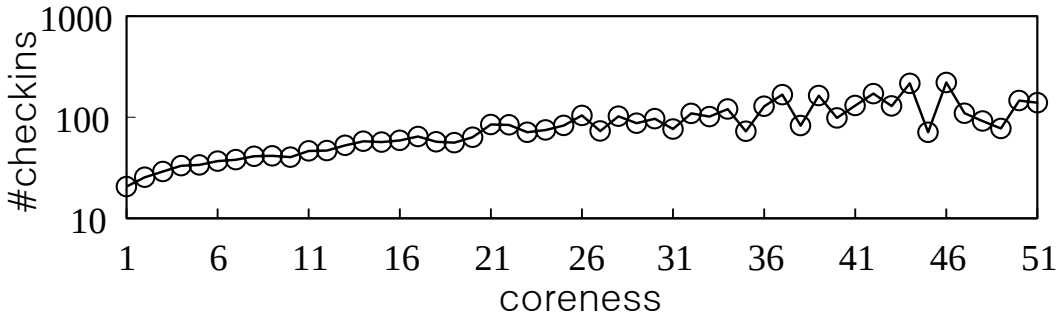


Figure 3.1: Check-in Number v.s. Coreness Value

ground-truth user engagement) for the users with coreness equals to k. As shown in Figure 3.1, the coreness value and check-in number in Gowalla [63] 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.

Table 3.1: Anchored k-Core v.s. Anchored Coreness in Fig. 2

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

Example 3.1.1 *Figure 3.2 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 3.1 records the results of anchored k-core (AK) problem and anchored coreness

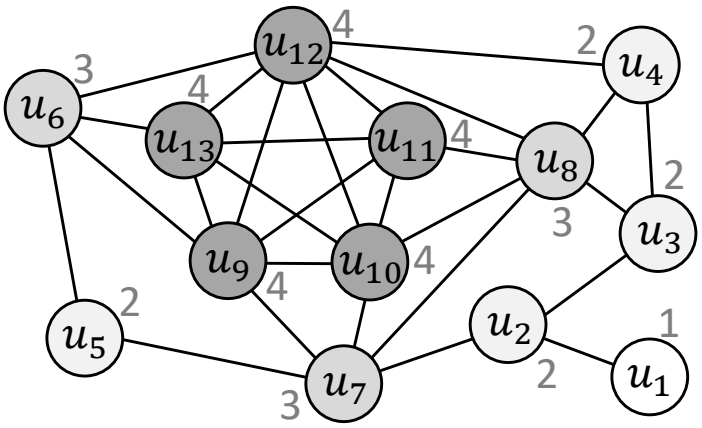


Figure 3.2: A Toy Example

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

Challenges. To the best of our knowledge, we are the first to study the anchored coreness (AC) problem. We prove the AC problem is NP-hard. Although the coreness gain can be computed in $O(m)$ time by core decomposition [10], 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 [102], 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 [102] 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 (Section 3.4.3) 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 $\mathcal{T}$ (Section 3.4.1) of core decomposition [10] 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 (Section 3.4.4) 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* (Section 3.4.5) 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 (Section 3.4.5) 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 $\mathcal{SP}$ (Section 3.5.1). 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* (Section 3.5.3) 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 chapter, we overcome all the challenges with above solutions. The preliminary version is published in [69]. 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.

Table 3.2: Summary of Notations throughout the Chapter

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 neighbour 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 i-th layer of the k-shell, i.e., $u \in H_k^i(G)$.
$x \rightsquigarrow u$	an upstair path from x to u
$CF(x)$	the candidate followers set of x
$d^+(x)$	the degree bound of x
$UB_\sigma(x)$	the upper bound of $\|\mathcal{F}(x)\|$

3.2 Preliminaries and Problem Statement

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 neighbour vertex set of u in G. Table 4.1 summarizes some notations used throughout this chapter. 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 3.2.1 *k-core [76, 85]. 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

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

Input : a graph G, an anchor set A

Output : $c^A(u, G)$ for each $u \in V(G)$

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

vertex in G has a unique coreness.

Definition 3.2.2 ***coreness***. *Given a graph G, the coreness of a vertex $u \in V(G)$, denoted by $c(u, G)$, is the largest k such that $C_k(G)$ contains u, i.e., $c(u, G) = max\{k \mid u \in C_k(G)\}$.*

Definition 3.2.3 ***core decomposition***. *Given a graph G, core decomposition of G is to compute the coreness of every vertex in $V(G)$.*

In this chapter, 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 [10], 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 14.

Definition 3.2.4 ***coreness gain***. *Given a graph G and an anchor set A, the coreness gain of G regarding A, denoted by $g(A, G)$, is the total increment of coreness for every vertex in $V(G) \setminus A$, i.e., $g(A, G) = \sum_{u \in V(G) \setminus A}(c^A(u) - c(u))$.*

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

3.3 Problem Analysis

Theorem 3.3.1 *Given a graph G, the anchored coreness problem is NP-hard.*

Proof 3.3.1 *We reduce the maximum coverage (MC) problem [54], 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 $\{e_1, .., e_d\} = \cup_{1 \leq i \leq c} T_i$, we construct a corresponding instance of the anchored coreness problem on a graph G. W.l.o.g., we assume $b < c < d$. Figure 3.3 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., $M = \cup_{1 \leq i \leq c} w_i$ where each w_i corresponds to the set T_i in the MC instance H. The part N contains d vertices, i.e., $N = \cup_{1 \leq i \leq d} v_i$ 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 $v_j \in (V(G) \setminus M)$, we have $deg(w_i) \leq d < deg(v_j)$. Recall that the core decomposition of G iteratively deletes the vertices with degree less than k and assigns the coreness of $k-1$ to the deleted vertices in current iteration, from $k = 1, 2, ...$ to $k = k_{max}$. Thus, the coreness of each $w_i \in M$ is $deg(w_i)$, as w_i can only be deleted when $k = deg(w_i) + 1$. The coreness of each $v_j \in N$ is d, as v_j is not deleted when $k = d$ (due to the d cliques), and v_j is deleted when $k = d + 1$ (due to the deletion of every $w_i \in M$). Similarly, the coreness of every vertex in a $(d+2)$-clique is $d+1$.

For each $w_i \in M$, even if all the neighbors of w_i are anchored, the coreness of w_i keeps 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 3.3.2 *For any $\epsilon > 0$, the anchored coreness problem cannot be approximated in polynomial time within a ratio of $(1 - 1/e + \epsilon)$, unless P=NP.*

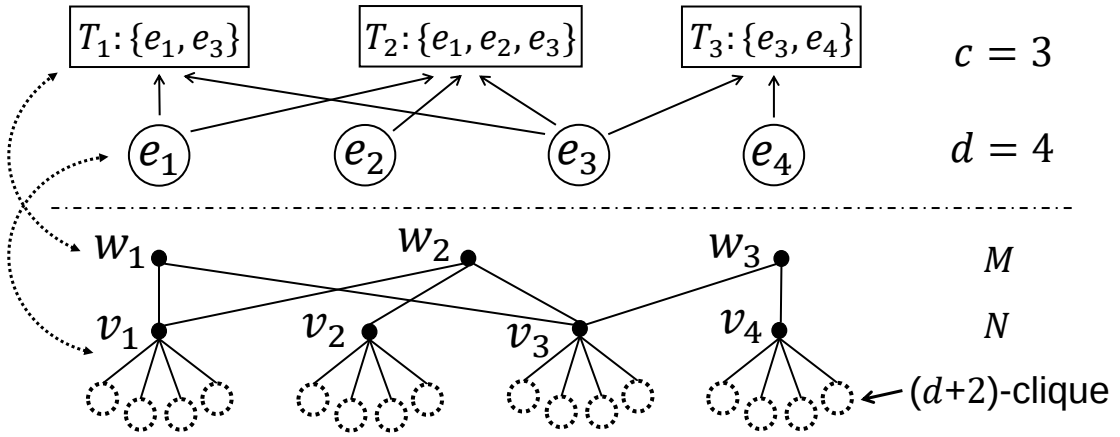


Figure 3.3: Construction Example for Hardness Proofs

Proof 3.3.2 *We use the reduction from the MC problem same to the proof of Theorem 3.3.1. For any $\epsilon > 0$, the MC problem cannot be approximated in polynomial time within a ratio of $(1 - 1/e + \epsilon)$, unless $P = NP$ [40]. We have an anchor set A for anchored coreness problem on G corresponding to a set collection C for MC problem, where each $w_i \in A$ corresponds to $T_i \in C$. Let $\gamma > 1 - 1/e$, if there is a solution with γ-approximation on the coreness gain for the anchored coreness problem, there will be a γ-approximate solution on optimal element number for the MC problem.*

Besides, the function of coreness gain is not submodular.

Theorem 3.3.3 *The function $g(\cdot)$ of coreness gain is not submodular.*

Proof 3.3.3 *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 $B = \{v_6\}$, $g(A) + g(B) = 0 < g(A \cup B) + g(A \cap B) = 4$.*

3.4 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 (Section 3.4.1), and the theorems of finding the candidate followers which may increase the coreness due to the anchoring (Section 3.4.2). Based on the tree and the theorems, we propose a mechanism to reuse the intermediate results across iterations (Section 3.4.3), and the algorithm to compute the coreness gain of one anchor by partially exploring the tree (Section 3.4.4). Combining the above with an upper bound technique for candidate anchors pruning, our final GAC algorithm is presented (Section 3.4.5).

3.4.1 Core Component Tree

Definition 3.4.1 ***k-core component.*** *Given a graph G and the k-core $C_k(G)$, a subgraph S is a k-core component if S is a connected component of $C_k(G)$.*

According to the definition of k-core, for every integer k, we have *disjointness* prop-

Table 3.3: 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 $v \in \mathcal{F}(x) \ \wedge \ \mathcal{T}[v].I = id$

erty: 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 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

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** $\mathcal{T}$

Table 3.3.

Algorithm 2 illustrates the structure of a core component tree. It can be implemented in $O(m)$ time as shown in [76]. 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 3.4.2 ***tree node classified adjacency*** (tca)**.** *For a given graph G, we scan the neighbour 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 3.4.3 ***subtree adjacent nodes set*** (sn) *Given a vertex u in a graph G, the subtree adjacent nodes set of u, denoted by $sn(u)$ is the id set of adjacent tree nodes with the associated coreness not less than $c(u)$, i.e., $id \in sn(u)$ iff $\exists v \in N(u, G)$ having $c(v) \geq c(u) \ \wedge \ \mathcal{T}[v].I = id$.*

Definition 3.4.4 ***parent adjacent nodes set*** (pn) *Given a vertex u in a graph G, the parent adjacent nodes set of u, denoted by $pn(u)$ is the id set of adjacent tree nodes with the associated coreness less than $c(u)$, i.e., $id \in pn(u)$ iff $\exists v \in N(u, G)$ having $c(v) < c(u) \ \wedge \ \mathcal{T}[v].I = id$.*

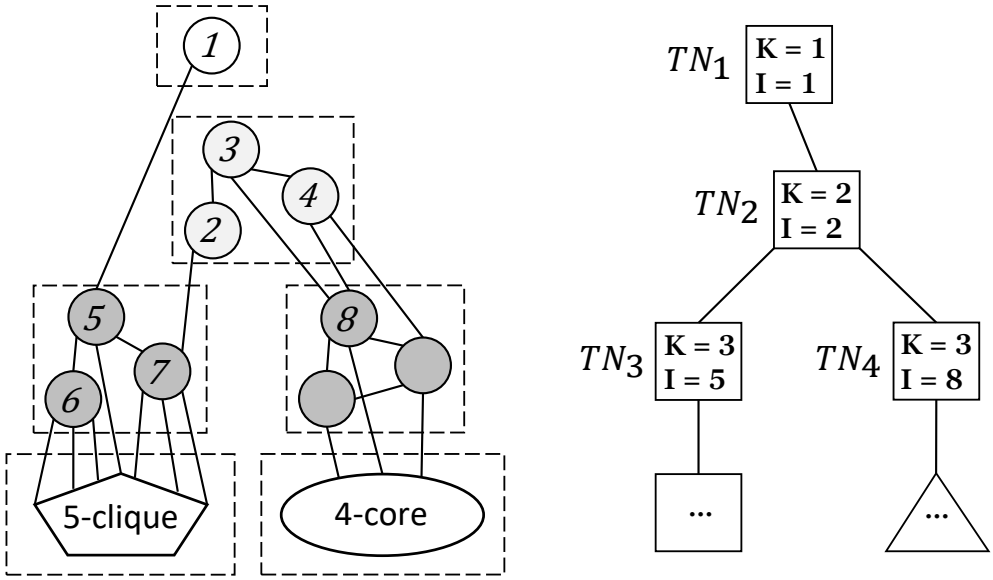


Figure 3.4: Core Component Tree

Example 3.4.1 *In Figure 3.4, 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 $\mathcal{T}$.

3.4.2 Restriction of Candidate Followers

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

Theorem 3.4.1 *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 3.4.1 *We prove it by contradiction. Suppose there is a non-anchor vertex $u \in V(G)$ with coreness increasing from k' to k^* after anchoring x and $k^* > k' + 1$. Let M be the k^*-core after x is anchored, we have $u \in M$ and $deg(v, M) \geq k^*$ for every vertex $v \in M$. If we delete x and its corresponding edges from M, we have $deg(v, M \setminus \{x \cup E(x)\}) \geq k^* - 1$ for every $v \in M$ because at most one edge is removed for each vertex $v \in M$. Thus, $M \setminus \{x \cup E(x)\} \subseteq C_{k^*-1}(G)$. As $u \in M$ and $u \neq x$, we have $u \in C_{k^*-1}(G)$ and thus $k' \geq k^* - 1$ which contradicts with $k^* > k' + 1$.*

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 3.4.1, $g(\{x\}) = |\mathcal{F}(x)|$. 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 Section 3.4.4. Note that, when we record the follower sets, we do not store the specific followers of a vertex x but only store the number of followers of x regarding each adjacent tree node, so the space cost of F is $\mathcal{O}(m)$. The candidate followers of a vertex x can be extracted as follows.

Theorem 3.4.2 *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 3.4.2 *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 [102] that any order following Algorithm 14 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 3.4.1; 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.

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 $u \in V(G)$

3.4.3 Reuse of Intermediate Results

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

According to Theorem 3.4.2, 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 Line 3-6, we remove the tree node ids from $rn(\cdot)$ where the followers cannot be reused by above affected vertices (Line 13-16). In the implementation, for a vertex u, we easily avoid duplicate removals in $rn(u)$ triggered by u's neighbors using tree node tags.

Algorithm 14 (Line 8) and Algorithm 2 (Line 9) both have $\mathcal{O}(m)$ time complexity. In Line 3-6 and Line 13-16, each edge is accessed at most one time, respectively. So, the time complexity of Algorithm 3 is $\mathcal{O}(m)$.

Lemma 3.4.1 *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 3.4.3 *We prove it by contradiction. To prove 1), suppose an $id \in rn(u)$ has $id \notin sn'(u)$. That means ⓐ $\mathcal{T}'[id].V$ does not contain any neighbor of u or ⓑ $\mathcal{T}'[id].V$ contains the neighbors of u but also contains another vertex whose $id' < id$ so $\mathcal{T}'[id].I = id'$ and $id' \in sn'(u)$.*

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

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

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

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

Theorem 3.4.3 *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 3.4.4 *Let O denote a vertex deletion order of core decomposition on G without anchoring x. Similar to the proof of Theorem 3.4.2, 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 3.4.1. Also, we have $\mathcal{T}'[id].K = \mathcal{T}[id].K$ and $\mathcal{T}'[id].V = \mathcal{T}[id].V$ from Lemma 3.4.1. 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 3.4.2. 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 3.4.3. 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 3.4.2 *In Figure 3.4, 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 $2 \in rn(\mathbf{2})$.*

3.4.4 Coreness Gain Computation

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

x.

Given a graph G, the **k-shell**, denoted by $H_k(G)$, is the set of vertices in G with coreness equal to k, i.e., $H_k(G) = V(C_k(G)) \setminus V(C_{k+1}(G))$. The vertices in the k-shell can be further divided to different vertex sets, named layers, according to their deletion sequence in the core decomposition (Algorithm 14). We use H_k^i to denote the i-layer of the k-shell, which is the set of vertices that are deleted in the i-th batch. Specifically, when $i = 1$, H_k^i is defined as $\{u \mid deg(u, C_k(G)) < k+1 \ \wedge \ u \in C_k(G)\}$. 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 i-th layer of the k-shell, i.e., $u \in H_k^i$. We record the shell-layer pair of every vertex u in $\mathcal{P}$. Specifically, for every vertex v, it is contained in the $(\mathcal{P}[v].i)$-th layer of the $(\mathcal{P}[v].k)$-shell in G. We define $\mathcal{P}[v_i] \prec \mathcal{P}[v_j]$ iff $\mathcal{P}[v_i].k < \mathcal{P}[v_j].k$ or $\mathcal{P}[v_i].k = \mathcal{P}[v_j].k \ \wedge \ \mathcal{P}[v_i].i < \mathcal{P}[v_j].i$.

Example 3.4.3 *In Figure 3.5 (a), 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.*

Definition 3.4.5 ***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*

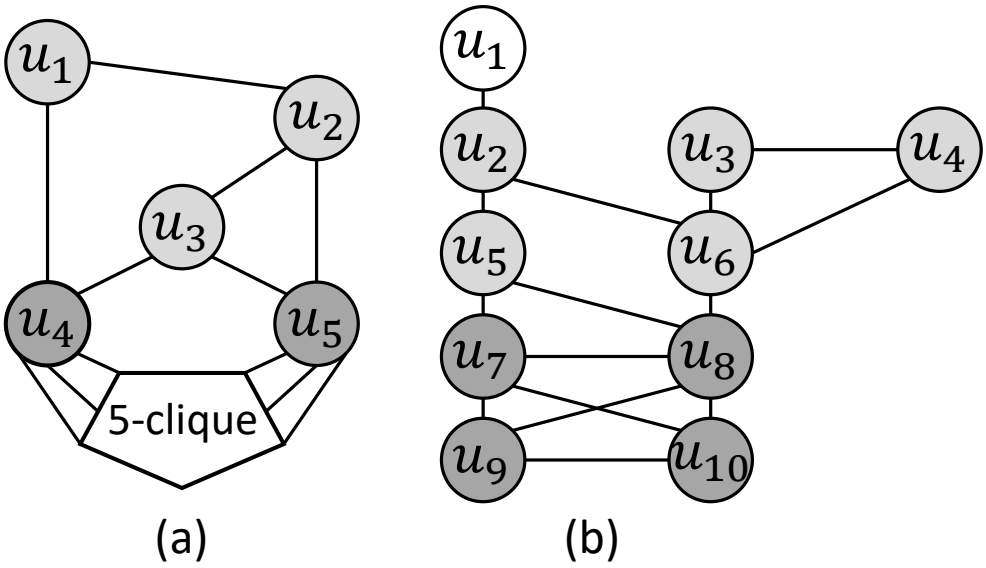


Figure 3.5: Figures for Example 4, 5, 6 & 7

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 3.4.4 *In Figure 3.5 (b), 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 3.4.5. 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 3.4.5), 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 3.4.5.*

Theorem 3.4.4 *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 3.4.5 *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 14). We denote the graph after anchoring x by G_x. For every vertex $v \in V(G_x)$ with $\mathcal{P}[v] \prec \mathcal{P}[x]$, we can follow the same deletion order O in the core decomposition of G_x, and then $c^x(v, G_x) = c(v, G)$ because the degree of v in the order keeps same when v is visited and to be deleted. Thus, $c^x(u, G_x) = c(u, G)$ and u is not a follower of x if $x \in N_u^2$. So $x \notin N_u^2$. (iii) Suppose $x \notin N_u^0 \cup N_u^1 \cup N_u^2$, u must have a neighbor $v_0 \in N_u^1 \cap C_{k+1}(G_x)$; otherwise, $c^x(u, G_x) = c(u, G)$ as in case (ii) following the deletion order O. Thus, if a vertex $v_i \in C_{k+1}(G_x) \setminus C_{k+1}(G)$, v_i must have a neighbor $v_{i+1} \in N_{v_i}^1 \cap C_{k+1}(G_x)$ or $v_{i+1} = x$. Recursively, $u \in \mathcal{F}(x)$ implies there is a path $(x, ..., u)$ which is an upstair path from x to u where each vertex in the path is a follower of x except x itself.

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

Input : x : the anchor, G : a social network, $\mathcal{T}$: the core component tree of G

Output : $F[x][\cdot]$: tree node classified follower sets of x

x is set *survived*;
for each non-reusable tree node $id \in sn(x) \setminus rn(x)$ **do**
  $H := \emptyset$;
  if $id = i_x$ **then**
    $H.push(u)$ **for** each $u \in tca^{\geq}_{=}(x)$;
  else
    $H.push(u)$ **for** each $u \in tca[x][id]$;
  while $H \neq \emptyset$ **do**
    $u \leftarrow H.pop()$;
    Compute $d^+(u)$;
    if $d^+(u) \geq c(u, G) + 1$ **then**
      u is set *survived*;
      for each $v \in tca^{\geq}_{=}(u)$ and $v \notin H$ **do**
        $H.push(v)$;
    else
      u is set *discarded* ;
      Shrink(u);
  $F[x][id] \leftarrow$ *survived* vertices$\setminus\{x\}$;
return $F[x]$

Algorithm 5: **Shrink**(u)

Input : u : the vertex for degree check

1 **for** each *survived* neighbor v with $v \neq x$ **do**
2 $\quad d^+(v) := d^+(v) - 1$;
3 $\quad T \leftarrow v$ **If** $d^+(v) < c(v, G) + 1$;
4 **for** each $v \in T$ **do**
5 $\quad v$ is set *discarded*;
6 $\quad$ **Shrink**(v);

Computing Followers. According to Theorem 3.4.4, the vertices without any upstair path from the anchor vertex x cannot be a follower of x. We use $CF(x)$ to denote all the candidate followers of an anchor x, i.e., the vertices that can be reached by x via upstair paths. Instead of doing core decomposition of the whole graph, we only need to explore the candidate followers $CF(x)$ to compute the follower set of x. We use $tca^{\leq}_{=}(u)$ to denote the set of u's neighbours where each neighbor v has $\mathcal{P}[v].k = \mathcal{P}[u].k \ \wedge \ \mathcal{P}[v].i \leq \mathcal{P}[u].i$. Similarly, $tca^{\geq}_{=}(u)$ contains every u's neighbour v with $\mathcal{P}[v].k = \mathcal{P}[u].k \ \wedge \ \mathcal{P}[v].i > \mathcal{P}[u].i$. 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, $tca^{\leq}_{=}(u)$ and $tca^{\geq}_{=}(u)$ are easily retrieved along with core decomposition.

Algorithm 4 shows the pseudo-code for computing the followers. In each iteration, we search the non-reusable tree nodes (Section 3.4.3) in $\mathcal{T}$ to compute the followers of x in the nodes (Line 2, Algorithm 4). We maintain a min heap H to store the candidate followers $CF(x)$ which will be explored (Line 3-7 and 13-14). The key of a vertex in H is its shell-layer pair with ties broken by the vertex id. In each tree node $id \in sn(x) \setminus rn(x)$, we explore $CF(x)$ in a layer-by-layer manner: from j-th layer to $(j+1)$-th layer starting from x.

In the layer-by-layer search, a vertex is set as **unexplored** if it has never been checked

with the degree constraint (Line 11). A vertex is set as **survived** if it survived the degree check (Line 12), otherwise it is set as **discarded** (Line 16). The *discarded* vertices will 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 $\{x\} \cup (tca_{\leqq}^{<}(u) \cap H) \cup tca_{=}^{>}(u)$, and $d_>(u)$ is the number of neighbors in $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$. The following theorem indicates that we can exclude a candidate follower u if $d^+(u) < c(u, G) + 1$. The discard of a vertex may invoke the discard of other vertices, as shown in Algorithm 5. When the deletion cascade terminates, the tags of all the vertices affected by the discard of u will be correctly updated.

Theorem 3.4.5 *A vertex $u \in CF(x)$ cannot be a follower of x if $d^+(u) < c(u, G) + 1$.*

Proof 3.4.6 *We denote the graph after anchoring x by G_x, and let $k^+ = c(u, G) + 1$. We show if $d^+(u) < c(u, G) + 1$, then $deg(u, C_{k^+}(G_x)) < k^+$, so u cannot be a follower of x. u's neighous can be divided into those in $\bigcup_{id \in pn(u)} tca[u][id]$, $tca_{\leqq}^{<}(u) \cup$*

$tca^{\geq}_{=}(u)$ and $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$, respectively. Obviously the neighbors of $\bigcup_{id \in pn(u)} tca[u][id]$ are not in $C_{k^+}(G_x)$, because they cannot increase the coreness by 2 according to Theorem 3.4.1. For the neighbors in $tca^{\leq}_{=}(u) \cup tca^{\geq}_{=}(u)$, they are all considered in $d^+_s(u)$ or $d^+_u(u)$, unless they are $discarded$ or never pushed to H, both of which mean they are not in $C_{k^+}(G_x)$. At last, for the neighbors in $\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]$, they satisfy $|\bigcup_{id \in sn(u) \setminus \{i_u\}} tca[u][id]| = d_>(u)$. Since $d^+(u)$ considers all the neighbors of u which are possible to be in $C_{k^+}(G_x)$, $d^+(u)$ is a degree bound of $deg(u, C_{k^+}(G_x))$.

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 3.4.5 *In Figure 3.5 (b), we explain an example of using Algorithm 4 to compute the followers of u_1 from a single tree node. For the core component tree $\mathcal{T}$, we can see there are three tree nodes TN_1, TN_2 and TN_3, where $TN_1.V = \{u_1\}$, $TN_1.K = 1$ and $TN_1.I = u_1$; $TN_2.V = \{u_2, u_3, u_4, u_5, u_6\}$, $TN_2.K = 2$ and $TN_2.I = u_2$; $TN_3.V = \{u_7, u_8, u_9, u_{10}\}$, $TN_3.K = 3$ and $TN_3.I = u_7$. Initially, u_1 itself is set survived and we push the only adjacent vertex u_2 which is in $tca[u_1][u_2]$ into the min Heap H. Then we pop u_2 and have $d^+_s(u_2) = 1$, $d^+_u(u_2) = 2$ and $d_>(u_2) = 0$, so u_2 survives the degree check since $d^+(u_2) = c(u_2) + 1$ and we set u_2 survived. We put the vertices of $tca^{\geq}_{=}(u_2)$ into the heap so u_5 and u_6 are now in H. We first explore u_5 and have $d^+_s(u_5) = 1$, $d^+_u(u_5) = 0$, $d_>(u_5) = 2$ and $d^+(u_5) = c(u_5) + 1$, so we set u_5 survived. As $tca^{\geq}_{=}(u_5) = \emptyset$, we do not put any more vertices into H for now. Then we explore u_6 and have $d^+_s(u_6) = 1$, $d^+_u(u_6) = 0$ and $d_>(u_6) = 1$. Note that, u_3 and u_4 are unexplored neighbors of u_6 in $tca^{\leq}_{=}(u_6)$, but they will not be added into H so cannot be counted in $d^+_u(u_6)$. $d^+(u_6) < c(u_6) + 1$ so we will discard it. As illustrated in Algorithm 5, for each survived neighbor of u_6 which is u_2, we make $d^+(u_2) = d^+(u_2) - 1 = 2$ so that $d^+(u_2) < c(u_2) + 1$. So we discard u_2 and 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 (Line 3-17, Algorithm 4) again, and use the existing $F[x][id]$ instead. If $id \notin rn(x)$, we execute the Line 3-17 of Algorithm 4 to find the correct followers.

3.4.5 The GAC Algorithm

We first introduce an upper bound of follower number.

Upper Bound Based Pruning. We introduce an easy-to-compute upper bound to further prune unpromising candidates before the computation of followers. For a vertex x, by Equation 3.1, we firstly get the upper bound of followers from its own tree node $\mathcal{T}[x]$. Then for each $id \in sn(x) \setminus \{i_x\}$, we get an upper bound $UB^{>}_{id}(x)$ by Equation 3.2. At last we can compute the total upper bound $UB_\sigma(x)$ by Equation 3.3. When $tca^{\geq}_{=}(u) = \emptyset$ for a vertex u, we set $UB_{i_u}(u)$ to 0.

$$UB_{i_x}(x) = \sum_{u \in tca^{\geq}_{=}(x)} (UB_{i_u}(u) + 1) \tag{3.1}$$

$$UB^{>}_{id}(x) = \sum_{u \in tca[x][id]} (UB_{i_u}(u) + 1) \tag{3.2}$$

$$UB_\sigma(x) = UB_{i_x}(x) + \sum_{id \in sn(x) \setminus \{i_x\}} UB^{>}_{id}(x) \tag{3.3}$$

Theorem 3.4.6 *Given a graph G and an anchor vertex x, $|F[x][i_x]| \leq UB_{i_x}(x)$, and for each $id \in sn(x) \setminus \{i_x\}$, $|F[x][id]| \leq UB^{>}_{id}(x)$. So, $g(\{x\}, G) \leq UB_{\sigma}(x)$.*

Proof 3.4.7 *According to the Equation 3.1 and Equation 3.2, 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 3.4.4, we can prove that $|F[x][i_x]| \leq UB_{i_x}(x)$ and $|F[x][id]| \leq UB^{>}_{id}(x)$ for each $id \in sn(x) \setminus \{i_x\}$. Then, based on Equation 3.3 and Theorem 3.4.2, 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 3.4.6 *In Figure 3.5(a), after getting the shell-layer pair of each vertex, $\mathcal{P}[u_1] = (2,1)$, $\mathcal{P}[u_2] = (2,2)$, $\mathcal{P}[u_3] = (2,3)$, and $\mathcal{P}[u_4] = \mathcal{P}[u_5] = (3,1)$. Now in $\mathcal{T}$, we have TN_1 where $TN_1.V = \{u_1, u_2, u_3\}$, $TN_1.K = 2$ and $TN_1.I = u_1$. $TN_2.V = \{u_4\}$ where $TN_2.K = 3$ and $TN_2.I = u_4$. $TN_3.V = \{u_5\}$ where $TN_3.K = 3$ and $TN_3.I = u_5$. Then we get a total ordering of them: $u_1 \prec u_2 \prec u_3 \prec u_4 \prec u_5$. We compute their upper bounds following this order. For u_4 and u_5, $UB_{u_4}(u_4) = UB_{u_5}(u_5) = 0$ since $tca^{\geq}_{=}(u_4) = tca^{\geq}_{=}(u_5) = \emptyset$. For u_3, $tca^{\geq}_{=}(u_3) = \emptyset$ so $UB_{u_1}(u_3) = 0$. $tca[u_3][u_4] = \{u_4\}$ and $tca[u_3][u_5] = \{u_5\}$, so that $UB^{>}_{u_4}(u_3) = (UB_{u_4}(u_4) + 1) = 1$ and $UB^{>}_{u_5}(u_3) = (UB_{u_5}(u_5) + 1) = 1$. Therefore, $UB_{\sigma}(u_3) = UB_{u_1}(u_3) + UB^{>}_{u_4}(u_3) + UB^{>}_{u_5}(u_3) = 2$. For u_2, $tca^{\geq}_{=}(u_2) = \{u_3\}$ so $UB_{u_1}(u_2) = (UB_{u_1}(u_3) + 1) = 1$, and $tca[u_2][u_5] = \{u_5\}$ so that $UB^{>}_{u_5}(u_2) = (UB_{u_5}(u_5) + 1) = 1$. Then we have $UB_{\sigma}(u_2) = UB_{u_1}(u_2) + UB^{>}_{u_5}(u_2) = 2$. At last, we get $tca^{\geq}_{=}(u_1) = \{u_2\}$ and $tca[u_1][u_4] = \{u_4\}$, so we can get $UB_{u_1}(u_1) = (UB_{u_1}(u_2) + 1) = 2$, $UB^{>}_{u_4}(u_1) = (UB_{u_4}(u_4) + 1) = 1$, $UB_{\sigma}(u_1) = UB_{u_1}(u_1) + UB^{>}_{u_4}(u_1) = 3$.*

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 14 (Line 1) to get the initial coreness of each vertex of the given graph G. Then, we apply Algorithm 2

(Line 2) to build the core component tree for the first time, followed by the computing of our upper bound of follower numbers (Line 3), which will be updated after anchoring every vertex (Line 12-13). Then, the greedy heuristic starts (Line 4). In each iteration, we use a to record the best anchor vertex found so far, and use λ to record the number of followers of the best anchor (Line 5). We sequentially compute the followers for the vertices in decreasing order of their upper bounds (Line 6). Only if the upper bound of a vertex u is larger than λ and u is not an existing anchor (Line 7), we will continue the follower computation for u (Line 8-10). Note that we will not compute the follower number for u in the tree nodes where the numbers of followers do not change from last iteration and can be reused. After the follower computation of current iteration, the best anchor a is added to the set A, and the degree of a is set to be positive infinity. After b iterations, Algorithm 6 returns the set A of b anchor vertices (Line 14).

3.5 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.

3.5.1 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. Example 4.3.1 and Example 3.5.2 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), then prove their correctness.

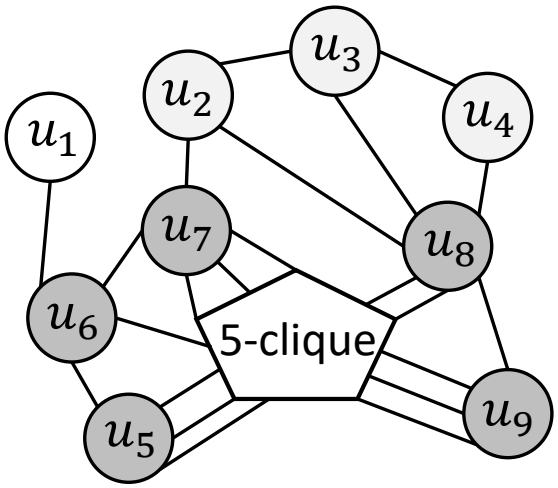


Figure 3.6: k-shell component

Definition 3.5.1 ***k-shell component***. *Given a graph G and the k-shell $H_k(G)$, a subgraph S_k^i is the i-th k-shell component of $H_k(G)$, if S_k^i is a maximal induced connected component of $H_k(G)$.*

Example 3.5.1 *In Figure 3.6, 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 $\exists (u, v) \in E(G) \ \wedge \ v \in SC.V$;

Table 3.4: 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

(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 $u \in SC.V$, $SC.h[u] = |\{v \mid (u, v) \in E(G) \ \wedge \ c(v, G) > c(u, G)\}|$;

(5) $SC.C$, the candidate anchor set in SC, which will be explained in Section 3.5.3 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 of the notations are summarized in Table 3.4.

Example 3.5.2 *In Figure 3.7, we have 5 shell component partitions SC_1, SC_2, SC_3, SC_4 and SC_5, which are affiliated 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 Figure 3.6. 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 $SC_4.h[u_7] = 3$.*

Table 3.5: Other Notations for DGAC

Notation	Definition
S_k^i	the i-th k-shell component
V_x	$(\bigcup_{SC \in \mathcal{SP}[x]} SC.V) \setminus \{x\}$
G_x	the graph G with x anchored
G_{SC}	the subgraph formed by $SC.V$, $SC.V^-$, $SC.E$
$N^{\leq}(u, SC)$; $N^{>}(u, SC)$	set of such u's neighbor v in SC with $\mathcal{P}[v].k = \mathcal{P}[u].k \wedge \mathcal{P}[v].i \leq \mathcal{P}[u].i$ 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 i-th slave machine
$LB(u)$	Equation 3.4
$\mathcal{LB}$	Equation 3.5

Algorithm 7 partitions the datagraph 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* (Line 4-7). Then we call Algorithm 16 (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 16), we get the complete $\mathcal{SP}$.

In Algorithm 16, for the vertex u, its neighbors $N(u, G)$ are classified into 3 categories. Line 2-5 include such $v \in N(u, G)$ with $c(v) < c(u)$ to SC. Line 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 16 for v, because $v \in SC.V$. (Line 6-13)

After choosing an anchor vertex in each iteration, we use Algorithm 9 to maintain the

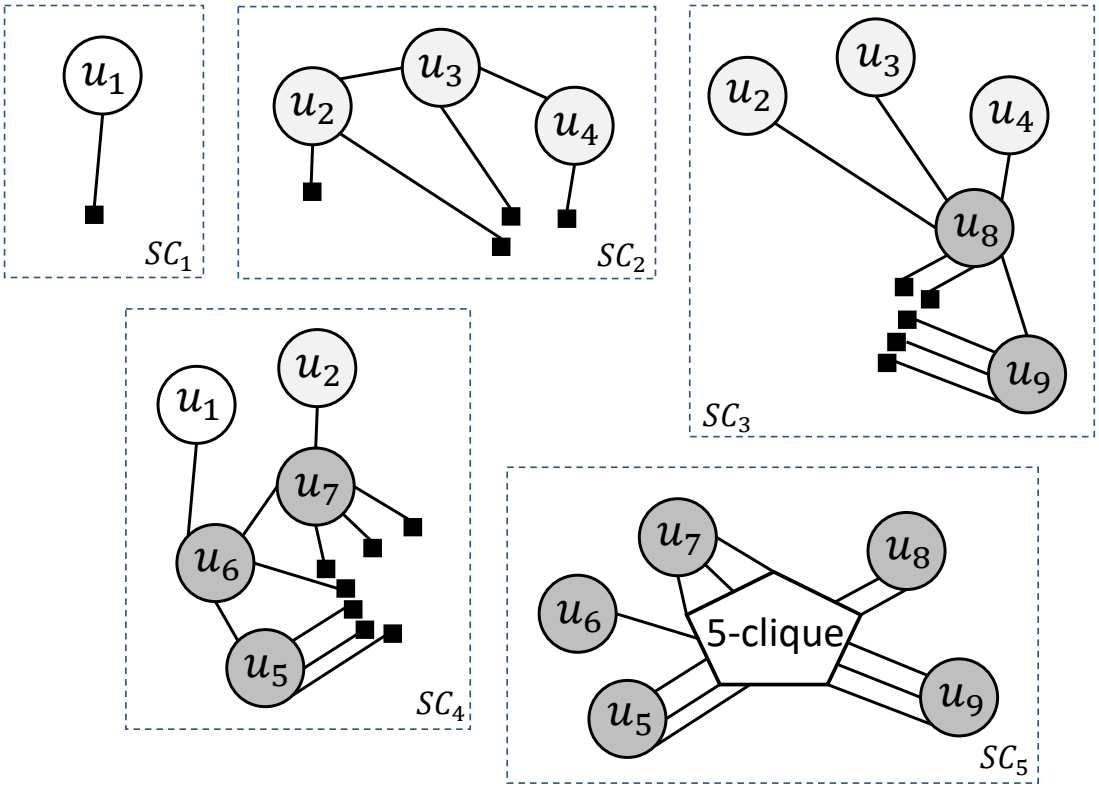


Figure 3.7: Shell Component Partition

partition for the next iteration. We prove the maintenance by Algorithm 9 is correct. We firstly set all the vertices as $unassigned$ (Line 1-2). For the anchoring of x, we define $V_x = (\bigcup_{SC \in \mathcal{SP}[x]} SC.V) \setminus \{x\}$. Then, in the post-anchor graph G_x with anchoring x, after updating the coreness of the anchor vertex and its followers (Line 3-5), any k-shell component S_k^i having $\exists v \in V_x$ s.t. $v \in V(S_k^i)$ and its affiliated shell component partition are updated by Line 6-13. The following lemmas and theorems prove the other shell component partitions remain the same.

Lemma 3.5.1 *For each non-anchor vertex $u \in V(G) \setminus A$, there is only one shell component partition SC having $u \in SC.V$.*

Proof 3.5.1 *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) $k = k' =$*

Algorithm 7: **ShellPartition**(G)

Input : G : the graph

Output : $\mathcal{SP}$: the shell component partition of G

```
1  CoreDecomp(G, ∅);
2  for each u ∈ V(G) in ascending c(u) order do
3      if u is unassigned then
4          SC ← an empty shell component partition;
5          SC.V := SC.V ∪ {u};
6          SC.h[u] := 0;
7          u is set assigned;
8          ShellConnect(u, G, SC);
9          SP[u] := SP[u] ∪ {SC};
10 return SP
```

$c(u)$; 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 $c(u, SC)$.

Lemma 3.5.2 *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 3.5.2 *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*

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

Input : u : a vertex, G : the graph, SC : the shell component partition containing u

```
1  for each v ∈ N(u, G) do
2      if c(v) < c(u) then
3          SC.E := SC.E ∪ {(u, v)};
4          SC.V⁻ := SC.V⁻ ∪ {v};
5          SP[v] := SP[v] ∪ {SC};
6      else if c(v) = c(u) then
7          SC.E := SC.E ∪ {(u, v)};
8          if v is unassigned then
9              SC.V := SC.V ∪ {v};
10             SC.h[v] := 0;
11             v is set assigned;
12             ShellConnect(v, G, SC);
13             SP[v] := SP[v] ∪ {SC};
14     else if c(v) > c(u) then
15         SC.h[u]++;
16     else
```

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

Proof 3.5.3 *Consider a non-anchor vertex $u \notin V_x$. According to Lemma 3.5.1, there exists one SC, in which $u \in SC.V$, and $x \notin SC.V \cup SC.V^-$. According to Lemma 3.5.2,*

$c(u, G) = c(u, SC)$. *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 3.5.2 *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 3.5.4 *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 3.4.1, 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 3.5.1,* $u \in V_x$, *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 neigbour 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. Line 6-13 ensure all the new created shell component partitions have been inserted into $\mathcal{SP}$. Theorem 3.5.2 proves part of the shell component partitions remain the same, and Line 14-17 collects all other expired shell component partitions. Then Line 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, Algorithm 7, Algorithm 16 and Algorithm 9 are all $\mathcal{O}(m)$, because Algorithm 7 and Algorithm 9 are both dominated by the subcall of Algorithm 16, and in Algorithm 16, each neighbor $v \in N(u, G)$ of each vertex u is accessed once.

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

Input : x : the anchor vertex, G : the graph

for each $u \in V(G)$ **do**
  u is set as $unassigned$;
$c(x) := +\infty$;
for each $u \in \mathcal{F}[x]$ **do**
  $c(u)$++;
for each $SC \in \mathcal{SP}[x]$ **do**
  for each $unassigned\ u \in SC.V$ with $u \neq x$ **do**
    $SC' \leftarrow$ an empty shell component partition;
    $SC'.V := SC'.V \cup \{u\}$;
    $SC'.h[u] := 0$;
    u is set $assigned$;
    ShellConnect(u, G, SC');
    $\mathcal{SP}[u] := \mathcal{SP}[u] \cup \{SC'\}$;
D is the set of expired SC;
for each $assigned$ node u and each $SC \in \mathcal{SP}[u]$ **do**
  if $u \in SC.V$ and SC is not new added **then**
    $D := D \cup \{SC\}$;
for each $SC \in D$ and each $u \in SC.V^- \cup SC.V$ **do**
  $\mathcal{SP}[u] := \mathcal{SP}[u] \setminus \{SC\}$;

3.5.2 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 3.5.3 *For each vertex* $u \in V(G)$, $|\mathcal{F}(u,G)| = \sum_{SC \in \mathcal{SP}[u]} |SF[u][SC]|$.

Proof 3.5.5 *Based on Theorem 3.5.1, for each* $SC \notin \mathcal{SP}[u]$, SC *does not have any follower of* u. *Based on Lemma 3.5.1, 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 3.5.2,* $SF[u][SC]$ *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 3.5.3, 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 Section 3.5.3. 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* (Section 3.4.4) 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), Line 3-15 can compute the followers. Similar to the *Degree Check* in Section 3.4.4, we develop *Degree Check in Partition*, and Theorem 3.5.4 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 3.5.4 *A vertex $u \in SC.V$ cannot be a follower if $d_P^+(u, SC) < c(u, G) + 1$.*

Proof 3.5.6 *According to the definitions of partition degree bound and the degree bound in Section 3.4.4, for a vertex $u \in SC.V$, $d_P^+(u, SC) = d^+(u, G)$. And based on the proof of Theorem 3.4.5, this theorem also holds.*

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.

Algorithm 10: **FindFollowers(**SC**)**

Input : SC : the shell component partition

Output : $SF[\cdot][SC]$: the computed followers in SC

```
1  Get N^≤(u, SC) and N^>(u, SC) for each u ∈ SC.V;
2  for each x ∈ SC.C by parallel multi-threads do
3      H := ∅;
4      H.push(x);
5      while H ≠ ∅ do
6          u ← H.pop();
7          if d_P^+(u, SC) ≥ c(u, G) + 1 or u = x then
8              u is set survived;
9              for each v ∈ N^>(u, SC) and v ∉ 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] ∪ {v};
16 return SF[·][SC]
```

3.5.3 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 Section 3.4.5 to *upper bound in partition*, 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 Section 3.4.5. Based on Theorem 3.4.6, 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 Line 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 section, 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 j-th task $\mathcal{S}(j)$ is distributed to $Slave_1$ satisfying $\sum_{i=1}^{j} UB_{SC_i}(u_i) \leq C_{avg}$ 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.

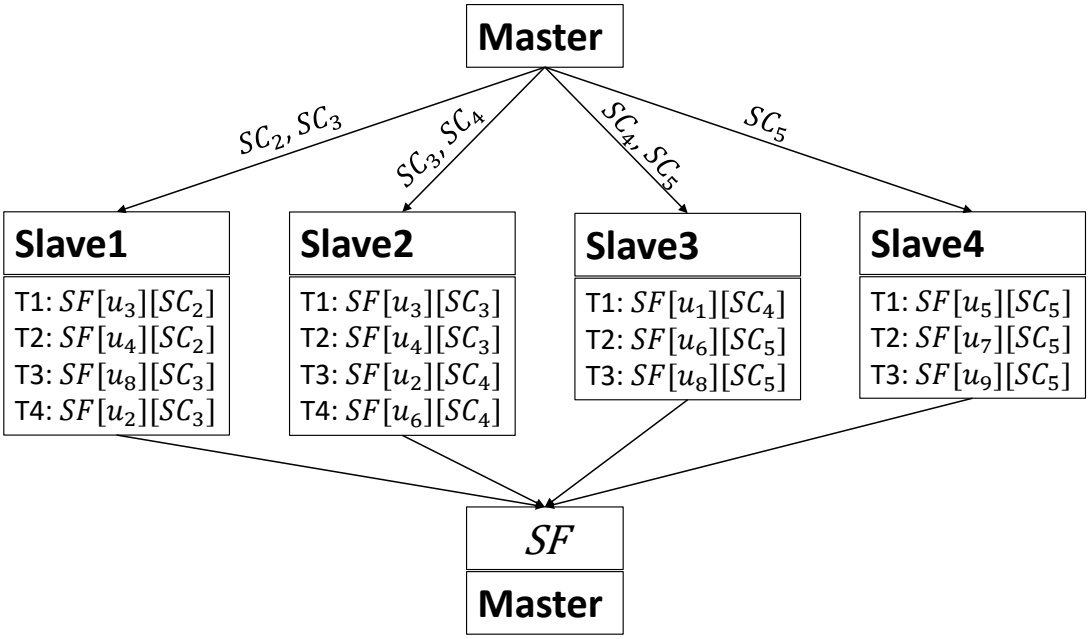


Figure 3.8: Computing Schedule

Example 3.5.3 *Figure 3.8 gives a scheduling example in the 5-machine distributed environment and the example is about the graph G in Figure 3.6. Firstly, the Master machine decomposes G into the five shell component partitions as in Figure 3.7. 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 $Slave1$, 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.

3.5.4 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 Section 3.5.1-3.5.3. 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 $u \in V(G) \setminus A$ as Equation 3.4. 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 Equation 3.5. Theorem 3.5.5 shows how the lower bound manages to prune some anchor candidates in each iteration.

$$LB(u) = \sum_{SC \in \mathcal{SP}[u] \wedge SF[u][SC] \neq \emptyset} |SF[u][SC]| \tag{3.4}$$

$$\mathcal{LB} = max\{LB(u) \mid u \in V(G) \setminus A\} \tag{3.5}$$

Theorem 3.5.5 *Given $\mathcal{LB}$ in an iteration, for a non-anchor vertex $u \in V(G) \setminus A$, $UB_\sigma(u) = \sum_{SC \in \mathcal{SP}[u]}(UB_{SC}(u))$. If $UB_\sigma(u) < \mathcal{LB}$, u is not the best anchor of the iteration.*

Algorithm 11: **PruneCandidates**(G)

Input : G : a social network with all the above definitions as global variables

```
1  Compute UB_SC(·) only for new added SC;
2  LB := 0;
3  for each u ∈ V(G) \ A do
4      UB_σ(u) := 0; LB(u) := 0;
5      for each SC ∈ SP[u] do
6          if SF[u][SC] ≠ ∅ then
7              UB_σ(u) := UB_σ(u) + |SF[u][SC]|;
8              LB(u) := LB(u) + |SF[u][SC]|;
9          else
10             UB_σ(u) := UB_σ(u) + UB_SC(u);
11     if LB(u) > LB then
12         LB := LB(u);
13 for each u ∈ V(G) \ A do
14     if UB_σ(u) > LB then
15         for each SC ∈ SP[u] with SF[u][SC] = ∅ do
16             SC.C := SC.C ∪ {u};
```

Proof 3.5.7 *According to Equation 3.5, there exists a u' with $LB(u') = \mathcal{LB}$. We have $|\mathcal{F}(u')| \geq LB(u')$ from Equation 3.4 and have $|\mathcal{F}(u)| \leq UB_\sigma(u)$ 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

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 λ := −1; a := null;
2 for each u ∈ V(G) \ A do
3     if ∃SC ∈ SP[u] with SF[u][SC] = ∅ then
4         Continue;
5     else
6         F(u, G) := ⋃_{SC∈SP[u]} SF[u][SC];
7         if |F(u, G)| > λ then
8             a := u; λ := |F(u, G)|;
9 return a
```

bound in partition of the vertices in the new added shell component partitions (Line 1). In Line 3-10, we compute the upper bound $UB_\sigma(u)$ of the total number of followers for each $u \in V(G) \setminus A$. 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, Line 11-12) is to record the maximum $LB(u)$ among $u \in V(G) \setminus A$. By $\mathcal{LB}$, we can prune such vertex u with $UB_\sigma(u) \leq \mathcal{LB}$ (Line 13-16). Only such 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 $u \in V(G) \setminus A$, 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

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 (Section 3.5.3) [*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

the anchor in the current iteration. We traverse all the non-anchor vertices (Line 2) and keep updating the best anchor so far (Line 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.

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 $SC.C$.

3.6 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 bold capital first letters when necessary. Table 4.5 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 1Gbps network. They all have 3.4GHz 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. Towards effectiveness, we mainly compare 6 algorithms with our `GAC` algorithm, including 4 heuristics, the exact solution, and the algorithm for anchored k-core problem. Towards the efficiency of our serial algorithm `GAC`, we incrementally equip the baseline with our proposed techniques to evaluate the performance. Towards the efficiency of our distributed algorithm `DGAC`, we compare its time cost (1 master + 8 slaves each with 8 threads as default) to `GAC`, 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 3.7 lists all the evaluated

Table 3.6: 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	10721	51
N.	325,729	1,497,134	6.5	3812	155
S.	281,903	2,312,497	16.4	38626	71
Y.	1,134,890	2,987,624	5.3	28754	51
D.	1,566,919	6,461,300	8.3	2023	118
L.	3,997,962	34,681,189	17.4	14815	360

algorithms.

3.6.1 Effectiveness

Comparison with Other Heuristics. In Figure 3.9, we compare the coreness gain from GAC with other heuristics (Rand, Deg, Deg−C, and SD). The motivation of the basic methods are as follows. For Deg, a vertex with larger degree means the anchoring of it can potentially influence more vertices through its large number of neighbour 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 $deg_{\succ}(u) = |\{v \mid v \in N(u,G) \ \& \ \mathcal{P}(v) \succ \mathcal{P}(u)\}|$ where $\mathcal{P}(u) = (k,i)$ means u is in the i-th 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 3.7 shows the details.

As shown in Figure 3.9 (a), the performance of Rand is the worst as it chooses random vertices to anchor. The performance of Deg and Deg−C are better than Rand as they choose vertices with large degrees to anchor. SD has more coreness gain because the vertices with higher successive degree have more candidate followers (Theorem 3.4.4).

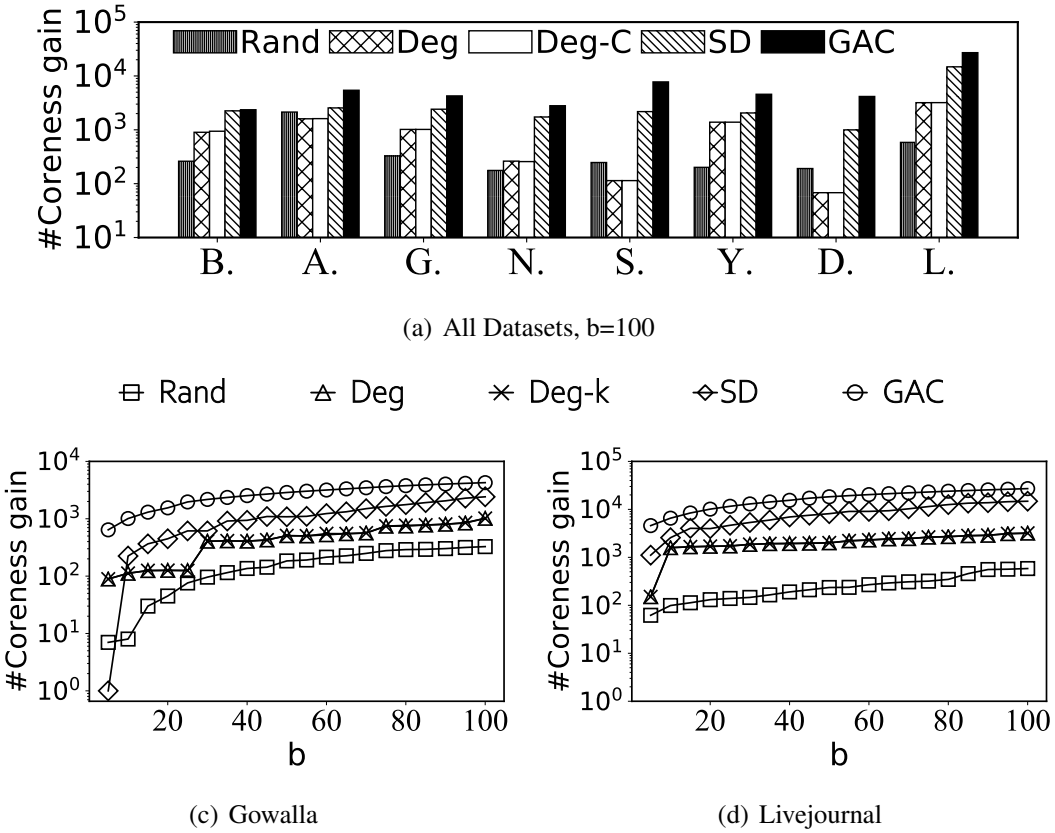
Table 3.7: Summary of Algorithms

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 [102]
GAC	Algorithm 6
DGAC	Algorithm 13
DGAC(-C)	DGAC without the time of data communication between machines.
GAC-U	GAC without upper bound pruning (Section 3.4.5)
GAC-U-R	GAC-U without result reusing (Algorithm 3)
Baseline	GAC-U-R using core decomposition (Algorithm 14) to compute coreness gain, without Algorithm 4

Compared with the above heuristics, GAC achieves the much larger coreness gains on all the datasets. The effect of varying b is shown in Figures 3.9 (c) and (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 neighbours, until the number of extracted vertices reaches 100. For both Brighkite and Arxiv, we extracted 10 such subgraphs and report the average coreness gain of them. The runtimes are also reported. Figure 3.10 shows that the coreness gain of GAC is always at least 70% of Exact, and GAC is faster than Exact by up to 5 orders of magnitude. Note that the coreness gain percentage of GAC over Exact may increase with larger b values, e.g., from $b = 4$ to $b = 5$.

Characteristics of Anchor Set. Table 3.8 shows the average degree of anchors (Deg_{anc})



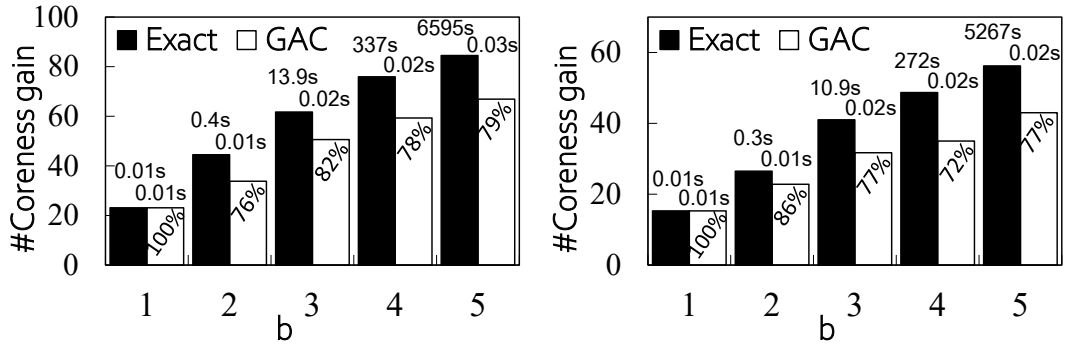
(a) All Datasets, b=100

(c) Gowalla

(d) Livejournal

Figure 3.9: Coreness Gain from Different Heuristics

from GAC is much larger than the average degree of all the vertices in the graph (Deg_{avg}). Then, we investigate the average ranking of an anchor in all the vertices regarding degree, coreness, and successive degree, denoted by p_{Deg}, p_{CN}, and p_{SD}, respectively. According to Theorem 3.4.4, 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 $\mathcal{O}^x_{Deg}$, $\mathcal{O}^x_{CN}$ and $\mathcal{O}^x_{SD}$, in ascending order of degree, coreness and successive degree, respectively. Then $p_{Deg} = \frac{\sum_{x \in A} \mathcal{O}^x_{Deg}}{|A||V(G)|}$, $p_{CN} = \frac{\sum_{x \in A} \mathcal{O}^x_{CN}}{|A||V(G)|}$ and $p_{SD} = \frac{\sum_{x \in A} \mathcal{O}^x_{SD}}{|A||V(G)|}$. Table 3.8 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



(a) Brightkite (b) Arxiv

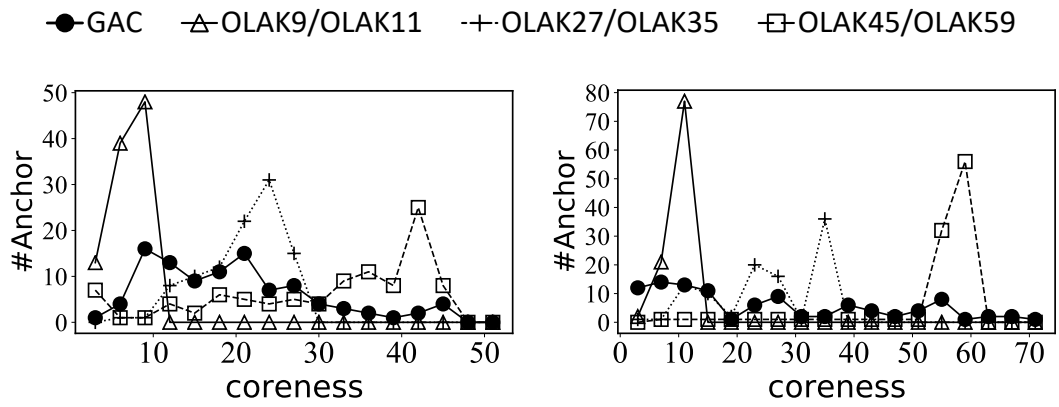
Figure 3.10: GAC v.s. Exact

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

Table 3.8: Characteristics of Anchor Set

with large successive degree are not effective anchors, as shown by SD in Figure 3.9. Moreover, Figure 3.11 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.

Analysis of Top-b Solutions In one iteration of the GAC algorithm, when there are more than one best anchor, all of which have the same largest coreness gain, we break the ties by the follower upper bound of the candidate anchors (Section 3.4.5). For clearness, we denote GAC by GAC−UB. Besides, we may use other criteria to break the ties in the greedy algorithm: choosing the vertex with the largest degree (denoted by GAC−DG), or randomly choosing a vertex (denoted by GAC−RD). As shown in Table 3.9, the coreness gains of different solutions (anchor sets) are very similar, where the values are denoted



(b) Gowalla (c) Stanford

Figure 3.11: Distribution of Anchors on Coreness

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

Table 3.9: Statistics of Top-b Solutions

by $Gain_{UB}$, $Gain_{DG}$ and $Gain_{RD}$ accordingly, and the largest value for each dataset is marked in bold. Moreover, as shown in Table 3.9, 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.

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 3.10: Coreness Gain, OLAK v.s. GAC

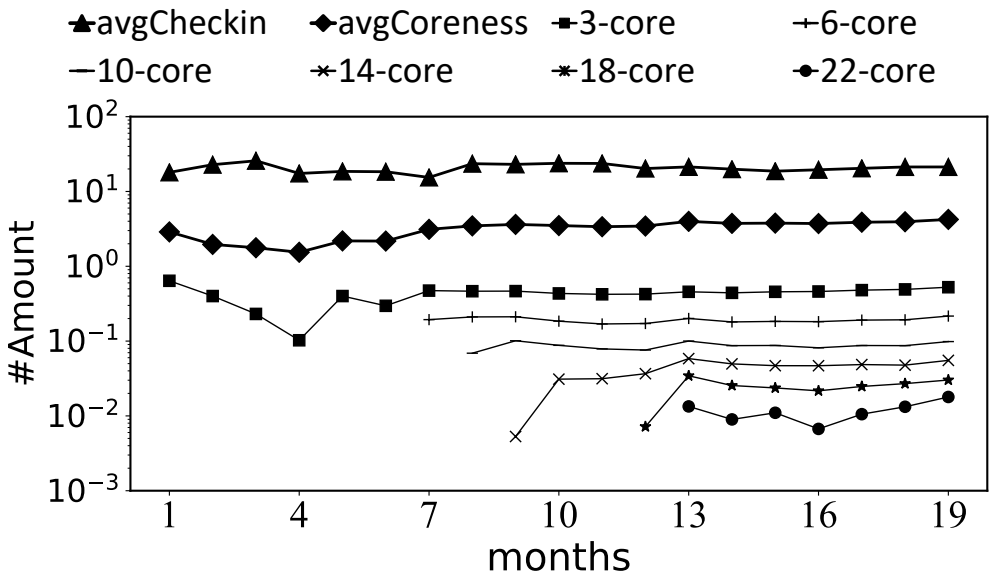


Figure 3.12: #Checkin, Coreness & k-Core Size

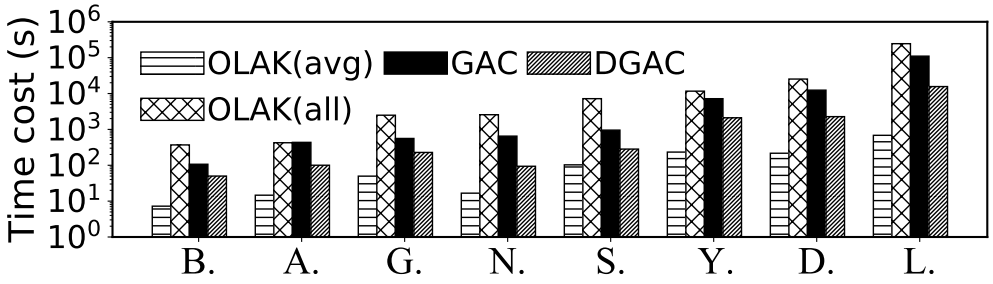
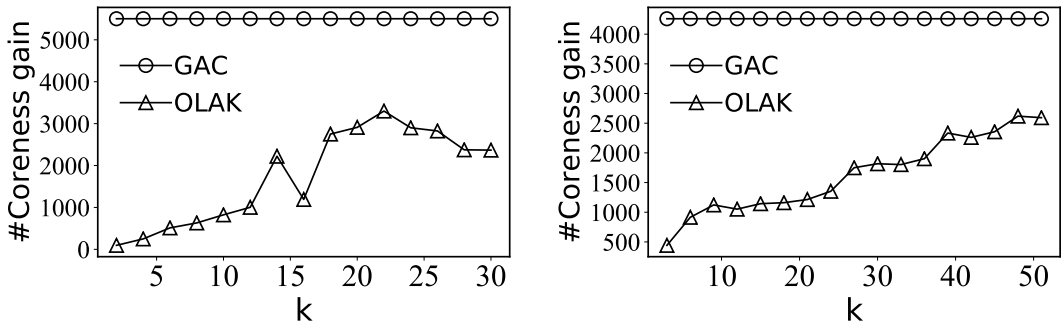


Figure 3.13: Time Cost, OLAK, GAC & DGAC

Correlation with #Checkin We generate 19 different networks from Gowalla based on the user check-ins, where the i-th network is the induced subgraph by the users with at least 1 check-in during the $(i + 1)$-th month, except for the first and the last months where the data is incomplete. We consider the number of user check-ins because a user with more friends may be more active in `Gowalla` network.

For each network, we divide the sum of #checkins, the sum of coreness, and the size of k-core, by the number of users, respectively. As shown in Figure 3.6.1, the



(a) Arxiv (b) Gowalla

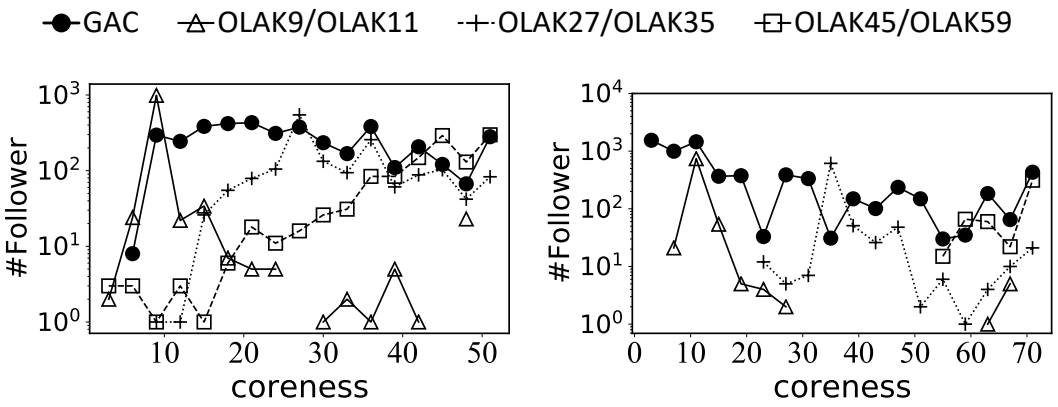
Figure 3.14: Coreness Gain on Different Inputs of k

Dataset	B.	A.	G.	N.	S.	Y.	D.	L.
$avg_{Jaccard}$	0.021	0.006	0.004	0.002	0.019	0.005	0.001	0.004
$max_{Jaccard}$	0.058	0.02	0.02	0.053	0.064	0.02	0.02	0.031

Table 3.11: Overlap of Followers Set, OLAK v.s. GAC

pattern of size proportions of k-cores are more fluctuated compared to the pattern of average #checkins and average coreness, especially for large k values. However, if we choose a small k for OLAK, it generally has small coreness gain as shown in Figure 3.14. The pattern of average coreness over the first 7 months in Figure 3.6.1 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 Figure 3.13, 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 Figure 3.13, 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 can-



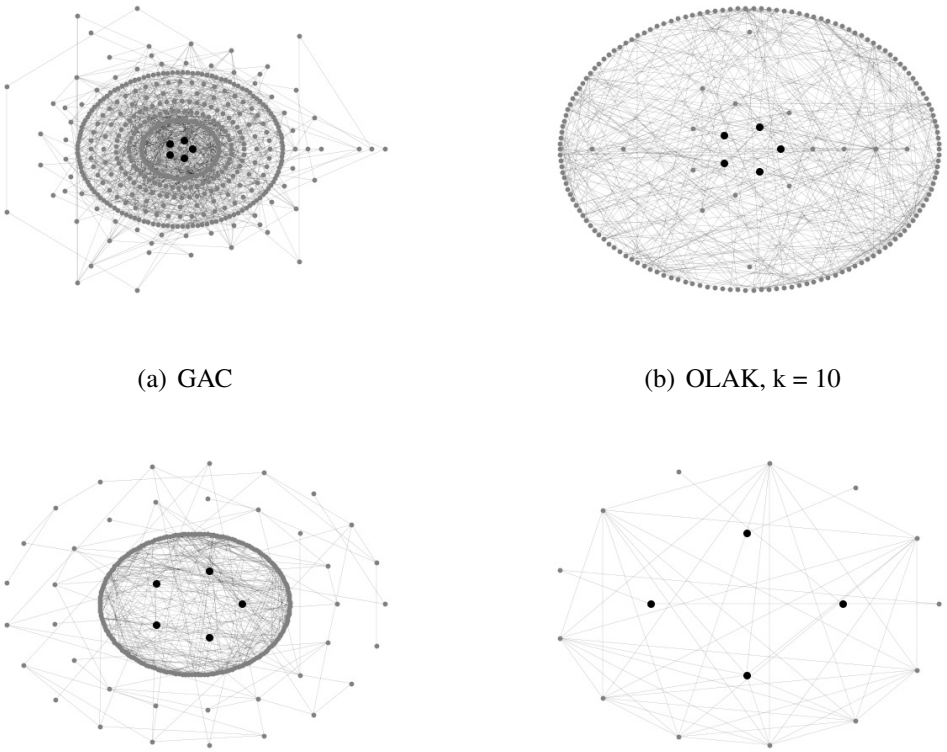
(b) Gowalla (c) Stanford

Figure 3.15: Distribution of Followers on Coreness

didate, 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 4.5, 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 3.10 is added to show that the largest coreness gain (denoted by max_{OLAK}) that OLAK can achieve only reaches 46%-77% of the coreness gain by GAC, on all the datasets. 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_{OLAK} and avg_{OLAK}) for different k values on each dataset. Table 3.10 also shows that avg_{OLAK} is only 4%-41% of the coreness gain of GAC. Besides, Figure 3.14 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.

Figure 3.11 shows the distribution of 100 anchors from GAC and OLAK on coreness value. Figure 3.15 shows the distribution of followers from GAC and OLAK on coreness

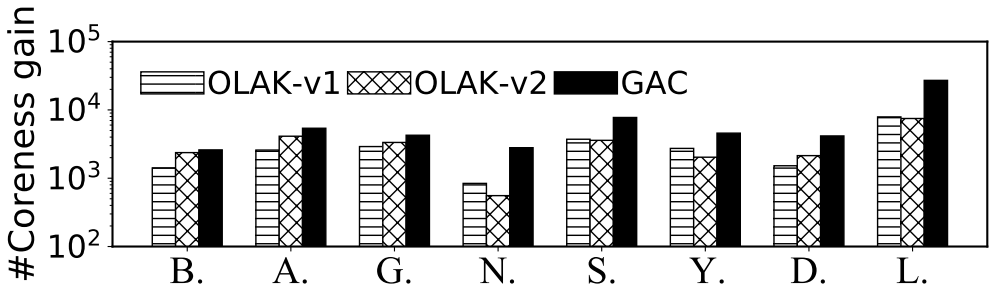


(a) GAC (b) OLAK, k = 10

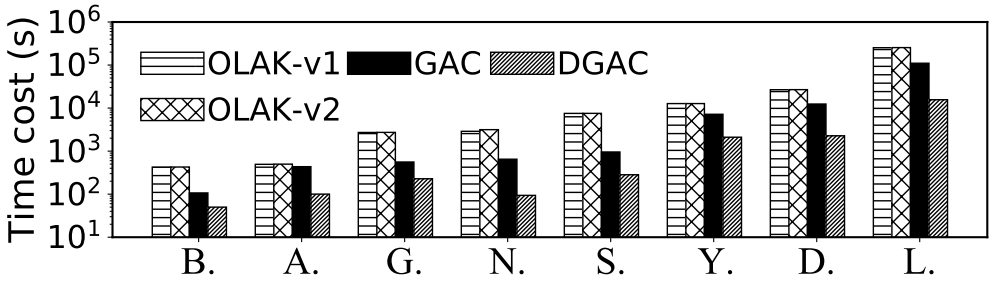
(c) OLAK, k = 20 (d) OLAK, k = 30

Figure 3.16: Case Study on DBLP, b = 5

value. For each coreness value x_i on the horizontal ordinate of Figure 3.11 (resp. Figure 3.15), 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 [102]. Besides, Figure 3.15 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 3.11. Specifically, for each



(a) Coreness Gain of variations of OLAK



(b) Time Cost of variations of OLAK

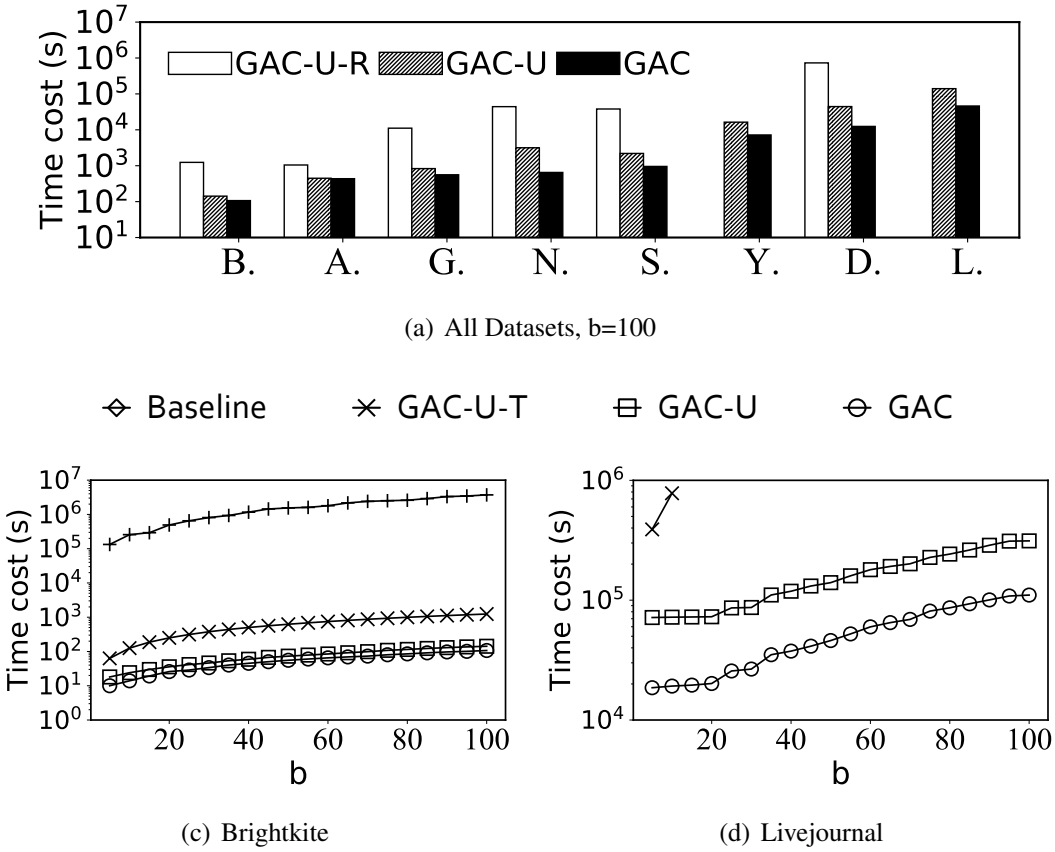
Figure 3.17: Variations of OLAK v.s. GAC & DGAC

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 $J^k = \frac{|A_{GAC} \cap A^k_{OLAK}|}{|A_{GAC} \cup A^k_{OLAK}|}$. Then, we compute $avg_{Jaccard}$ (resp. $max_{Jaccard}$) which is the average (resp. maximum) value of J^k of all the inputs k. From Table 3.11, we can find the $avg_{Jaccard}$ and $max_{Jaccard}$ 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 Figure 3.16. 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 visualise 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 5-th anchor has no follower, so there are 4 anchors in Figure 3.16 (d). 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 Figure 3.16, which means the followers of GAC are more diverse and this is consistent with the result of Figure 3.15.

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 3.17 (a) 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 3.17 (b) 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.



(a) All Datasets, b=100

(c) Brightkite

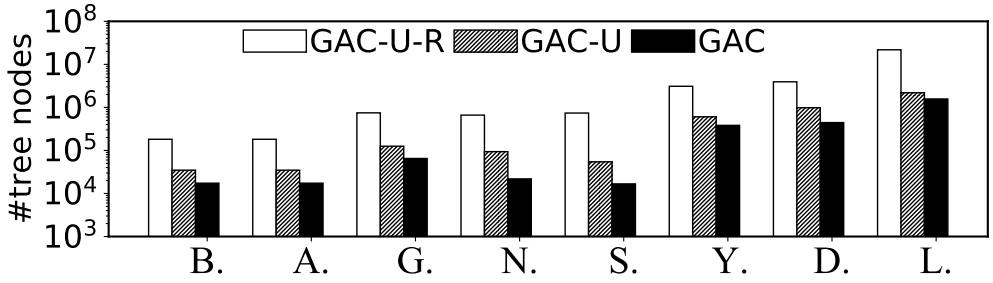
(d) Livejournal

Figure 3.18: Time Cost of Different Algorithms

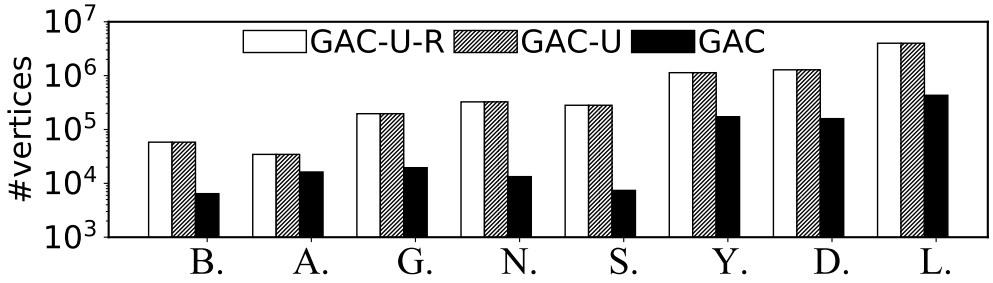
3.6.2 Efficiency of GAC

Overall Performance. Figure 3.18(a) shows the total running time of `GAC`, `GAC-U` and `GAC-U-R` on all the 8 datasets when $b = 100$. `GAC-U-R` does not return on `Youtube` and `Livejournal` after 10 days and thus the runtime is not reported. With our reusing mechanism (Algorithm 3), `GAC-U` is faster than `GAC-U-R` by 1 order of magnitude on average. Further benefitting from the upper bound based pruning (Section 3.4.5), the runtime of `GAC` is usually faster than `GAC-U` by more than 3 times. The details are as follows.

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



(a) visited tree nodes



(b) visited vertices

Figure 3.19: Visited Amount

magnitude on Brightkite, as shown in Figure 3.18(c). As it is very time-consuming to compute the coreness gain of candidate anchors using core decomposition, we can only report the runtime of Baseline on Brightkite.

Intermediate Result Reusing. By applying the core component tree (Section 3.4.1) and the result-reusing mechanism (Algorithm 3), GAC−U always outperforms GAC−U−R on runtime by at least 1 order of magnitude, as shown in Figure 3.18. 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 Figure 3.19(a), the number of visited tree nodes of GAC−U is around 10% of GAC−U−R.

Candidate Anchors Pruning. In Figure 3.18, we can see our final algorithm GAC achieves further speedup based on GAC−U when the upper bound pruning is equipped

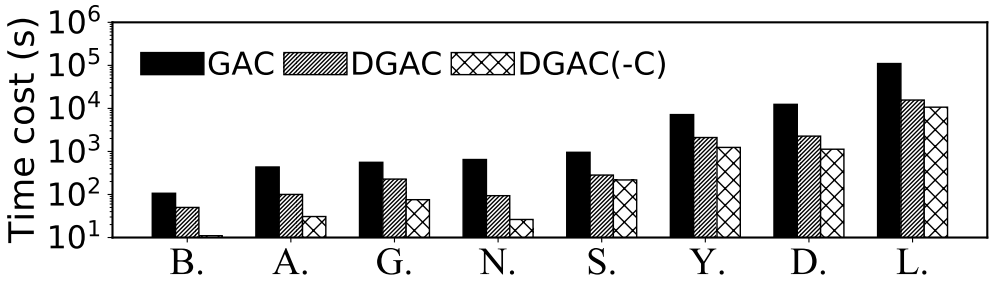
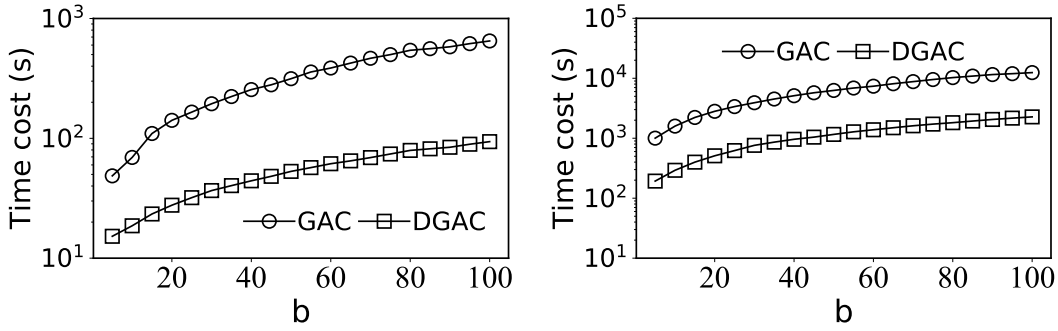


Figure 3.20: Time Cost, GAC, DGAC & ideal DGAC



(a) NotreDame (b) DBLP

Figure 3.21: Time Cost varing b, GAC v.s. DGAC

(Section 3.4.5). The processing time of GAC is only $20\% - 30\%$ of GAC−U because GAC reduces the search space by pruning the vertices with insufficient upper bounds of coreness gains. In Figure 3.19(a)-(b), the number of visited tree nodes and the number of visited vertices in GAC are much less than that in GAC−U.

3.6.3 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 Figure 3.20, we use DGAC(−C) to denote the

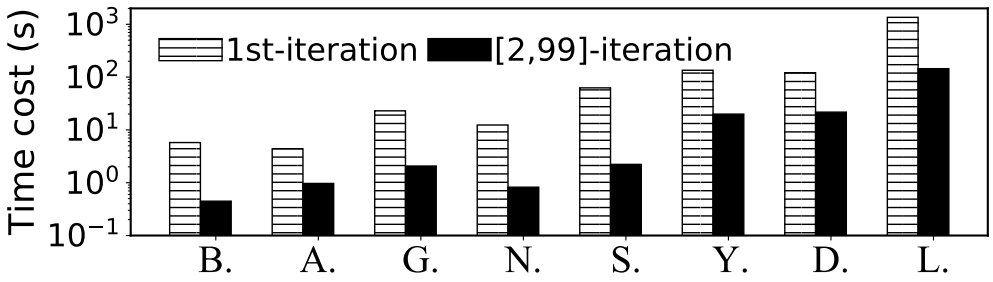
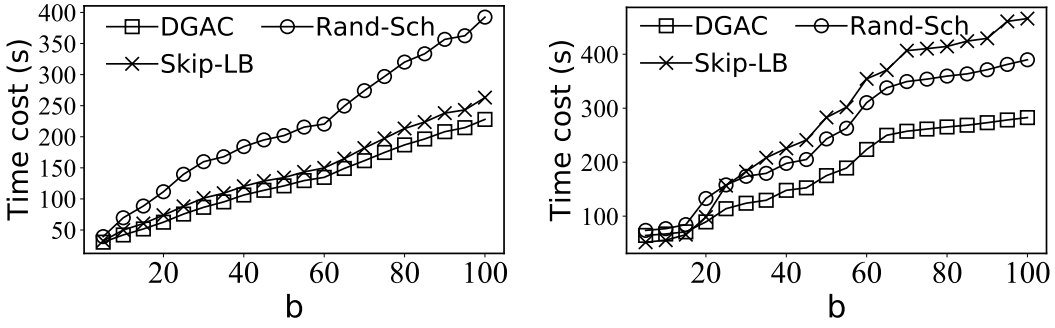


Figure 3.22: Time Cost, 1st-iteration v.s. Average of [2, 99]-iteration



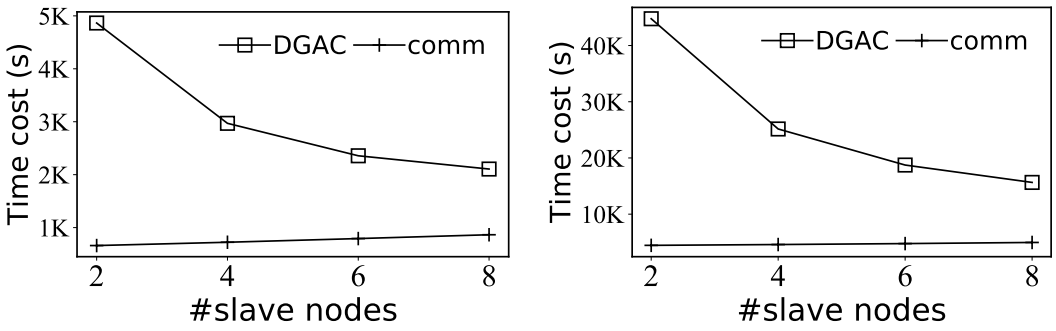
(a) Gowalla (b) Stanford

Figure 3.23: Time Cost of DGAC (skipping individual components)

total running time minus communication time of DGAC, to also show the 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 3.21(a)-(b) show 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 (Section 3.5.2), computing resource scheduling (Section 3.5.3) and lower bound based pruning (Section 3.5.4). 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 1st iteration and the average time cost of 2-99 iterations in the greedy algorithm. The results are shown in Figure 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 Figure 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 Figure 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



(a) Youtube (b) Livejournal

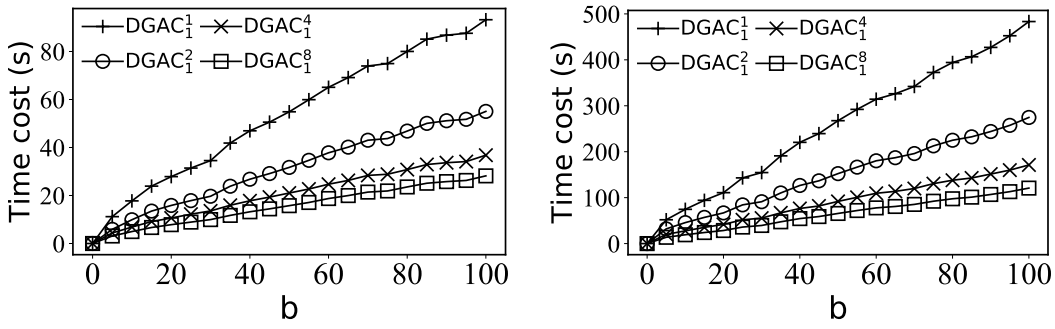
Figure 3.24: Time Cost, Varying the Number of Machines (8 threads)

strategy and lower bound pruning are effective for accelerating DGAC.

3.6.4 Scalability of DGAC

Varying the Number of Machines. We show the scalability of DGAC, varying the number of slave machines. In Figure 3.24, 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 (Section 3.5.3) as the estimation of computational amount of finding followers. The time cost of data communication (comm in Figure 3.24(a) - (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 Figure 3.25. As Section 3.5.4 illustrates, our DGAC algorithm is easily adpated to a parallel algorithm on one machine with multithreads, in which we simply treat each thread as one slave machine having only one



(a) Brightkite (b) Arxiv

Figure 3.25: Time Cost, Varying the Number of Threads (1 machine)

thread in DGAC to share the followers computing tasks. DGAC$_1^1$, DGAC$_1^2$, DGAC$_1^4$ and DGAC$_1^8$ are conducted on 1 single machine with 1, 2, 4 and 8 threads respectively. Figure 3.25 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 DGAC$_1^1$ and 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.

3.7 Chapter Conclusion

In this chapter, 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 [69]. 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.

Chapter 4

User Influence Monitoring for Network Stability

4.1 Introduction

The structural stability of a network indicates the ability of the network to maintain a sustainable service and/or to defend the attacks from the competitors. In the study of network stability, it is essential to monitor the engagement of the nodes in a network, and then motivate or protect the critical nodes [69, 74]. For instance, Friendster was a once popular social network with over 115 million users, but it is suspended due to contagious leave of users with low engagement [42] [87].

Complex networks are often modeled as graphs when studied. In graph theory, the k-core is defined as the maximal subgraph where each vertex has at least k neighbors (degree) in the subgraph [76] [85]. Given a graph, the k-core decomposition iteratively removes 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. Existing works have well studied the effect of coreness on capturing node engagement. In [74], the

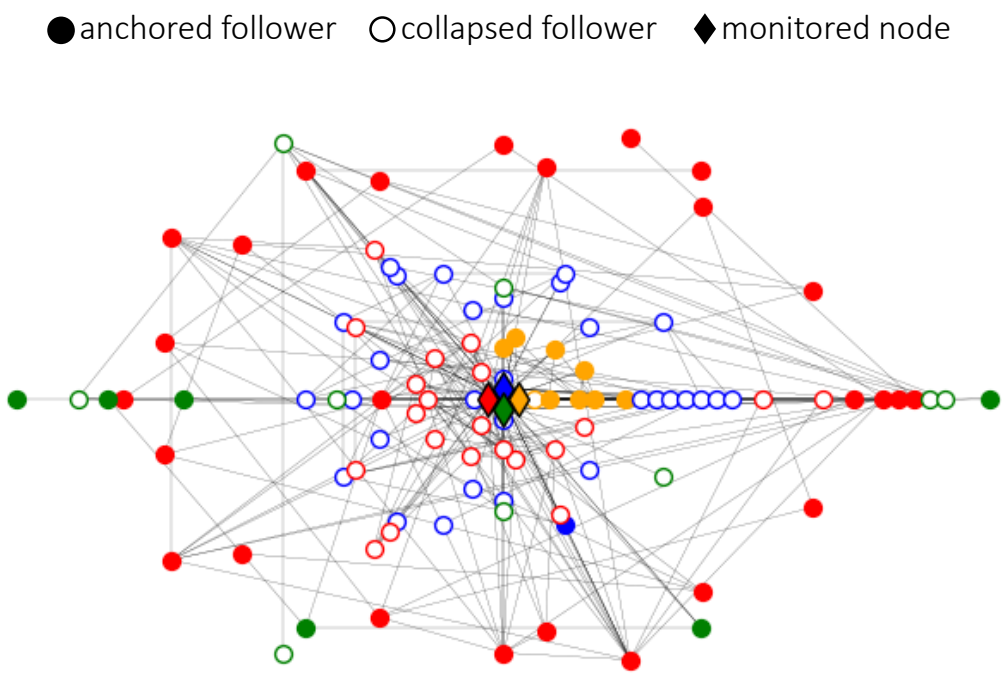


Figure 4.1: Node Monitoring on `Gowalla`

coreness of a node is demonstrated as the "best practice" for its engagement. In [69], the engagement of a node (e.g., the check-in number of a node) is further validated as positive correlated with its coreness based on real-life experiments.

The weakening and strengthening of nodes are the two natural engagement dynamics in a network. Specifically, in [103] (resp. [11]), when weakening (resp. strengthening) a node, we say it is *collapsed* (resp. *anchored*) such that its degree is regarded as 0 (resp. $+\infty$). When a node x is collapsed (resp. anchored), all the other nodes which decrease (resp. increase) the coreness values due to collapsing (resp. anchoring) x are called x's *collapsed followers* (resp. *anchored followers*). As far as we know, against the weakening and strengthening of a node, the engagement dynamic change of other nodes has not been systematically studied. Thus, in this chapter, we compute the collapsed and anchored followers of each node to monitor its influence on the engagement dynamics of other nodes.

Example 4.1.1 *In the social network* `Gowalla` *[63], we monitor 4 nodes, the red, orange, blue and green diamonds at the center of Figure 4.1. The collapsed followers (resp. anchored followers) of a monitored node are visualized by the hollow circles (resp. solid circles) with the same color. All the followers having the same coreness are presented at the same distance from the center. We can find the following phenomena: 1) Different nodes have very different number of followers, e.g., the orange node has the least followers compared with the other 3 nodes; 2) The ratio of followers of the two categories varies from different monitored nodes, e.g., the vast majority followers of the blue (resp. orange) node are collapsed followers (resp. anchored followers), but the green and red nodes have relatively even number of collapsed and anchored followers; 3) The coreness of the followers are diverse, e.g., the followers are presented in many different distances from the center even though we only monitor 4 nodes.*

Based on Example 4.1.1, the applications of the new problem studied in this chapter can be summarized into two aspects: 1) the monitored node, and 2) the follower nodes. For 1), it is important to know how each monitored node can influence the network structural stability, which is reflected by the collapsed power and anchored power. For instance, the influence of the orange node is much less than the red node, in terms of both collapsed and anchored powers. Specifically, the orange node has only a few anchored followers and few collapsed followers. However, the engagement strengthening (resp. weakening) of the red node can influence its large number of anchored (resp. collapsed) followers. These anchored (resp. collapsed) followers originally have diverse engagement levels as they have diverse coreness values, and they are likely to increase (resp. decrease) their engagement levels. For 2), it is also important to know the identities of those specific follower nodes which change their engagement levels due to the collapsing and anchoring of a monitored node. For instance, in Example 4.1.1, when the social network holder finds the blue node has collapsed, as it has plenty of collapsed followers,

these follower nodes need to be paid attention because they are in the risk of engagement decrease.

In this chapter, we study the integration and the efficient computation towards anchored and collapsed followers. It is obvious that the real-life social networks are always evolving, i.e., new signed-up user, new added friends of a user etc. After a network evolves, the change of vertices' collapsed followers and anchored followers can be drastic which will be shown in our experimental study. Thus, we also aim to efficiently maintain the correct collapsed and anchored followers for each node against edge insertions and deletions, which helps us systematically monitor the node influence on network structural stability.

Challenges. To the best of our knowledge, no existing work studies each node's influence for network structural stability. Existing works only find the best b collapse [101] (resp. anchor [69]) vertices to minimize (resp. maximize) the total coreness from all the vertices, without computing the collapsed (resp. anchored) follower sets of every single vertex. The only common concept is the definition of collapsing (resp. anchoring) a vertex, but the computation objective is completely different. Our problem becomes more challenging when maintaining the follower sets when the network evolves.

The naive core decomposition [10] is a straightforward method to compute the collapsed and anchored follower sets for each vertex, simply by regarding the degree of each vertex as 0 and $+\infty$. However, it is certainly cost-prohibitive due to the massive search space, i.e., when computing the collapsed or anchored follower set of each vertex, the core decomposition algorithm traverses all the edges in the graph again, which makes the time complexity be $\mathcal{O}(n \cdot m)$.

Core maintenance [105] is only to update the coreness value of each vertex itself, after an edge is inserted into or removed from the graph. However, it cannot compute or update the collapsed and anchored follower sets of each vertex. The computation

of the collapsed and anchored follower sets is a non-trivial process based on the coreness of each vertex, and the maintenance of the two follower sets becomes even more challenging against the network evolving.

Our Solution. Because of the novelty on the first study and the challenge on designing the algorithm given no existing approach, we propose our new solution.

Firstly, we propose an offline algorithm to efficiently compute the collapsed and anchored follower sets of each vertex in parallel. Specifically, the graph is divided into multiple *shell components* induced by all the maximal connected subgraphs where each vertex has the same coreness. Therefore, for either collapse or anchor a vertex x, the follower sets are formed by some vertex subsets from the shell components. We prove those vertex subsets are not overlapped and can be independently computed within each shell component, so that any parallel architecture can be utilized to concurrently compute the collapsed and anchored follower sets.

Secondly, we propose an online algorithm to efficiently update the collapsed and anchored follower sets of each vertex. When an edge is inserted into or removed from the graph, we first adopt the state-of-the-art core maintenance algorithm 'k-order' [105] to correctly update each vertex's coreness. Then some new induced shell components are efficiently collected, and the collapsed and anchored follower sets of any vertex can be updated only based on the new shell components. Because the scale of new shell components against one inserted or removed edge is constant, our online maintenance algorithm can achieve 3 orders of magnitude faster than redoing the offline algorithm.

Contributions. We summarize the contributions of this chapter as follows.

- We propose a new model that first time studies the collapsed power and anchored power to reflect the node influence on network structural stability by two symmetrical perspectives, i.e., one for node strengthening and the other for node weakening, which are corresponding to decreasing and increasing other nodes' engage-

ments respectively.

- We integrate the computation of collapsed power and anchored power of each node by a novel concept, i.e., shell component, by which any parallel architecture can be utilized to efficiently offline compute the two powers of each node.

- An online maintenance algorithm is proposed to efficiently update the collapsed power and anchored power of each node, when an edge is inserted into or removed from the network. This makes our model more applicable to real-life scenarios.

- Experiments conducted on 8 real-life datasets demonstrate that both our offline computation and online maintenance algorithms are efficient, and our new proposed model is effective.

4.2 Preliminaries and Problem Statement

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 neighbour vertex set of u in G. Note that we may omit the input graph for all the notations in the chapter when the context is clear, e.g., using $deg(u)$ instead of $deg(u, G)$.

Definition 4.2.1 *k-core. 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.

Table 4.1: Common Notations throughout the Chapter

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, w, x	a vertex in G
$N(u, G)$	the set of neighbors of u in G
$deg(u, G)$	$+\infty$ if u is anchored, 0 if u is collapsed, $\|N(u, G)\|$ otherwise
$C_k(G)$	the k-core of G
$c(u, G)$	the original coreness of u in G
$c_x^-(u, G)$	the coreness of u in G with collapsing x
$c_x^+(u, G)$	the coreness of u in G with anchoring x
${}^-\mathcal{F}(x, G)$	the collapsed follower set of x in G
${}^+\mathcal{F}(x, G)$	the anchored follower set of x in G
$H_k(G)$	the k-shell of G
$\mathcal{SC}[v]$	the only shell component containing v
S, $S.V$, $S.E$, $S.c$	a shell component with its vertex set, edge set and coreness value
$\mathcal{C}[S]$	the collapser candidate set of S
$\mathcal{A}[S]$	the anchor candidate set of S
${}^-F[x][S]$	the collapsed follower set of x in S
${}^+F[x][S]$	the anchored follower set of x in S

Definition 4.2.2 ***coreness***. *Given a graph G, the coreness of a vertex $u \in V(G)$, denoted by $c(u, G)$, is the largest k such that $C_k(G)$ contains u, i.e., $c(u, G) = max\{k : u \in C_k(G)\}$.*

Definition 4.2.3 ***core decomposition***. *Given a graph G, core decomposition of G is to compute the coreness of every vertex in $V(G)$.*

In this chapter, once a vertex x in the graph G is **collapsed** (resp. **anchored**), the degree of x is regarded as zero (resp. positive infinity), i,e., $deg(x, G) = 0$ (resp. $deg(x, G) = +\infty$). Every collapsed (resp. anchored) vertex is called an **collapser** or **collapser vertex** (resp. **anchor** or **anchor vertex**). The existence of collapser vertices (resp. anchor vertices) may change the coreness of other vertices. We use $c_x^-(u, G)$ (resp. $c_x^+(u, G)$) to denote the coreness of u in G with collapsing (resp. anchoring) x.

The computation of core decomposition is shown in Algorithm 14, in which we re-

Algorithm 14: **CoreDecomp**(G)

Input : a graph G

Output : $c(u, G)$ for each $u \in V(G)$

```
k ← 1;
while exist vertices in G do
    while ∃u ∈ V(G) with deg(u) < k do
        deg(v) ← deg(v) − 1 for each v ∈ N(u, G);
        remove u and its adjacent edges from G;
        c(u, G) ← k − 1;
    k ← k + 1;
return c(u, G) for each u ∈ V(G)
```

cursively delete the vertex with the smallest degree in the graph G. The time complexity is $\mathcal{O}(m)$. Note that, the existence of collapser vertices and anchor vertices does not change the adjacent vertices of each vertex in the graph, but it induces subtle difference when conducting core decomposition. When there exists a collapser vertex x, x is automatically deleted at first as its degree is zero. When there exists an anchor vertex x, we do not delete x as its degree is positive infinity.

Definition 4.2.4 ***collapsed follower set***. *Given a graph G and the collapser vertex x, the collapsed follower set of x in G is denoted by ${}^{-}\mathcal{F}(x, G)$, and includes all other vertices decreasing their coreness with collapsing x in G, i.e.,* ${}^{-}\mathcal{F}(x, G) = \{u \in V(G) : u \neq x \wedge c_x^-(u, G) < c(u, G)\}$.

Definition 4.2.5 ***anchored follower set***. *Given a graph G and the anchor vertex x, the anchored follower set of x in G is denoted by ${}^{+}\mathcal{F}(x, G)$, and includes all other vertices increasing their coreness with anchoring x in G, i.e.,* ${}^{+}\mathcal{F}(x, G) = \{u \in V(G) : u \neq x \wedge c_x^+(u, G) > c(u, G)\}$.

Problem Statement. Given a graph G, for each $v \in V(G)$, we compute ${}^{-}\mathcal{F}(v, G)$ and ${}^{+}\mathcal{F}(v, G)$. When an edge (u, u') is inserted into (resp. removed from) G, we have $E(G) \leftarrow E(G) \cup \{(u, u')\}$ (resp. $E(G) \leftarrow E(G) \setminus \{(u, u')\}$). Now for each $v \in V(G)$, we maintain ${}^{-}\mathcal{F}(v, G)$ and ${}^{+}\mathcal{F}(v, G)$.

Note that, vertex insertion (resp. removal) can be regarded as a sequence of edges insertion (resp. removal).

4.3 The Offline and Online Algorithms

We introduce our algorithms of offline computing the collapsed follower set (`CFS`) and anchored follower set (`AFS`) for each vertex in a graph, and online maintaining the two sets each time when an edge is inserted to or removed from the graph. The computation is based on *shell component* (4.3.1), by which the graph is partitioned into multiple atom units. Any parallel architecture such as multi-threads can then be utilized to compute the `CFS` and `AFS` in parallel. When an edge is inserted or removed, 4.3.2 shows how we maintain the atom units. We prove that, by only conducting computation regarding those new updated atom units, the `CFS` and `AFS` for each vertex can be efficiently maintained. The further accelerating techniques for computing `CFS` and `AFS` are introduced in 4.3.3.

4.3.1 The Shell Component Based Framework

Definition 4.3.1 ***k*-shell.** *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., $H_k(G) = V(C_k(G)) \setminus V(C_{k+1}(G))$.*

Definition 4.3.2 ***shell component*.** *Given a graph G and the k-shell $H_k(G)$, a subgraph S is a shell component of $H_k(G)$, if S is a maximal induced connected component of $H_k(G)$.*

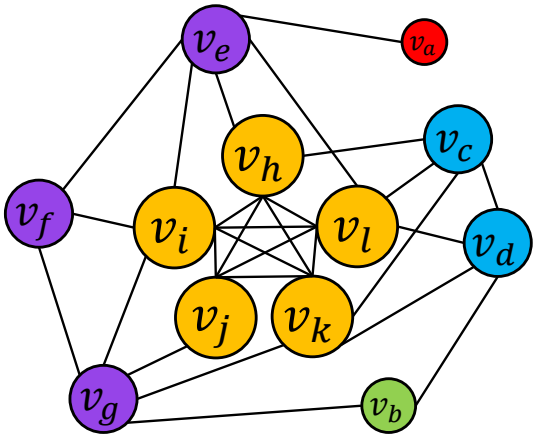


Figure 4.2: Illustration of Shell Component.

Theorem 4.3.1 *For each vertex $v \in V(G)$, there exists only one shell component S having $v \in V(S)$.*

Proof 4.3.1 *We prove it by contradiction. Assuming there are S_1 and S_2 with $v \in V(S_1)$ and $v \in V(S_2)$. Then we have, for each $u \in V(S_1)$, $c(u) = c(v)$ and u is connected with v; For each $u \in V(S_2)$, $c(u) = c(v)$ and u is connected with v. Thus, $S_1 \cup S_2$ satisfies the definition of one single shell component, which contradicts with our assumption. Proof completes.*

Example 4.3.1 *In Figure 4.2, we have $H_1(G) = \{v_a\}$, $H_2(G) = \{v_b\}$, $H_3(G) = \{v_c, v_d, v_e, v_f, v_g\}$ and $H_4(G) = \{v_h, v_i, v_j, v_k, v_l\}$. For $H_1(G)$, S_1 is the only shell component of $H_1(G)$ with $V(S_1) = \{v_a\}$. For $H_2(G)$, S_2 is the only shell component with $V(S_2) = \{v_b\}$. But within $H_3(G)$, we have two shell components S_3 and S_4, in which $V(S_3) = \{v_c, v_d\}$ and $V(S_4) = \{v_e, v_f, v_g\}$, because S_3 and S_4 are not connected by the edges among $H_3(G)$. For $H_4(G)$, S_5 is the only shell component with $V(S_5) = \{v_h, v_i, v_j, v_k, v_l\}$. The graph in Figure 4.2 will be often referred in the following examples, so we summarize the notations in Table 4.2 for your convenience.*

For a shell component S of $H_k(G)$, we denote $S.V$, $S.E$ and $S.c$ as the vertex set, edge set and the coreness of the vertices in S, i.e., $S.V = V(S)$, $S.E = E(S)$ and

Table 4.2: Shell Components in Figure 4.2

Shell component	S_1	S_2	S_3	S_4	S_5
Vertex set	v_a	v_b	v_c, v_d	v_e, v_f, v_g	v_h, v_i, v_j, v_k, v_l

Algorithm 15: **ShellDecomp**(G)

Input : G : the graph

Output : $\mathcal{SC}$: the index of shell components in G

1 **CoreDecomp**(G);
2 **for** each $unassigned$ $u \in V(G)$ **do**
3 $S \leftarrow$ a new shell component;
4 $S.c := c(u, G)$;
5 $S.V := S.V \cup \{u\}$;
6 u is set $assigned$;
7 **ShellConnect**(u, S, $\mathcal{SC}$);
8 $\mathcal{SC}[u] := S$;
9 **return** $\mathcal{SC}$

$S.c = c(v, G)$ $\forall v \in S.V$. We use the structure $\mathcal{SC}$ to index the shell components for all the vertices, in which for each $v \in V(G)$, $\mathcal{SC}[v]$ is the only shell component having $v \in \mathcal{SC}[v].V$ (Theorem 4.3.1). Algorithm 15 and Algorithm 16 illustrate the process of decomposing each vertex into its shell component.

In Algorithm 15, 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 each vertex marked $unassigned$ as default (Line 2). Each time meeting an *unassigned* vertex $u \in V(G)$, we create a new shell component S (Line 3), set the related domains of S and set u as *assigned* (Line 4-6). Then we call Algorithm 16 (details in the next paragraph) to recursively collect all the vertices which should be in S (Line 7), followed by assigning $\mathcal{SC}[u] := S$ (Line 8). When all the vertices are set $assigned$ (in Algorithm 15

Algorithm 16: **ShellConnect**(u, S, $\mathcal{SC}$)

Input : u : a vertex, S : the shell component containing u, $\mathcal{SC}$: the shell component index

```
1 for each v ∈ N(u, G) do
2     if c(v) = c(u) then
3         S.E := S.E ∪ {(u, v)};
4         if v is unassigned then
5             S.V := S.V ∪ {v};
6             v is set assigned;
7             ShellConnect(v, S, SC);
8             SC[v] := S;
```

or Algorithm 16), we get the complete $\mathcal{SC}$. The time complexity of Algorithm 15 is $\mathcal{O}(m)$ as we traverse each vertex's neighbors once.

In Algorithm 16, for the vertex u, we traverse all its neighbors in $N(u, G)$ (Line 1). If $c(v) = c(u)$ (Line 2), we add the edge (u, v) into $S.E$ (Line 3). Note that (u, v) and (v, u) are the same in our setting. Only if v is *unassigned* (Line 4), we add v into $S.V$ and recursively call Algorithm 16 to find all the vertices of S (Line 5-8).

$\mathcal{C}[\cdot]$ and $\mathcal{A}[\cdot]$. We define two affiliated structures, the collapser candidate set $\mathcal{C}[\cdot]$ and the anchor candidate set $\mathcal{A}[\cdot]$ w.r.t. all the shell components of G. For a shell component S, $\mathcal{C}[S] = \{v : v \in S.V \vee c(v, G) > S.c \wedge N(v, G) \cap S.V \neq \emptyset\}$, and $\mathcal{A}[S] = \{v : v \in S.V \vee c(v, G) < S.c \wedge N(v, G) \cap S.V \neq \emptyset\}$. $\mathcal{C}[\cdot]$ and $\mathcal{A}[\cdot]$ can be straightforward computed according to their definitions, when traversing each vertex's neighbors in Algorithm 15 or Algorithm 16, without changing the time complexity. We omit the specific process due to the limited space.

Lemma 4.3.1 *If a vertex x is collapsed in G, any other vertex $u \in V(G) \setminus \{x\}$ decreases*

its coreness by at most 1.

Proof 4.3.2 *Please refer to the proof of Theorem 4 in [101].*

Lemma 4.3.2 *If a vertex x is anchored in G, any other vertex $u \in V(G) \setminus \{x\}$ increases its coreness by at most 1.*

Proof 4.3.3 *Please refer to the proof of Theorem 4.6 in [69].*

Theorem 4.3.2 *For collapsing x in G, another vertex u is x's collapsed follower indicates $x \in \mathcal{C}[S]$ s.t. $u \in S.V$.*

Proof 4.3.4 *We prove it by contradiction. Assuming u decreases its coreness since x is collapsed, $u \in S.V$ and $x \notin \mathcal{C}[S]$. This firstly means $x \notin S.V$, according to the definition of $\mathcal{C}[S]$, then we have the following possible situations: 1) $c(x, G) < S.c$; 2) $c(x, G) = S.c$; 3) $c(x, G) > S.c \wedge N(x, G) \cap S.V = \emptyset$.*

For situation 1), No matter x is collapsed or not, x is always deleted before u in core decomposition (Algorithm 14), so that $c_x^-(u) = c(u)$ which contradicts with our assumption.

For situation 2), Consider deleting $V(G)$ in core decomposition without collapsing x. After all the vertices with coreness less than $S.c$ are deleted, for each $S' \neq S$ s.t. $S'.c = S.c$, it is available to delete $S'.V$ before $S.V$. And $S.V$ can remain in the graph still satisfying for each $v \in S.V$, $deg(v) \geq S.c$. We denote the deleted (resp. remained) vertex set so far as V^d (resp. V^r). For each $v \in V^r \setminus S.V$, $c(v) > S.c$. Because $c(x) = S.c$ and $x \notin S.V$, we can conclude $x \in V^d$. For the graph with collapsing x (deleted firstly), let us then delete $V^d \setminus \{x\}$. Now the remaining graph are still induced by V^r satisfying for each $v \in V^r$, $deg(v) \geq S.c$. Since $u \in V^r$, u is not a collapsed follower of x, which contradicts with our assumption.

For situation 3), in graph G without collapsing x, for each $v \in S.V$, we denote $N^>(v) = \{w \in N(v, G) : c(w) > S.c\}$. As $c(v) = S.c$, we have $|N(v, S) \cup N^>(v)| \geq$

$S.c$. With collapsing x in G, core decomposition firstly deletes each $v \in V(G)$ s.t. $c_x^-(v) < S.c$. Reconsider each $w \in N^>(v)$ s.t. each $v \in S.V$. For situation 3), $w \neq x$. And based on Lemma 4.3.1, $c_x^-(w) \geq S.c$. Thus, w remains in the graph. Now consider each $v \in S.V$, $N^>(v)$ are complete in the remaining graph, so we still have $|N(v,S) \cup N^>(v)| \geq S.c$. Thus, v is not a collapsed follower of x. As $u \in S.V$, this contradicts with our assumption.We prove it by contradiction. Assuming there are S_1 and S_2 with $v \in V(S_1)$ and $v \in V(S_2)$. Then we have, for each $u \in V(S_1)$, $c(u) = c(v)$ and u is connected with v; For each $u \in V(S_2)$, $c(u) = c(v)$ and u is connected with v. Thus, $S_1 \cup S_2$ satisfies the definition of one single shell component, which contradicts with our assumption. Proof completes.

Theorem 4.3.3 *For anchoring x in G, another vertex u is x's anchored follower indicates $x \in \mathcal{A}[S]$ s.t. $u \in S.V$.*

Proof 4.3.5 *We prove it by contradiction. Assuming u increases its coreness since x is anchored, $u \in S.V$ and $x \notin \mathcal{A}[S]$. This firstly means $x \notin S.V$, according to the definition of $\mathcal{A}[S]$, then we have the following possible situations: 1) $c(x,G) > S.c$; 2) $c(x,G) = S.c$; 3) $c(x,G) < S.c \wedge N(x,G) \cap S.V = \emptyset$.*

For situation 1), No matter x is anchored or not, x is always deleted after u in core decomposition (Algorithm 14), so that $c_x^+(u) = c(u)$ which contradicts with our assumption.

For situation 2), Consider deleting $V(G)$ in core decomposition without anchoring x. After all the vertices with coreness less than $S.c$ are deleted, it is available to delete $S.V$ before each $S'.V$ s.t. $S' \neq S$ and $S'.c = S.c$. And each such S' can remain in the graph still satisfying for each $v \in S'.V$, $deg(v) \geq S.c$. We denote the deleted vertex set so far as V^d. Because $c(x) = S.c$ and $x \notin S.V$, we can conclude $x \notin V^d$. For the graph with anchoring x, as $x \notin V^d$, we can safely still delete V^d following the same vertex

sequence. Thus for each $v \in V^d$, $c_x^+(v) = c(v)$. Since $u \in V^d$, u is not an anchored follower of x, which contradicts with our assumption.

For situation 3), in graph G without anchoring x, for each $v \in S.V$, we denote $N^<(v) = \{w \in N(v, G) : c(w) < S.c\}$ and $N^>(v) = \{w \in N(v, G) : c(w) > S.c\}$. As $c(v) = S.c$, we have $|N(v, S) \cup N^>(v)| < S.c + 1$. When conducting core decomposition with anchoring x in G, reconsider each $w \in N^<(v)$ s.t. each $v \in S.V$. For situation 3), $w \neq x$. And based on Lemma 4.3.2, $c_x^+(w) < S.c+1$, thus each such w is not in the $(S.c+1)$-core. Now consider each $v \in S.V$, $N^<(v)$ are not in the $(S.c+1)$-core and we still have $|N(v, S) \cup N^>(v)| < S.c+1$. Thus, v is not an anchored follower of x. As $u \in S.V$, this contradicts with our assumption.

Parallel Computation. For each $v \in V(G)$, We define ${}^-F[v][S] = \{u \in S.V : u \in {}^-\mathcal{F}(v, G)\}$ and ${}^+F[v][S] = \{u \in S.V : u \in {}^+\mathcal{F}(v, G)\}$, so as to compute v's collapsed followers and anchored followers in each atom unit, i.e., the shell component. Based on Theorems 4.3.2 and 4.3.3, we know that for the vertex v, its collapsed followers (resp. anchored followers) are only from each $S.V$ s.t. $v \in \mathcal{C}[S]$ (resp. $v \in \mathcal{A}[S]$). Thus, in Algorithm 17, we parallelly compute ${}^-F[v][S]$ (resp. ${}^+F[v][S]$) w.r.t. $\{S : v \in \mathcal{C}[S]\}$ (resp. $\{S : v \in \mathcal{A}[S]\}$). Then ${}^-\mathcal{F}(v, G) = \bigcup_{\{S:\, v \in \mathcal{C}[S]\}} {}^-F[v][S]$ and ${}^+\mathcal{F}(v, G) = \bigcup_{\{S:\, v \in \mathcal{A}[S]\}} {}^+F[v][S]$. In order to unify the offline computation and the online maintenance when any edge is inserted or removed, Algorithm 17 takes a new shell component set $\hat{S}$ as an input. Now we let $\hat{S}$ contain all the shell components in G, i.e., $\hat{S} := \bigcup_{u \in V(G)} \{\mathcal{SC}[u]\}$, then call `ParallelFollowerComp`($\mathcal{C}[\cdot]$, $\mathcal{A}[\cdot]$, $\hat{S}$). For each $S \in \hat{S}$, whenever there exists an available thread (Line 1), the computation in Line 2-7 can be conducted, followed by releasing this thread. As the number of shell components in G is much more than the number of available threads, the dynamic scheduling ensures the computing resources are mostly utilized in parallel. Based on theorem 4.3.1, each ${}^-F[v][S]$ (resp. ${}^+F[v][S]$) does not overlap with others,

which means the threads are never wasted on redundant computations. The functions `FindCollapsedFollowers` (Line 3) and `FindAnchoredFollowers` (Line 6) are equipped with further accelerating techniques, which are presented by Algorithm 19 and Algorithm 20 in subsection 4.3.3, respectively.

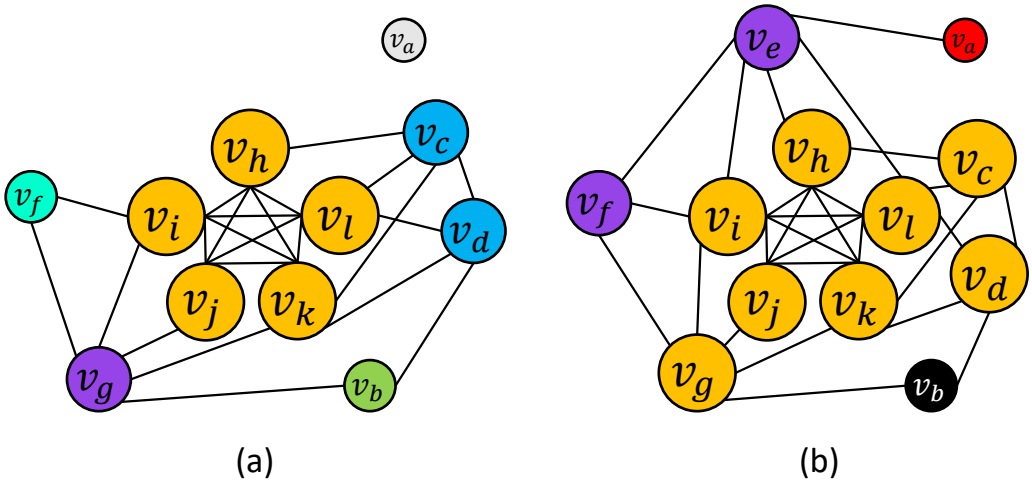


Figure 4.3: Followers Computation.

Example 4.3.2 *For the graph in Figure 4.2, Figure 4.3 (a) shows the graph with collapsing v_e. We retain the notations in Example 4.3.1. Please refer to Table 4.2 for your convenience. There are two shell components S_1 and S_4 such that $v_e \in \mathcal{C}[S_1]$ and $v_e \in \mathcal{C}[S_4]$. Then ${}^{-}F[v_e][S_1]$ and ${}^{-}F[v_e][S_4]$ can be computed in parallel. We have ${}^{-}F[v_e][S_1] = \{v_a\}$ with $c^{-}_{v_e}(v_a) = 0$, and ${}^{-}F[v_e][S_4] = \{v_f\}$ with $c^{-}_{v_e}(v_f) = 2$.*

Example 4.3.3 *For the graph in Figure 4.2, Figure 4.3 (b) shows the graph with anchoring v_b. We retain the notations in Example 4.3.1. Please refer to Table 4.2 for your convenience. There are three shell components S_2, S_3 and S_4 such that $v_b \in \mathcal{A}[S_2]$, $v_b \in \mathcal{A}[S_3]$ and $v_b \in \mathcal{A}[S_4]$. ${}^{+}F[v_b][S_2]$, ${}^{+}F[v_b][S_3]$ and ${}^{+}F[v_b][S_4]$ can be computed in parallel. We have ${}^{+}F[v_b][S_2] = \emptyset$, ${}^{+}F[v_b][S_3] = \{v_c, v_d\}$ with $c^{+}_{v_b}(v_c) = c^{+}_{v_b}(v_d) = 4$, ${}^{+}F[v_b][S_4] = \{v_g\}$ with $c^{+}_{v_b}(v_g) = 4$.*

Algorithm 17: **ParallelFollowerComp**($\mathcal{C}[\cdot]$, $\mathcal{A}[\cdot]$, $\hat{S}$)

Input : $\mathcal{C}[\cdot]$: the collapsed candidate set, $\mathcal{A}[\cdot]$: the anchored candidate set, $\hat{S}$: the new shell component set

Output : ${}^{-}\mathcal{F}(v, G)$ and ${}^{+}\mathcal{F}(v, G)$ for each $v \in V(G)$

1 **for** each $S \in \hat{S}$ in `dynamic multithreads` **do**
2 **for** each $v \in \mathcal{C}[S]$ **do**
3 ${}^{-}F[v][S] :=$ **FindCollapsedFollowers**(v, S);
4 ${}^{-}\mathcal{F}(v, G) := {}^{-}\mathcal{F}(v, G) \cup {}^{-}F[v][S]$;
5 **for** each $v \in \mathcal{A}[S]$ **do**
6 ${}^{+}F[v][S] :=$ **FindAnchoredFollowers**(v, S);
7 ${}^{+}\mathcal{F}(v, G) := {}^{+}\mathcal{F}(v, G) \cup {}^{+}F[v][S]$;
8 **return** ${}^{-}\mathcal{F}(v, G)$ and ${}^{+}\mathcal{F}(v, G)$ for each $v \in V(G)$

4.3.2 The Maintenance w.r.t. Edge Streaming

We consider the following two situations of edge streaming: 1) a single edge is removed from the graph; 2) a single edge is inserted into the graph. Please note that, multiple edges and vertices streaming can be regarded as a sequence of situations 1) and 2). We maintain the collapsed follower set ${}^{-}\mathcal{F}(v, G)$ and anchored follower set ${}^{+}\mathcal{F}(v, G)$ for each $v \in V(G)$ by Algorithm 18 which unifies situations 1) and 2). Specifically, in the updated graph G ($E(G) \cup \{(v_s, v_t)\}$ or $E(G) \setminus \{(v_s, v_t)\}$), we firstly adopt the state-of-the-art algorithm of core maintenance in [105] to update $c(u, G)$ for each $u \in V(G)$ (Line 1), followed by resetting each vertex as *unassigned* (Line 2-3). In Line 4, $\mathcal{SC}'[\cdot]$, $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$ are initially copied from the current $\mathcal{SC}[\cdot]$, $\mathcal{C}[\cdot]$ and $\mathcal{A}[\cdot]$, and they will be properly updated. Line 5-13 collects the new shell components into $\hat{S}$. Without the need to call Algorithm 15 again for the whole graph, the maintenance process only needs to start from each vertex u in $\mathcal{SC}[v_s]$ and $\mathcal{SC}[v_t]$ (Line 6). Line 7-12 do the same

operations as Line 3-8 of Algorithm 15. Please note that $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$ can be updated straightforward during Line 6-13 according to their definitions (Line 14), because only $\mathcal{A}'[S']$ and $\mathcal{C}'[S']$ for each $S' \in \hat{S}$ need to be updated. We denote an updated vertex set as $U = \bigcup_{S' \in \hat{S}} S'.V$ (Line 15). By traversing all the old shell components of the vertices in U (using $\mathcal{SC}$ instead of $\mathcal{SC}'$ in Line 16), we can remove the expired collapsed followers and anchored followers (Line 17-20, using $\mathcal{C}[\cdot]$ and $\mathcal{A}[\cdot]$ instead of $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$). At last, for the new shell component set $\hat{S}$, we call Algorithm 17 with inputting the updated $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$ (Line 21) to compute the new followers.

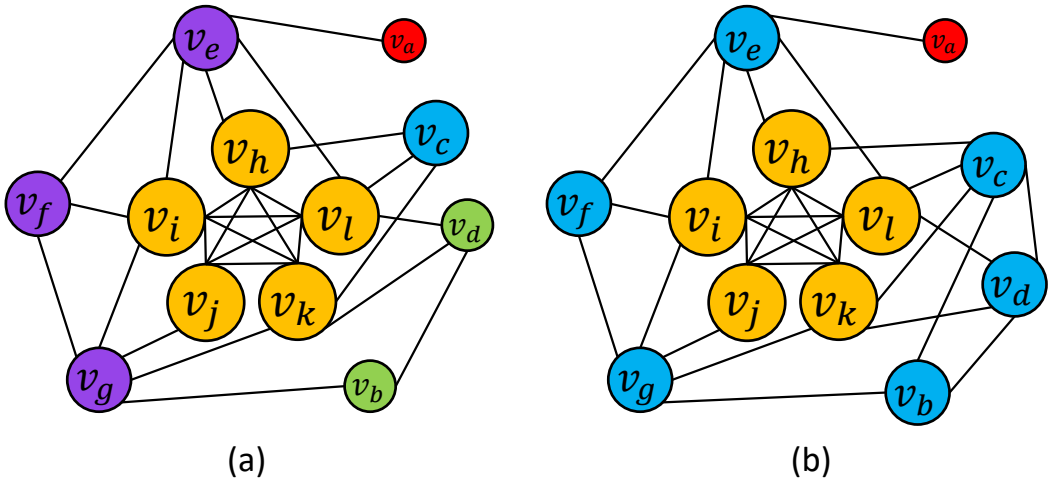


Figure 4.4: Followers Maintenance.

Example 4.3.4 *In Figure 4.4 (a), we remove the edge (v_c, v_d) from the graph in Figure 4.2. All the notations in Example 4.3.1 (Table 4.2) are retained. S_2 and S_3 are expired. We have the new S_6 with $S_6.V = \{v_b, v_d\}$ and $S_6.c = 2$, and the new S_7 with $S_7.V = \{v_c\}$ and $S_7.c = 3$. Table 4.3 shows all the updated shell components and the vertices in $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$. The collapsed and anchored followers of S_1, S_4 and S_5 retain the same (row 2-4). Only the collapsed and anchored followers of S_6 and S_7 need to be computed (row 5-6).*

Example 4.3.5 *In Figure 4.4 (b), we insert the edge (v_b, v_c) into the graph in Figure 4.2. All the notations in Table 4.2 are retained. S_2, S_3 and S_4 are expired. We have the new S_8 with $S_8.V = \{v_b, v_c, v_d, v_e, v_f, v_g\}$ and $S_8.c = 3$. Table 4.4 shows all the updated shell components and the vertices in $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$. The collapsed and anchored followers of S_1 and S_5 retain the same (row 2-3). Only the collapsed and anchored followers of S_8 need to be computed (row 4).*

Table 4.3: Followers Maintenance w.r.t. Removing (v_c, v_d)

	$\mathcal{C}'[\cdot]$	$\mathcal{A}'[\cdot]$
S_1	v_a, v_e	v_a
S_4	v_e, v_f, v_g, v_h, v_i, v_j, v_k, v_l	v_a, v_e, v_f, v_g
S_5	v_h, v_i, v_j, v_k, v_l	v_c, v_d, v_e, v_f, v_g, v_h, v_i, v_j, v_k, v_l
S_6	v_b, v_d, v_g, v_k, v_l	v_b, v_d
S_7	v_c, v_h, v_k, v_l	v_c

Table 4.4: Followers Maintenance w.r.t. Inserting (v_b, v_c)

	$\mathcal{C}'[\cdot]$	$\mathcal{A}'[\cdot]$
S_1	v_a, v_e	v_a
S_5	v_h, v_i, v_j, v_k, v_l	v_c, v_d, v_e, v_f, v_g, v_h, v_i, v_j, v_k, v_l
S_8	v_b, v_c, v_d, v_e, v_f, v_g, v_h, v_i, v_j, v_k, v_l	v_a, v_b, v_c, v_d, v_e, v_f, v_g

Algorithm 18: **FollowerMaintain(**(v_s, v_t)**,** G**)**

Input : (v_s, v_t) : the inserted or removed edge, G : the graph with $E(G) \cup \{(v_s, v_t)\}$ or $E(G) \setminus \{(v_s, v_t)\}$

1 Conduct the core maintenance [105] to get $c(u, G)$ for each $u \in V(G)$;

2 **for** each $u \in V(G)$ **do**

3 u is set as $unassigned$;

4 $\mathcal{SC}'[\cdot], \mathcal{C}'[\cdot], \mathcal{A}'[\cdot] \leftarrow \mathcal{SC}[\cdot], \mathcal{C}[\cdot], \mathcal{A}[\cdot]$;

5 $\hat{S} \leftarrow$ the new shell component set;

6 **for** each $unassigned\ u \in S.V$ s.t. each $S \in \{\mathcal{SC}[v_s], \mathcal{SC}[v_t]\}$ **do**

7 $S' \leftarrow$ a new shell component;

8 $S'.c := c(u, G)$;

9 $S'.V := S'.V \cup \{u\}$;

10 u is set $assigned$;

11 **ShellConnect**(u, S', $\mathcal{SC}'$);

12 $\mathcal{SC}'[u] := S'$;

13 $\hat{S} = \hat{S} \cup \{S'\}$;

14 $\mathcal{C}'[\cdot]$ and $\mathcal{A}'[\cdot]$ are updated while doing Line 6-13;

15 $U \leftarrow \bigcup_{S' \in \hat{S}} S'.V$;

16 **for** each $S \in \bigcup_{u \in U} \{\mathcal{SC}[u]\}$ **do**

17 **for** each $v \in \mathcal{C}[S]$ **do**

18 ${}^{-}\mathcal{F}(v, G) := {}^{-}\mathcal{F}(v, G) \setminus {}^{-}F[v][S]$;

19 **for** each $v \in \mathcal{A}[S]$ **do**

20 ${}^{+}\mathcal{F}(v, G) := {}^{+}\mathcal{F}(v, G) \setminus {}^{+}F[v][S]$;

21 **ParallelFollowerComp**($\mathcal{C}'[\cdot]$, $\mathcal{A}'[\cdot]$, $\hat{S}$);

4.3.3 The Efficient Followers Computation

Now we introduce our efficient algorithms of computing the collapsed followers and anchored followers of one vertex in one shell component. For each shell component S in G, Algorithm 19 computes ${}^{-}F[v][S]$ for each $v \in \mathcal{C}[S]$, and Algorithm 20 computes ${}^{+}F[v][S]$ for each $v \in \mathcal{A}[S]$. Both of the algorithms need the *higher coreness support* of each $u \in V(G)$, which is introduced as follows.

Higher Coreness Support. For each $u \in V(G)$, we define its higher coreness support, denoted by $HS(u)$, as the number of u's neighbors having higher coreness than u, i.e.,$HS(u) = |\{v \in N(u, G) : c(v) > c(u)\}|$. The higher coreness supports are incidentally computed and recorded when traversing each vertex's neighbours in Algorithm 15.

Collapsed Followers Computation. In Algorithm 19, we utilize a queue Q (Line 1) to explore the collapsed followers, starting from the collapser vertex x. If $x \in S.V$, x is set *discarded* and pushed into Q (Line 2-3). Note that, all the vertices in $S.V$ are not *discarded* initially, and any *discarded* vertex (except for x) becomes a collapsed follower. If $x \notin S.V$, for each $u \in N(x, G) \cap S.V$, we reduce $HS(u)$ by 1 and push u into Q (Line 4-6). Then we traverse Q until it becomes empty (Line 7). Each time when we pop the top vertex u (Line 8), if $u \neq x$, we need to decide whether it is *discarded* (Line 9-12). Specifically, we compute a degree upper bound $d^{+}(u)$ as Line 10 shows. If $d^{+}(u) < S.c$, u is set *discarded*, and we push each $v \in N(u, S)$ into Q (Line 13-15). Note that, we can avoid repeatedly push the same vertex into Q (Line 14). After traversing Q, all the *discarded* vertices in $S.V$ except for x form ${}^{-}F[x][S]$ (Line 16).

Algorithm 19: **FindCollapsedFollowers(x, S)**

Input : x : the collapser candidate, S : a shell component

Output : ${}^{-}F[x][S]$: the collapsed follower set of x in S

```
1  Q ← a queue;
2  if x ∈ S.V then
3      x is set discarded; Q.push(x);
4  else
5      for each u ∈ N(x, G) ∩ S.V do
6          HS(u) := HS(u) − 1; Q.push(u);
7  while Q ≠ ∅ do
8      u ← Q.pop();
9      if u ≠ x then
10         d⁺(u) := HS(u) + |{v : v ∈ N(u, S) ∧ v is not discarded}|;
11         if d⁺(u) < S.c then
12             u is set discarded;
13     if u is discarded then
14         for each v ∈ N(u, S) with v is not discarded and v ∉ Q do
15             Q.push(v);
16 ⁻F[x][S] ← discarded vertices in S.V except for x ;
17 return ⁻F[x][S]
```

Anchored Followers Computation. We use Algorithm 20 to compute ${}^{+}F[x][S]$, the anchored followers of x in S. The algorithm is straightforward adapted from the Algorithm 4 of [69], so we omit the theoretical analysis and the proof of algorithm correctness. Please also refer to *Degree Check* and Theorem 4.15 in section 4.4 of [69]. Instead, Example 4.3.6 illustrates the process of Algorithm 20 in detail. The core of Algorithm 20 is utilizing the *layer value* of each vertex, which needs to be illustrated firstly as follows.

Layer Value. 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 14). We use H_k^i to denote the i-layer of the k-shell, which is the set of vertices that are deleted in the i-th batch. Specifically, when $i = 1$, H_k^i is defined as $\{u : deg(u, C_k(G)) < k+1 \ \wedge \ u \in C_k(G)\}$. The deletion of the 1st-layer will produce the 2nd-layer. Recursively, when $i > 1$, $H_k^i = \{u : deg(u, G_i) < k+1 \ \wedge \ u \in G_i\}$ where $G_1 = C_k(G)$ and G_i is the subgraph induced by $V(G_{i-1}) \setminus H_k^{i-1}$ on $C_k(G)$. For each vertex $u \in V(G)$, there is only one H_k^i s.t. $u \in H_k^i$ during core decomposition, then we denote $l(u) = i$ as the *layer value* of u. Obviously, the layer values are incidentally computed and recorded when conducting Algorithm 14.

Algorithm 20: **FindAnchoredFollowers(x, S)**

Input : x : the anchor candidate, S : a shell component

Output : ${}^{+}F[x][S]$: the anchored follower set of x in S

$H \leftarrow$ a min heap w.r.t. the layer value of each vertex;
if $x \in S.V$ **then**
    x is set *survived*; $H.push(x)$;
else
    for each $u \in N(x, G) \cap S.V$ **do**
        $HS(u) := HS(u) + 1$; $H.push(u)$;
while $H \neq \emptyset$ **do**
    $u \leftarrow H.pop()$;
    if $u \neq x$ **then**
        $d^{+}(u) := HS(u) + |\{v : v \in N(u, S) \wedge l(v) \leq l(u) \wedge (v \textit{ is survived} \vee v \in H)\}| + |\{v : v \in N(u, S) \wedge l(v) > l(u) \wedge v \textit{ is not discarded}\}|$;
        if $d^{+}(u) \geq S.c + 1$ **then**
            u is set *survived*;
    if u is *survived* **then**
        for each $v \in N(u, S)$ with $l(v) > l(u)$ and $v \notin H$ **do**
            $H.push(v)$;
    else
        u is set *discarded* ;
        Shrink(u) (Algorithm 21);
${}^{+}F[x][S] \leftarrow$ *survived* vertices in $S.V$ except for x ;
return ${}^{+}F[x][S]$

Algorithm 21: **Shrink**(u)

Input : u : the shrinked vertex

```
1 for each survived neighbor v with v ≠ x do
2     d+(v) := d+(v) − 1;
3     if d+(v) < S.c + 1 then
4         v is set discarded;
5         T.add(v);
6 for each v ∈ T do
7     Shrink(v);
```

Example 4.3.6 *For the graph G in Figure 4.2 (Table 4.2), we follow all the notations in Example 4.3.1. For v_b, we need to compute ${}^{+}F[v_b][S_2]$, ${}^{+}F[v_b][S_3]$ and ${}^{+}F[v_b][S_4]$. We follow the process of* `FindAnchoredFollowers`*(v_b, S_3) in Algorithm 20 as an example. As $v_b \notin S_3.V$ (Line 4), we have $HS(v_d) = 2 + 1 = 3$ (originally $HS(v_d) = 2$ w.r.t. v_k and v_l) and v_d is pushed into H (Line 5-6). After v_d is popped (Line 8), we compute $d^+(v_d)$ (Line 10). Because $l(v_d) = 1$, $l(v_c) = 2$ and v_c is not discarded, $d^+(v_d) = 3+0+1 = 4$. Since $S_3.c = 3$, v_d is set survived (Line 11-12) and v_c is pushed into H (Line 13-15). Then v_c is popped and $d^+(v_c) = 3+1+0 = 4$, so v_c is set survived and no more vertex is pushed into H. Now we can return ${}^{+}F[v_b][S_3] = \{v_c, v_d\}$. Note that, any vertex is neither discarded or survived unless it is explicitly set so. Once a vertex needs to be set discarded (Line 17), it may cause a cascade of vertices discarded. We call Algorithm 21 (Line 18) which recursively discards all the need-to-be vertices.*

Table 4.5: Statistics of Datasets

Dataset	Nodes	Edges	d_{avg}	d_{max}	k_{max}
Facebook(**F.**)	22,470	170,823	15.2	709	56
Brightkite(**B.**)	58,228	194,090	6.7	1098	52
Github(**H.**)	37,700	289,003	15.3	9458	34
Gowalla(**G.**)	196,591	456,830	9.2	10721	51
NotreDame(**N.**)	325,729	1,497,134	6.5	3812	155
Stanford(**S.**)	281,903	2,312,497	16.4	38626	71
Youtube(**Y.**)	1,134,890	2,987,624	5.3	28754	51
DBLP(**D.**)	1,566,919	6,461,300	8.3	2023	118

Table 4.6: Percentage of Valid Collapsers & Anchors

Dataset	F.	B.	H.	G.	N.	S.	Y.	D.
Collapsers	61%	44%	49%	50%	24%	38%	28%	69%
Anchors	69%	70%	78%	74%	50%	33%	64%	54%

4.4 Experimental Evaluation

Datasets. We use 8 real-life datasets for experiments. Facebook, Brightkite, Github, Gowalla & Youtube are from [63]. NotreDame, Stanford and DBLP are from [59]. Due to the Space limitation, we abbreviate each dataset's name as a unique bold capital letter when necessary as in Table 4.5. Table 4.5 also shows the statistics of the datasets, listed in increasing order of edge numbers.

Algorithms. To the best of our knowledge, no existing work studies each node's influence for network structural stability. Other works aim to find a small number of critical nodes to maximize (resp. minimize) other nodes' engagements, e.g., anchored k-core (resp. collapsed k-core), or maximize the spread of information, e.g., influence maximization. These works cannot be compared with our method in a reasonable way.

Parameters. All the programs are implemented in C++ and compiled with G++ on Linux. The server has 3.4GHz Intel Xeon CPU with 4 cores (8 threads available) and Redhat system. We adopt OpenMP to utilize the multithreads of the machine.

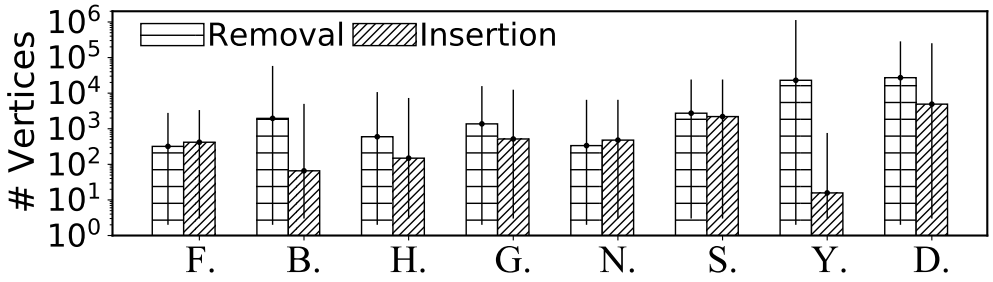
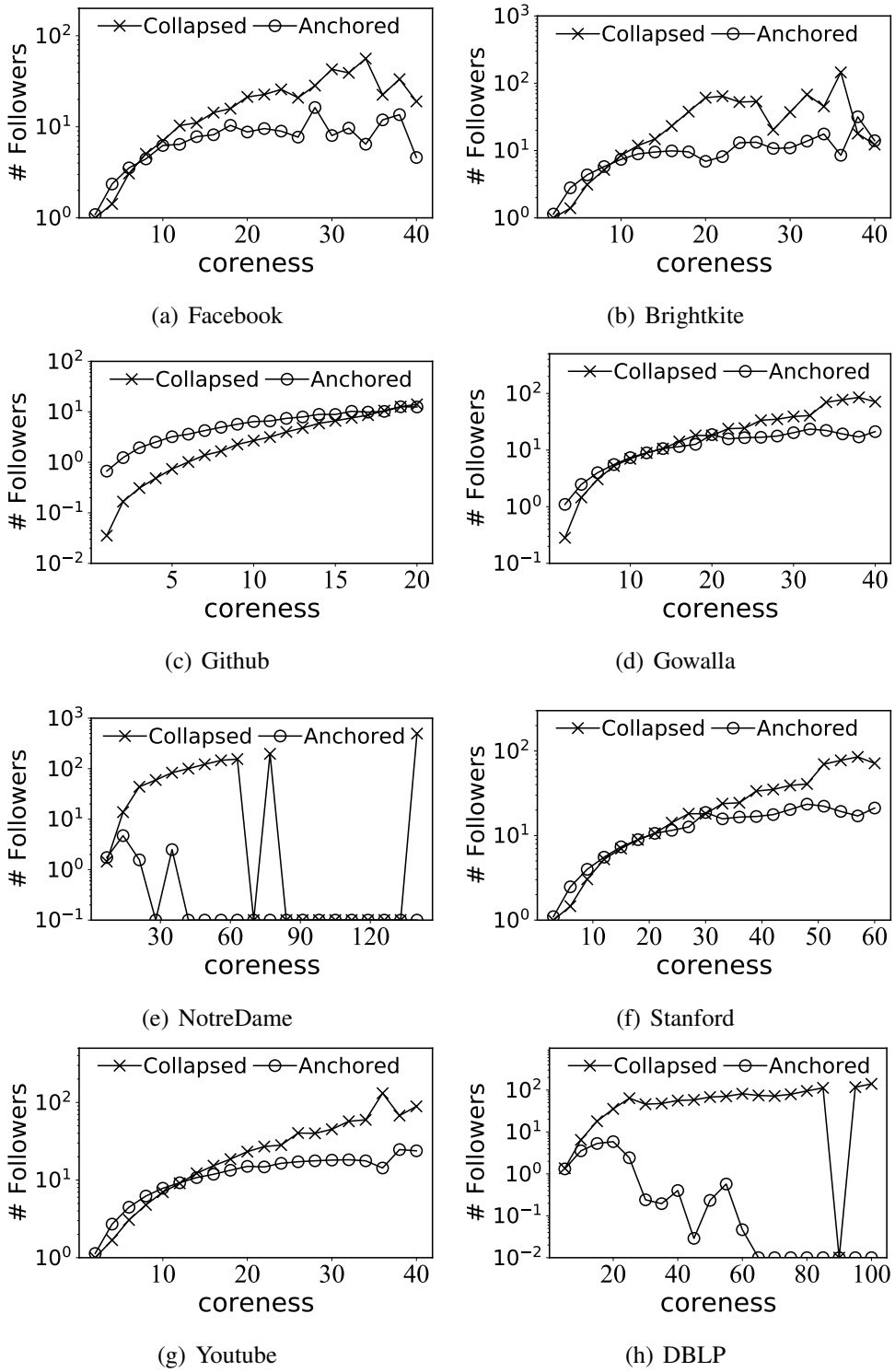


Figure 4.5: # Updated Vertices w.r.t. Edge Streaming

4.4.1 Effectiveness

Percentage of Valid Collapsers & Anchors. In Table 4.6, for each dataset, we present the percentage of vertices which have non-empty collapsed follower set (resp. anchored follower set), which we call the valid collapsers (resp. valid anchors). Firstly, we can find the percentages are higher than 50% on most datasets for both the valid collapsers and valid anchors. This means both the collapsing and anchoring behaviors of vertices indeed have apparent effects on the network, which demonstrates the necessity of monitoring each vertex's collapsed and anchored follower set. Secondly, we find on some datasets such as `Stanford` and `DBLP`, the number of valid collapsers are more than valid anchors. However, on all the other datasets, the number of valid anchors are more than valid collapsers. This means vertex collapsing and vertex anchoring have different influence on different networks due to the different network structures, thus it is necessary to monitor each vertex's influence via both the collapsed and anchored follower set.

Followers Distribution on Vertex Coreness. In Figure 4.6, we present the distribution of the number of followers (collapsed and anchored) on vertex's coreness value. For each dataset, k_{max} denotes the largest coreness value from all the vertices, which is shown in the statistics of Table 4.5. We divide the coreness range $[1, k_{max}]$ into 20



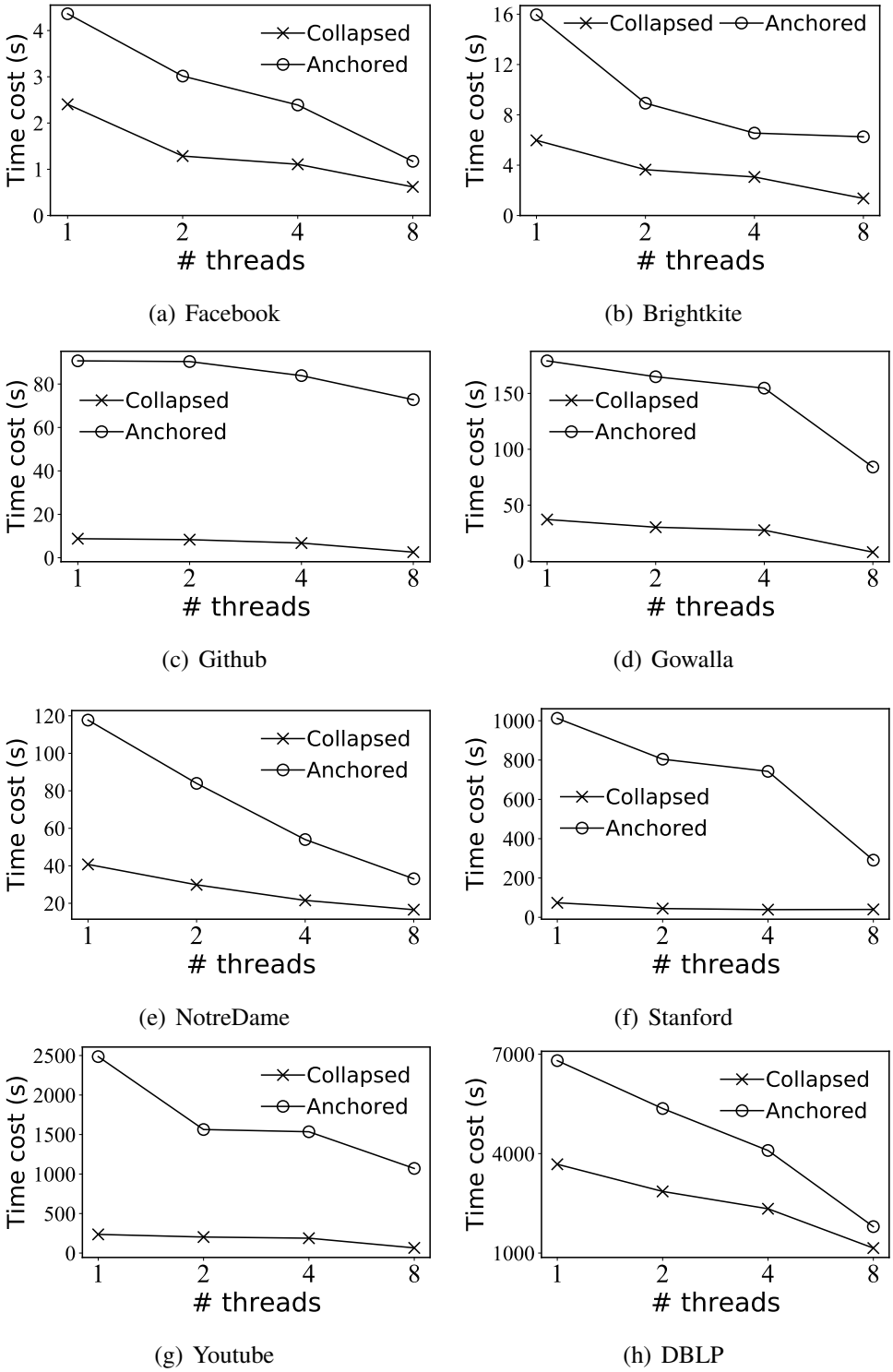
(a) Facebook (b) Brightkite

(c) Github (d) Gowalla

(e) NotreDame (f) Stanford

(g) Youtube (h) DBLP

Figure 4.6: # Followers Distribution on Vertex Coreness



(a) Facebook (b) Brightkite

(c) Github (d) Gowalla

(e) NotreDame (f) Stanford

(g) Youtube (h) DBLP

Figure 4.7: Computation Efficiency and Scalability

equal integer-width intervals, with the last remained interval combined with the second last interval. For each coreness value x_i on the horizontal ordinate of Figure 4.6, we compute the mean number of collapsed followers (resp. anchored followers) of all the vertices having their own coreness value within $(x_{i-1}, x_i]$. Note that, it is not necessary for a graph that, for each value in $[1, k_{max}]$, there exists vertices with coreness equal to the value. For the values which no vertex has the coreness equal to, the mean numbers of collapsed or anchored followers are considered as zero, e.g., the 'Collapsed' lines of `NotreDame` and `DBLP` in Figure 4.6. We then have the following observations. Firstly, we can find on most coreness values on most datasets, the mean number of collapsed followers are more than anchored followers. However, remember that in Table 4.6, the number of valid anchors are more than valid collapsers on the contrary. This means the collapsed followers and anchored followers reflect two different views of node influence on network structural stability and it is necessary to monitor both of them. Secondly, we find that, except when the coreness value is little, the number of collapsed and anchored followers are not always positive-correlated with the coreness value. This demonstrates that, we cannot simply decide a node's influence on network structural stability simply based on its own engagement, and it is essential to actually compute the collapsed and anchored followers.

Amount of Updated Vertices w.r.t. Edge Streaming. On each dataset, we randomly remove 100 edges and insert 100 edges. Each time an edge is removed or inserted, we record the number of vertices which have updates in either the collapsed or anchored follower set. The result is shown in Figure 4.5. The main boxes show the mean amount of updated vertices. We can find that, either removing or inserting a single edge can cause the update amount of other vertices from 10^1 to 10^4 on average. We also add the error-bar on each box to show the minimum and maximum amount of updated vertices among the 100 edge removal and insertion. We can find that the minimum amount of up-

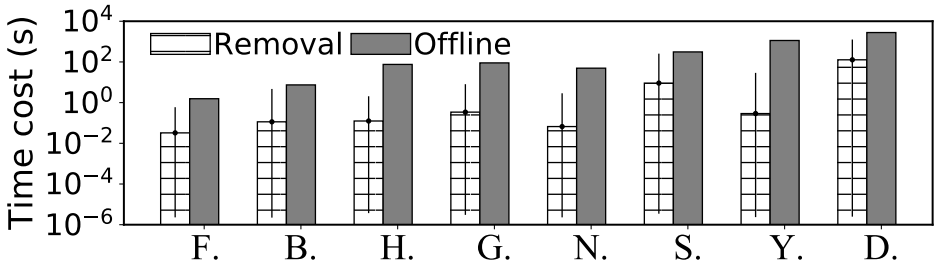
dated vertices are generally close to 1 and the maximum amount can reach up to around 10^5 in extreme cases (`Brightkite` and `Youtube`). The significant amount of vertices in need of updating the follower sets demonstrates the necessity of our maintaining technique regarding edge streaming.

4.4.2 Efficiency

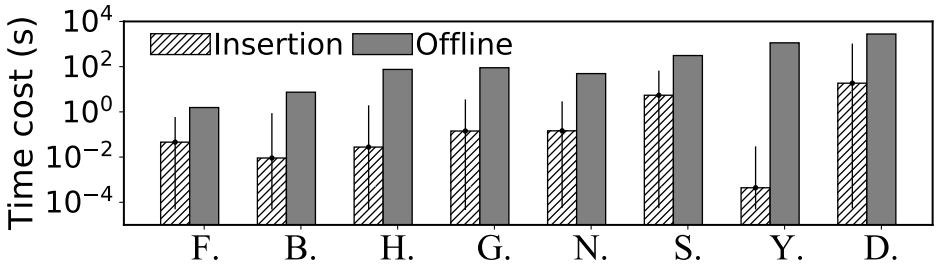
Offline Computation Efficiency and Scalability. We vary the number of threads from 1, 2, 4 to 8 and test the efficiency and scalability of our offline algorithm of computing the collapsed and anchored follower set of each vertex. The time cost is presented in Figure 4.7. We can find that, on all the datasets, the time cost tends to be less with increasing the number of threads, for the computation of both collapsed and anchored followers. This means our parallel algorithm based on separating the computation into each shell component helps us take the advantage of parallel architecture. We find the time cost is not always inversely proportional to the number of threads. This is because different shell components can have drastically different number of collapsed (resp. anchored) candidates, and the overall time cost depends on the thread (or threads) running the computation for the shell components having large number of candidates. Thus, the future work can consider to deign the technique of separating such large shell components, in order to more evenly distribute the computation tasks across multiple threads.

Online Maintenance Efficiency. We test the efficiency of our online maintenance algorithm. As each real-life temporal network may have its own biased dynamic pattern, in order to thoroughly test the efficiency of our maintenance algorithm in diverse situations of edge insertion and removal, we randomly sample 100 edges for removal and insertion respectively in all the datasets. Each time an edge is removed (resp. inserted), we record the time cost of maintaining both the collapsed and anchored follower set for all the vertices. In Figure 4.8, the main boxes compare the mean time cost of the 100 maintenance

to the offline computation time, for edge removal (a) and edge insertion (b), respectively. We can find that, the maintenance time is 2 to 4 orders of magnitude faster than the offline computation, and the maintenance of edge insertion is generally faster than the maintenance of edge removal. We also add the error-bars to show the minimum and maximum maintenance time among the 100 edge removals and 100 edge insertions. We can find that, in the worst cases, the maintenance time cost is still less than the time cost of offline computation. And in the optimal cases, the maintenance time cost is constantly little regardless of the scale of datasets. Overall, the experiments presented in Figure 4.8 demonstrate the efficiency of our maintaining technique for evolving networks.



(a) Edge Removal



(b) Edge Insertion

Figure 4.8: Maintenance Time w.r.t. Edge Streaming

4.5 Chapter Conclusion

In this chapter, we innovatively propose a model that estimates a node's influence on the structural stability of the network by two views: the collapsed power and anchored power. Then a parallel algorithm is proposed to compute each node's collapsed and anchored followers offline, equipped with online maintenance technique to update the nodes' followers when the network is evolving. Our effectiveness experiments on real-life datasets demonstrate the necessity of both the collapsed power and anchored power. Our efficiency experiments show that both our offline and online algorithms are efficient.

Chapter 5

Summary and Future Work

The study on social networks becomes increasingly important with the growing capacity and activity networking sites, e.g., TikTok, Instagram and Facebook etc. Towards the structural stability of social networks which is an important indicator for both the network holders and the participants, three fundamental objectives are explored in this thesis: (i) globally reinforce the stability of social networks considering the engagements of all the users; (ii) develop the efficient parallel solution for the proposed social network reinforcement model; and (iii) monitor each user's influence regarding its role in network structural stability.

5.1 Summary

Towards objective (i), we propose a new model, named anchored coreness, which is innovative to globally reinforce the social network stability. Anchored coreness aims to anchor a set of vertices such that the coreness gain from all the vertices is maximized. As we prove this problem is NP-hard and APX-hard, we resort to greedy heuristic to propose an efficient algorithm `GAC`. `GAC` is based on a novel tree structure so that the intermediate results between different iterations of the greedy algorithm can be reused.

GAC is also equipped with effective pruning techniques to reduce the search space of the computations for the anchor candidates. We conduct extensive experiments on 8 real-life social networks, which demonstrate the effectiveness of our new proposed model and the efficiency of our greedy algorithm.

Towards objective (ii), based on GAC, we further extend the greedy algorithm to distributed parallel computing environment to provide a scalable solution for massive social networks. The parallel algorithm DGAC is based on a novel graph partition strategy which ensures the computations regarding different anchor candidates can be concurrently and independently conducted on different machines or threads. More experiments on the 8 real-life social networks show that DGAC holds good scalability with more machines and more threads in a machine sharing the computation. Our proposed parallel algorithm also sheds light on the computations for other problems on hierarchical decomposition, e.g., truss decomposition. It implies that the computation holds data locality and can be divided into independent units.

Towards objective (iii), we propose a novel model to estimate a user's influence on the structural stability of a social network, by two views: the collapsed power and anchored power. The two powers are corresponding to two natural status of a user: engagement weakening and engagement strengthening. The computations of each user's collapsed followers and anchored followers are integrated to an offline parallel algorithm, and online maintenance technique is also proposed to update each user's followers when the social network is dynamically evolving. We conducted extensive effectiveness and efficiency experiments on 8 real-life social networks, which demonstrate 1) both the collapsed power and anchored power are necessary to capture; 2) both our offline and online algorithms are efficient.

5.2 Future Work

Deep learning has revolutionized many data mining and artificial intelligence tasks in recent years, e.g., computer vision, speech recognition and natural language processing etc. All the data in these tasks are typically represented in the Euclidean space. However, many real-life data such as this thesis's studied object, i.e., social networks, cannot be represented in Euclidean space but are represented by graphs. Luckily, many studies on extending deep learning approaches for graph data have emerged, among which graph neural networks (GNNs) are the most typical deep learning architecture. Besides, reinforcement learning and graph embedding are also the modern machine learning methods which enjoy more popularity nowadays. This section mainly discusses the potential of machine learning models on further improving the effectiveness and efficiency of the models proposed in this thesis.

Firstly, the computation of a vertex's anchored followers is a common subroutine of our global reinforcement model GAC, its parallel distributed extension DGAC and our user influence monitoring model. Although we apply the pruning rules to reduce the search space, it is still rather a time-consuming process. Thus, we can consider to train a supervised learning model to estimate the anchored followers number for each vertex. Specifically, we can offline compute the anchored followers number for part of vertices as the training set, then train the parameters of one of the classic GNNs, e.g., GCN [57], GraphSAGE [47] and GAT [92]. After the training, GPU architecture can be used to efficiently estimate other vertices' anchored followers number, which is potentially much faster than computing each vertices's actual anchored followers. With the help of transfer learning technique, we can further adapt the estimation model trained from smaller social networks to larger social networks without much more time of training.

Secondly, our proposed model anchored coreness is a combinatorial optimization problem, which has much potential to be improved from the pure greedy heuristic, i.e.,

the coreness gain of our pure greedy algorithm `GAC` may has a not little gap from the optimal coreness gain. Utilizing supervised learning to estimate each vertex's anchored followers number still does not consider the unseen combinatorial structure of vertices. In [33], Dai *e*t al.claim the instances of the same type of optimization problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data. This brings an opportunity for automatically designing a heuristic algorithm which exploits the structure of such recurring problems, which is addressed by a unique combination of reinforcement learning and graph embedding in [33]. Thus, we can consider to apply such methods to learn a new heuristic algorithm for the anchored coreness problem which may enjoy both better coreness gain and computation efficiency.

Bibliography

[1] MPICH. https://https://www.mpich.org/.

[2] OpenMP. https://www.openmp.org/.

[3] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.

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

[5] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy. Distributed k-core view materialization and maintenance for large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2439–2452, 2014.

[6] R. D. Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3(1):113–126, 1973.

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

[8] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *DEBS*, pages 161–168. ACM, 2016.

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

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

[11] K. Bhawalkar, J. M. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma. Preventing unraveling in social networks: The anchored k-core problem. In *ICALP*, pages 440–451, 2012.

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

[13] M. P. Blanco, T. M. Low, and K. Kim. Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In *HPEC*, pages 1–7. IEEE, 2019.

[14] M. Bola and B. A. Sabel. Dynamic reorganization of brain functional networks during cognition. *NeuroImage*, 114:398–413, 2015.

[15] F. Bonchi, A. Khan, and L. Severini. Distance-generalized core decomposition. In *SIGMOD*, pages 1006–1023, 2019.

[16] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[17] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *Proceedings of the 20th international conference on World wide web*, pages 665–674, 2011.

[18] J. Cannarella and J. A. Spechler. Epidemiological modeling of online social network dynamics. *arXiv preprint arXiv:1401.4208*, 2014.

[19] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, 104(27):11150–11154, 2007.

[20] T. H. Chan, M. Sozio, and B. Sun. Distributed approximate k-core decomposition and min-max edge orientation: Breaking the diameter barrier. In *IPDPS*, pages 345–354. IEEE, 2019.

[21] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1):173–186, 2013.

[22] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, pages 205–216, 2013.

[23] S. Chen, J. Fan, G. Li, J. Feng, K.-l. Tan, and J. Tang. Online topic-aware influence maximization. *Proceedings of the VLDB Endowment*, 8(6):666–677, 2015.

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

[25] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.

[26] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1240–1248, 2012.

[27] R. Chitnis, F. V. Fomin, and P. A. Golovach. Parameterized complexity of the anchored k-core problem for directed graphs. *Inf. Comput.*, 247:11–22, 2016.

[28] R. H. Chitnis, F. V. Fomin, and P. A. Golovach. Preventing unraveling in social networks gets harder. In *AAAI*, 2013.

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

[30] A. Conte, D. Firmani, M. Patrignani, and R. Torlone. Shared-nothing distributed enumeration of 2-plexes. In *CIKM*, pages 2469–2472. ACM, 2019.

[31] A. Conte, T. D. Matteis, D. D. Sensi, R. Grossi, A. Marino, and L. Versari. D2K: scalable community detection in massive networks via small-diameter k-plexes. In *SIGKDD*, pages 1272–1281. ACM, 2018.

[32] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002, 2014.

[33] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

[34] A. Das, S. Sanei-Mehri, and S. Tirthapura. Shared-memory parallel maximal clique enumeration from static and dynamic graphs. *ACM Trans. Parallel Comput.*, 7(1):5:1–5:28, 2020.

[35] P. Domingos and M. Richardson. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 57–66, 2001.

[36] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense implicit communities in the web graph. *TWEB*, 3(2):7:1–7:36, 2009.

[37] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*, pages 364–375. Springer, 2011.

[38] H. Esfandiari, S. Lattanzi, and V. S. Mirrokni. Parallel and streaming algorithms for k-core decomposition. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1396–1405. PMLR, 2018.

[39] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *PVLDB*, 10(6):709–720, 2017.

[40] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[41] S. Galhotra, A. Arora, and S. Roy. Holistic influence maximization: Combining scalability and efficiency with opinion-aware models. In *Proceedings of the 2016 International Conference on Management of Data*, pages 743–758, 2016.

[42] D. García, P. Mavrodiev, and F. Schweitzer. Social resilience in online communities: the autopsy of friendster. In *Conference on Online Social Networks*, pages 39–50, 2013.

[43] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *AAAI*, pages 44–50, 2014.

[44] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan. A data-based approach to social influence maximization. *Proc. VLDB Endow.*, 5(1):73–84, 2011.

[45] A. Goyal, W. Lu, and L. V. Lakshmanan. Celf++ optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*, pages 47–48, 2011.

[46] Q. Guo, S. Wang, Z. Wei, and M. Chen. Influence maximization revisited: Efficient reverse reachable set generation with bound tightened. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2167–2181, 2020.

[47] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[48] X. He, G. Song, W. Chen, and Q. Jiang. Influence blocking maximization in social networks under the competitive linear threshold model. In *Proceedings of the 2012 siam international conference on data mining*, pages 463–474. SIAM, 2012.

[49] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Trans. Parallel Distrib. Syst.*, 31(6):1287–1300, 2020.

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

[51] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Trans. Parallel Distrib. Syst.*, 29(11):2416–2428, 2018.

[52] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *IPDPS Workshops*, pages 1482–1491. IEEE Computer Society, 2017.

[53] H. Kabir and K. Madduri. Parallel k-truss decomposition on multicore systems. In *HPEC*, pages 1–7. IEEE, 2017.

[54] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[55] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In L. Getoor, T. E. Senator, P. M. Domingos, and C. Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 137–146. ACM, 2003.

[56] W. Khaouid, M. Barsky, S. Venkatesh, and A. Thomo. K-core decomposition of large networks on a single PC. *PVLDB*, 9(1):13–23, 2015.

[57] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

[59] J. Kunegis. The konect project. `http://konect.cc/`.

[60] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[61] S. Lei, S. Maniu, L. Mo, R. Cheng, and P. Senellart. Online influence maximization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 645–654, 2015.

[62] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429, 2007.

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

[64] R. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, and Z. Zheng. Skyline community search in multi-valued networks. In *SIGMOD*, pages 457–472, 2018.

[65] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2013.

[66] Y. Li, J. Fan, Y. Wang, and K.-L. Tan. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 30(10):1852–1872, 2018.

[67] Y. Li, D. Zhang, and K.-L. Tan. Real-time targeted influence maximization for online advertisements. 2015.

[68] J.-H. Lin, Q. Guo, W.-Z. Dong, L.-Y. Tang, and J.-G. Liu. Identifying the node spreading influence with largest k-core values. *Physics Letters A*, 378(45):3279–3284, 2014.

[69] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang. Global reinforcement of social networks: The anchored coreness problem. In *SIGMOD*, pages 2211–2226. ACM, 2020.

[70] B. Liu, G. Cong, D. Xu, and Y. Zeng. Time constrained influence maximization in social networks. In *2012 IEEE 12th international conference on data mining*, pages 439–448. IEEE, 2012.

[71] W. Lu, W. Chen, and L. V. S. Lakshmanan. From competition to complementarity: Comparative influence diffusion and maximization. *Proc. VLDB Endow.*, 9(2):60–71, 2015.

[72] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[73] F. D. Malliaros, M.-E. G. Rossi, and M. Vazirgiannis. Locating influential nodes in complex networks. *Scientific reports*, 6:19307, 2016.

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

[75] A. Mandal and M. A. Hasan. A distributed k-core decomposition algorithm on spark. In *BigData*, pages 976–981. IEEE Computer Society, 2017.

[76] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.

[77] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.

[78] F. Morone, G. Del Ferraro, and H. A. Makse. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics*, 15(1):95, 2019.

[79] M. P. O'Brien and B. D. Sullivan. Locally estimating core numbers. In *2014 IEEE International Conference on Data Mining*, pages 460–469. IEEE, 2014.

[80] N. Ohsaka, T. Akiba, Y. Yoshida, and K.-i. Kawarabayashi. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[81] J. Pattillo, N. Youssef, and S. Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.

[82] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *SIGKDD*, pages 228–238, 2005.

[83] J. Qiu, J. Tang, H. Ma, Y. Dong, K. Wang, and J. Tang. Deepinf: Social influence prediction with deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2110–2119, 2018.

[84] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

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

[86] K. Seki and M. Nakamura. The collapse of the friendster network started from the center of the core. In *ASONAM*, pages 477–484, 2016.

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

[88] Y. Shao, L. Chen, and B. Cui. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*, pages 613–624, 2014.

[89] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In *HPEC*, pages 1–6. IEEE, 2017.

[90] B. Tootoonchi, V. Srinivasan, and A. Thomo. Efficient implementation of anchored 2-core algorithm. In *ASONAM*, pages 1009–1016, 2017.

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

[92] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[93] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[94] J. Wang, J. Cheng, and A. W.-C. Fu. Redundancy-aware maximal cliques. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 122–130, 2013.

[95] Y. Wang, Q. Fan, Y. Li, and K. Tan. Real-time influence maximization on dynamic social streams. *Proc. VLDB Endow.*, 10(7):805–816, 2017.

[96] Z. Wang, Q. Chen, B. Hou, B. Suo, Z. Li, W. Pan, and Z. G. Ives. Parallelizing maximal clique and k-plex enumeration over graph data. *J. Parallel Distributed Comput.*, 106:79–91, 2017.

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

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

[99] M. Ye, X. Liu, and W.-C. Lee. Exploring social influence for recommendation: a generative model approach. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 671–680, 2012.

[100] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*. USENIX Association, 2010.

[101] F. Zhang, J. Xie, K. Wang, S. Yang, and Y. Jiang. Discovering key users for defending network structural stability. *World Wide Web*, pages 1–23, 2021.

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

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

[104] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing*, 53(2):352–369, 2010.

[105] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 337–348. IEEE Computer Society, 2017.

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

[107] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*, pages 480–491, 2012.

[108] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin. Enumerating maximal k-plexes with worst-case time guarantee. In *AAAI*, pages 2442–2449. AAAI Press, 2020.