Cohesive Subgraph Mining on Social Networks

by

Fan Zhang

B.E. Zhejiang University, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

the Centre for Artificial Intelligence (CAI)
the Faculty of Engineering and Information Technology (FEIT)
the University of Technology Sydney (UTS)

August, 2017

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Candidate

ACKNOWLEDGEMENTS

First and foremost, I would like to deliver my sincere gratitude to my supervisor Prof. Ying Zhang for his continuous support of my PhD study and research, especially for his professionalism, patience, passion and diligence. His guidance extends my knowledge in computer science, improves my capacity in scientific research and elevates my love in exploring the fields which are significant, undiscovered and challenging. Besides the character of supervisor, Ying has also been a friend and mentor. Thanks to his confidence in me, I never lost hope when experiencing failures and was always positive during my PhD study. Without his consistent and illuminating instruction, this thesis could not have reached its present form.

Secondly, I would like to express my great gratitude to my co-supervisor Dr. Lu Qin for his constant encouragement and guidance, especially for his brilliant ideas and inspirations. His work efficiency gave me the hope to conduct good research without abandoning too many of other interests, which prevented me to be negative in the early stage of my PhD study. Lu always has confidence in solving research problems, regardless of their complexities, which encourages me to keep thinking and challenging myself.

Thirdly, parts of the work in this thesis were conducted in collaboration with Prof. Xuemin Lin and Dr. Wenjie Zhang. I thank them for supporting the work presented in this thesis. A special thanks to Prof. Lin for providing a rigorous and self-motivated research environment, and the support for my academic career. I learnt the characteristics of an excellent researcher from Prof. Lin - passion, diligence and earnestness in the research.

I would also like to thank the following people at UNSW and UTS, Australia: Dr. Xiaoyang Wang, Dr. Shiyu Yang, Dr. Lijun Chang, Dr. Zengfeng Huang, Dr. Xin Cao, and Dr. Gaoping Zhu, for sharing your brilliant ideas and experiences. Thanks to Ms. Haiyan Hua, Ms. Chen Zhang, Dr. Xing Feng, Ms. Shan Xue, Dr. Long Yuan, Dr. Longbin Lai, Dr. Xiang Wang, Dr. Jianye Yang, Dr. Chengyuan Zhang, Mr. Xubo Wang, Mr. Fei Bi, Mr. Wei Li, Mr. Dong Wen, Mr. Haida Zhang, Ms. Qing Bing, Ms. Wen Li, Ms. Lu Liu, Ms. Shenlu Wang, Ms. Conggai Li, Mr. Yang Yang, Mr. Kai Wang, Mr. Mingjie Li, Mr. Wentao Li and Mr. Hanchen Wang, for sharing the happiness and bitterness with me during my PhD study. The time we spent together will be memorized forever.

Last but not least, thank my family: my father Mr. Lingchi Zhang and my mother Mrs. Liqin Zhou, for bringing me a happy and wonderful life in the world, and other relatives for their support, encouragement and love.

PUBLICATIONS

- **Fan Zhang**, Ying Zhang, Lu Qin, Wenjie Zhang, Xuemin Lin. When Engagement Meets Similarity: Efficient (k,r)-Core Computation on Social Networks. PVLDB 2017. (Chapter 3)
- **Fan Zhang**, Wenjie Zhang, Ying Zhang, Lu Qin, Xuemin Lin. OLAK: An Efficient Algorithm to Prevent Unraveling in Social Networks. PVLDB 2017. (Chapter 4)
- **Fan Zhang**, Ying Zhang, Lu Qin, Wenjie Zhang, Xuemin Lin. Efficiently Reinforcing Social Networks over User Engagement and Tie Strength. Under submission. (Chapter 4)
- **Fan Zhang**, Ying Zhang, Lu Qin, Wenjie Zhang, Xuemin Lin. Finding Critical Users for Social Network Engagement: The Collapsed k-Core Problem. AAAI 2017. (Chapter 5)

TABLE OF CONTENT

CERTIFICATE OF AUTHORSHIP/ORGINALITY iii

ACKNOWLEDGEMENTS iv

PUBLICATIONS vi

TABLE OF CONTENT vii

LIST OF FIGURES x

LIST OF TABLES xii

ABSTRACT xiii

Chapter 1 Introduction 1
- 1.1 Motivation . . . 2
  - 1.1.1 Cohesive Subgraph Discovery . . . 2
  - 1.1.2 Prevent Network Unraveling . . . 4
  - 1.1.3 Finding Critical Users . . . 6
- 1.2 Contribution . . . 7
  - 1.2.1 Cohesive Subgraph Discovery . . . 7
  - 1.2.2 Prevent Network Unraveling . . . 8
  - 1.2.3 Finding Critical Users . . . 10
- 1.3 Organization . . . 11

Chapter 2 Literature Review 12
- 2.1 Cohesive Subgraph Models . . . 12
  - 2.1.1 Clique . . . 12
  - 2.1.2 k-Core . . . 14
  - 2.1.3 k-Truss . . . 16
  - 2.1.4 Other Models . . . 18
- 2.2 Social Network Components . . . 19
  - 2.2.1 User Engagement . . . 19

2.2.2 User Similarity 20
2.2.3 Tie Strength 21
2.3 Mining Attributed Graphs 22

Chapter 3 Cohesive Subgraph Discovery **24**
3.1 Introduction 24
3.2 Preliminaries 28
3.2.1 Problem Definition 29
3.2.2 Problem Complexity 31
3.3 The Clique-based Approach 33
3.4 Warming Up for Our Approach 34
3.5 Finding All Maximal (k,r)-Cores 36
3.5.1 Reducing Candidate Size 36
3.5.2 Early Termination 39
3.5.3 Checking Maximal 40
3.5.4 Advanced Enumeration Method 42
3.6 Finding the Maximum (k,r)-core 45
3.6.1 Algorithm for Finding the Maximum One 45
3.6.2 Size Upper Bound of (k,r)-Core 46
3.6.3 Algorithm for (k,k')-Core Upper Bound 49
3.6.4 Finding the Top-m Maximal (k,r)-Cores 50
3.7 Search Order 51
3.7.1 Important Measurements 51
3.7.2 Finding the Maximum (k,r)-Core 52
3.7.3 Enumerating All Maximal (k,r)-Core 53
3.7.4 Checking Maximal 54
3.8 Performance Evaluation 54
3.8.1 Experimental Setting 54
3.8.2 Effectiveness 57
3.8.3 Efficiency 60
3.9 Conclusion 66

Chapter 4 Prevent Network Unraveling **67**
4.1 Introduction 67
4.2 Preliminaries 73
4.2.1 Problem Definition 73
4.2.2 Problem Complexity 76
4.3 Our Approach 77
4.3.1 Motivation 77
4.3.2 Reducing the Number of Candidate Anchors 79
4.3.3 Efficiently Finding Followers 80

4.3.4 The OLAK Algorithm . . . 90
4.4 Prevent Collapse of Strong Tie Communities . . . 94
4.4.1 The Problem of Anchored k-Truss . . . 95
4.4.2 The Problem Hardness . . . 97
4.4.3 The Edge Onion Layers . . . 101
4.4.4 Candidate Anchors and Followers . . . 103
4.4.5 Efficiently Finding Candidate Followers . . . 104
4.4.6 Finding Followers with Early Termination . . . 107
4.4.7 The AKT Algorithm . . . 111
4.5 Performance Evaluation . . . 113
4.5.1 Experimental Setting for Anchored k-Core . . . 113
4.5.2 Effectiveness of OLAK . . . 115
4.5.3 Efficiency of OLAK . . . 120
4.5.4 Experimental Setting for Anchored k-Truss . . . 124
4.5.5 Effectiveness of AKT . . . 126
4.5.6 Efficiency of AKT . . . 129
4.6 Conclusion . . . 132

Chapter 5 Finding Critical Users **133**
5.1 Introduction . . . 133
5.2 Preliminaries . . . 136
5.2.1 Problem Definition . . . 136
5.2.2 Problem Complexity . . . 138
5.3 Our Approach . . . 142
5.3.1 Motivation . . . 142
5.3.2 Reducing Candidate Collapsers . . . 143
5.3.3 CKC Algorithm . . . 145
5.4 Performance Evaluation . . . 146
5.4.1 Experimental Setting . . . 146
5.4.2 Effectiveness . . . 147
5.4.3 Efficiency . . . 150
5.5 Conclusion . . . 152

Chapter 6 Epilogue **153**
6.1 Conclusions . . . 153
6.2 Future Work . . . 155

REFERENCES **156**

LIST OF FIGURES

3.1 Group by Friendship and Interests 26
3.2 Group by Friendship and Locations 27
3.3 Example of the Search Tree 35
3.4 Pruning and Retaining Candidates 39
3.5 Early Termination and Check Maximal 41
3.6 Upper Bound Examples 48
3.7 Case Study on DBLP (k=15, r=top 3‰) 58
3.8 Case Study on Gowalla (k=10, r=10km) 59
3.9 (k,r)-core Statistics 60
3.10 Evaluate Clique-based Method 61
3.11 Evaluate Pruning Techniques 62
3.12 Evaluate Upper Bounds 62
3.13 Evaluate Search Orders 63
3.14 Performance on Four Datasets 64
3.15 Effect of k and r for Enumeration 65
3.16 Effect of k and r for Maximum 65

4.1 Motivating Example for Anchored k-Core 68
4.2 Motivating Example for Anchored k-Truss 70
4.3 Bounded Anchors and Followers 80
4.4 Onion Layer Structure (L_0^3) 83
4.5 Early Termination 87
4.6 Followers Pruning and Upper Bound Pruning 91
4.7 Construction Example for NP-hardness Proof, $k = 4$ 98
4.8 Examples for Non-submodular 100
4.9 Examples for Edge Onion Layers $\mathcal{L}$, $k = 4$ 104
4.10 Number of Followers 116
4.11 Greedy vs Exact 117
4.12 Effect of k and b 118
4.13 Case Studies 119
4.14 Reducing Candidate Anchors 120
4.15 Pruning Candidate Followers 121

4.16 Effectiveness of Early Termination 122
4.17 Further Pruning Candidate Anchors 123
4.18 Running Time on Different Datasets 124
4.19 Effect of k and b 125
4.20 Number of Followers 127
4.21 Greedy vs Exact 128
4.22 Case Studies 129
4.23 Reducing Candidate Anchors and Followers 130
4.24 Effect of Algorithms with Different k and b 131
4.25 Running Time on Different Datasets 131

5.1 Motivating Example 134
5.2 Examples for NP-hardness Proof 139
5.3 Examples for Non-submodular 142
5.4 Number of the Followers 148
5.5 Greedy vs Optimal 149
5.6 Case Study on DBLP, k=20, b=1 149
5.7 Effectiveness of Reducing Candidate Collapsers 151
5.8 Performance of Baseline and CKC 151

LIST OF TABLES

3.1 The summary of notations 29
3.2 Statistics of Datasets 55
3.3 Summary of Algorithms 56

4.1 Summary of Notations 74
4.2 Summary of New Notations 96
4.3 Statistics of Datasets 113
4.4 Summary of Algorithms for Anchored k-Core 114
4.5 Summary of Algorithms for Anchored k-Truss 126

5.1 Summary of Notations 137
5.2 Statistics of Datasets 147

ABSTRACT

Graphs are widely used to represent the abundant information in social networks for discovering promising communities, reinforcing network stability, and finding critical users, to name a few. Cohesive subgraph mining, as one of the most fundamental problems in graphs, gains increasing popularity in social network study for its effectiveness. In this thesis, some basic social components are considered in cohesive subgraphs to better accommodate various real-life applications.

Firstly, we investigate the problem of (k,r)-core which intends to find cohesive subgraphs on social networks considering both user engagement and similarity. Efficient algorithms are proposed to enumerate all *maximal* (k,r)-cores and find the *maximum* (k,r)-core, where both problems are shown to be NP-hard. Effective pruning techniques and search orders substantially reduce the search space of two algorithms. A novel upper bound enhances performance of the maximum (k,r)-core computation. Comprehensive experiments on real-life data demonstrate that the algorithms efficiently find interesting communities.

Secondly, we study the problem of the anchored k-core, which was introduced by Bhawalkar and Kleinberg *e*t al. in the context of user engagement in social networks. The problem has been shown to be NP-hard and inapproximable. We propose an efficient algorithm, namely `OLAK`, as the first to solve the problem on general graphs. An *onion layer* structure is designed together with efficient

candidates exploration, early termination and pruning techniques to significantly simplify computation and greatly reduce the search space.

Besides considering user engagement, we further explore the unraveling phenomenon with tie strength, which leads us to the model of k-truss. We then investigate the anchored k-truss problem which is also NP-hard and propose an *edge onion layer* structure based algorithm, namely `AKT`. Efficient candidate exploration and pruning techniques are designed based on the *edge onion layers*. Comprehensive experiments on real-life graphs for the above two problems demonstrate the effectiveness and efficiency of our proposed methods.

Finally, we study the leave of critical users, which may greatly break network engagement. Accordingly, we propose the collapsed k-core problem to find the vertices whose leave can lead to the smallest k-core. We prove the problem is NP-hard. Then, an efficient algorithm is proposed, which significantly reduces the number of candidate vertices to speed up computation. Comprehensive experiments on real-life social networks demonstrate effectiveness of the model and efficiency of the proposed techniques.

Chapter 1

Introduction

With the growth of capacity and activity in social networking sites, such as Facebook and Twitter, the study of users and communities on social networks is becoming increasingly important. To effectively mine cohesive communities, analyze network structures and find critical users, social networks can be naturally modeled as graphs, where vertices represent individuals and edges represent friendships. Cohesive subgraphs, as one of fundamental components in graphs, are induced by groups of well-connected nodes which correspond to social communities. Although a variety of cohesive subgraph models have been proposed and extensively studied, some significant but unsolved problems, based on these models, are still in urgent need of research strength. In this thesis, three basic components of social networks are studied, together with cohesive subgraphs: (i) user engagement; (ii) user similarity; and (iii) tie strength. Considering specific scenarios with these components on cohesive subgraphs, three fundamental objectives on social networks are explored: (i) discover subgraphs with cohesiveness on both user engagement and similarity; (ii) prevent unraveling of social networks based on subgraph decompositions; and (iii) find critical users to reinforce communities. The details are introduced in the following.

1.1 Motivation

1.1.1 Cohesive Subgraph Discovery

Nowadays data becomes diverse and complex in real-life social networks, which not only consist of users and friendship, but also have various attribute values on each user. As such, social networks can be further modeled as *attributed graphs* where vertices represent users, edges represent friendship and vertex attribute is associated with specific properties, such as locations or keywords. Most of the existing work only consider the structure cohesiveness of the subgraphs. However, in practice we usually need to consider both structure and attribute perspectives when we aim to find a cohesive subgraph. We move beyond the simple structure-based cohesive subgraph models and advocate a complicated but more realistic cohesive subgraph model on attributed graphs, namely (k,r)-core. Particularly, we consider two intuitive and important criteria for a cohesive subgraph in real-life social networks: *engagement* and *similarity*.

User Engagement. In social networks, the behavior of an individual can be influenced by that of his/her friends. It is a common practice to encourage the engagement of the group members by using the positive influence from their friends in the same group (e.g., [14, 36, 57, 65, 89]); that is, ensure there are a considerable number of friends for each individual user (vertex) in the group (subgraph). In [14], Bhawalkar and Kleinberg et al. use the game-theory to formally demonstrate that the popular k-core model can lead to a stable group (i.e., a cohesive subgraph regarding graph structure). In this thesis, we adopt the k-core model on the graph structure, where each vertex in the subgraph has at least k neighbors (*structure constraint*).

User Similarity. In addition to the engagement, we usually need to consider attribute similarities among users (vertices) in the group (subgraph). The

similarity of two users can be derived from a given set of attributes (e.g., location, interests, and user generated content), which varies in different scenarios (e.g., [15, 43, 46, 67, 80]). By connecting two users (vertices) whose similarity exceeds a given threshold r, we get a *similarity graph* to capture the similarities among users. In this thesis, we adopt the well-known *clique* model to capture the cohesiveness of users from similarity perspective; that is, the vertices of a cohesive subgraph in this thesis form a clique on the similarity graph, which can ensure pairwise similarity among users (*similarity constraint*).

The Model of (k, r)-Core. The engagement and similarity criteria may often be used together to measure the sustainability of social groups. For instance, Facebook shows that both engagement (the number of friends in the group) and similarity (e.g., similar pages liked and distance closeness) are two important criteria when an existing Facebook group is recommended to a user [1]. To capture both engagement and similarity, we introduce the (k,r)-core model which is defined as follows. We say a connected subgraph of G is a (k,r)-core if and only if it satisfies both structure and similarity constraints. More specifically, given an attributed graph G, a (k,r)-core is a k-core of G (*structure constraint*) and the vertex set of the (k,r)-core induces a clique on the corresponding similarity graph (*similarity constraint*). A (k,r)-core is maximal if none of its supergraphs is a (k,r)-core. It is less interesting to find non-maximal (k,r)-cores. In this thesis, we aim to efficiently enumerate the maximal (k,r)-cores.

In many social network applications, (k,r)-core can help to discover interesting groups, which are promising to become stable and active. These groups can greatly enhance the users' stickiness and their experience. For instance, game designers should increase the *stickiness* of games by encouraging, or even forcing, team playing [21]. By identifying candidate groups with high quality in terms of the engagement and similarity, (k,r)-core can help to quickly discover and rec-

ommend new groups. These groups have good potential to become active and stable.

The size of a social group is an important factor to measure the potential impact and influence of this group. The groups with larger size are more likely to attract attention from the social network companies and users. Thus, it is interesting to find the maximum (k,r)-core which is the (k,r)-core with the largest number of vertices, and find the top-m maximal (k,r)-cores with the top-m largest numbers of vertices. The maximum (k,r)-core can help to evaluate the social impact potential of the interest-based social groups. The top-m maximal (k,r)-cores can identify m groups in which the companies are more likely to be interested. Consequently, we also study the problem of efficiently finding the maximum (k,r)-core and top-m maximal (k,r)-cores.

1.1.2 Prevent Network Unraveling

In social networks, the behavior of an individual can be influenced by that of his/her friends. As mentioned, user engagement on social networks has been widely studied in recent years (e.g., [14, 25, 26, 27, 28, 65, 89]) to model the behavior of users where each user may choose to remain engaged in, or leave, a group or a community. In a basic model, a user will remain engaged if, and only if, at least k of his/her friends are engaged. A user with less than k friends engaged will leave. His/her departure may be contagious and form a cascade of departures in the network. This procedure is called *network unraveling* in which active individuals may leave by the negative influence of his/her friends. As shown in [14], the unraveling process stops when the remaining engaged individuals correspond to the k-core of the network, a well-known concept in graph theory, which is the maximal induced subgraph in which *every* vertex has at least k neighbors.

To prevent unraveling in social networks, Bhawalkar and Kleinberg *et* al. [14] formally introduce the problem of anchored k-core. The aim is to retain (anchor) some users with incentives to ensure they will not leave regardless of the behavior of others, so that the largest number of users will remain engaged when the unraveling stops. Formally speaking, it is to anchor a set of b vertices such that the induced k-core is the largest one. This problem has a wide range of applications, and can help to identify users whose participation is critical to overall engagement of the networks.

Tie Strength. Besides user engagement, tie strength among users is another fundamental component in social networks [33, 42, 71, 81]. The model of k-core fits well with the communities which do not require tie strength, such as a study group where students may discuss with other students in the group, even though their relationships are not quite strong. Nevertheless, some community types need to consider the strength of ties, e.g., a programming team in a company, where every member would like enough familiar colleagues in this team to conduct better collaboration and avoid unexpected troubles. Using the numbers of triangles to detect strong ties has been widely studied and verified to be effective [33, 71, 81].

The strong tie communities correspond to the model of k-truss [29] where each edge is contained in at least $k-2$ triangles in the subgraph. Besides ensuring the strength of ties, a subgraph of k-truss is also a (k-1)-core and a (k-1)-edge-connected subgraph where the latter remains connected whenever fewer than k-1 edges are removed in the subgraph. Recently, solving k-truss based community and decomposition problems appears in view [47, 48, 50, 83, 100, 104]. It further motivates us to study the problem of anchored k-truss to prevent collapse of strong tie communities, which is to retain (anchor) some users in the network such that the k-truss community has the largest number of users.

1.1.3 Finding Critical Users

As mentioned above, the network unraveling can correspond to the well-known model k-core on the consideration of user engagement [14, 85, 89]. Towards the dynamic of a social network, we can assume all users at a time stamp are initially engaged. A user will remain engaged if and only if sufficient number of friends are currently engaged. The friend number threshold can be learned according to ground-truth communities or decided by users. A user with insufficient number of friends engaged will drop out which may lead to the cascade of user departure. The unraveling tends to stop when the network reaches a k-core, whose size can be used to measure the overall engagement of the social network.

Here we further consider the possibility than a user in the k-core community may still choose to leave, which partly exists in real-life social networks. There can be numbers of unexpected reasons for a user to leave a community, such as interest conflict, hobby change, moving away from current country and so on. The departure of some users incurs negligible influence to the community, while the leave of some critical users can collapse the community enormously. It motivates us to find the critical users who have the power to break the community.

A natural question is that, given a limited budget b, how to find b vertices (i.e., users) in a network so that we can get the smallest k-core by removing these b vertices. This problem is named the collapsed k-core problem in this thesis, which aims to collapse the engagement of the network with the greatest extent for a given budget b. By developing an efficient and scalable solution for this problem, we can quickly identify critical users whose leave will collapse the network most severely. These users are critical for the overall engagement of social networks. For instance, we can find most valuable users, to sustain or destroy the engagement of the networks. We can also evaluate the robustness of network engagement against the vertex attack.

1.2 Contribution

1.2.1 Cohesive Subgraph Discovery

Although there is a linear algorithm for k-core computation [12] (i.e., only consider structure constraint), we prove that the problem of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core are both NP-hard because of the similarity constraint involved. A straightforward solution is the combination of the existing k-core and clique algorithms, e.g., enumerating the cliques on similarity graph and then checking the structure constraint on the graph. In Chapter 3, we show that this is not promising because of the isolated processing of structure and similarity constraints. In this thesis, we show that the performance can be immediately improved by considering two constraints (i.e., two pruning rules) at the same time without explicitly materializing the similarity graph. Then our technique contributions focus on further reducing the search space of two mining algorithms from the following three aspects: (i) effective pruning, early termination and maximal check techniques; (ii) (k,k')-core based approach to derive tight upper bound for the problem of finding the maximum (k,r)-core; and (iii) good search orders in two mining algorithms.

The following is a summary of our principal contributions for the two (k,r)-core problems.

- We advocate a novel cohesive subgraph model, called (k,r)-core, to capture the cohesiveness of subgraphs from both the graph structure and the vertex attributes. We prove that the problem of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core are both NP-hard.

- We develop efficient algorithms to enumerate the maximal (k,r)-cores with candidate pruning, early termination and maximal checking techniques.

- We also develop an efficient algorithm to find the maximum (k,r)-core. Particularly, a novel (k,k')-core based approach is proposed to derive a tight upper bound for the size of the candidate solution.

- Based on some key observations, we propose three search orders for enumerating maximal (k,r)-cores, checking maximal (k,r)-cores, and finding maximum (k,r)-core algorithms, respectively.

- Our empirical studies on real-life data demonstrate that interesting cohesive subgraphs can be identified by maximal (k,r)-cores and maximum (k,r)-core. The extensive performance evaluation shows that the techniques proposed in this thesis can greatly improve the performance of two mining algorithms.

1.2.2 Prevent Network Unraveling

It is shown in [14] that the anchored k-core problem is NP-hard and inapproximable when $k \geq 3$. Two following works show the problem is also NP-hard even on a planar graph [25, 26]. A polynomial-time algorithm in graphs with bounded tree-width was proposed in [14], but to the best of our knowledge there is no practical algorithm for general large-scale graphs. To avoid enumerating all possible anchor sets with size b, we resort to greedy heuristics, where the best anchor is calculated in each iteration by computing the k-core for each possible anchor vertex. As demonstrated in our empirical study, a straightforward implementation of the greedy algorithm is very time-consuming. For instance, it would take more than a month to find one best anchor for k-core on the medium size network `Yelp` with $552,339$ vertices and $1,781,908$ edges. The reasons are two-fold. (1) The large number of candidate anchors. It is clear that we do not need to anchor the vertices in the k-core of the graph. However, the number of

remaining vertices is still large. (2) Although the computing time of k-core is linear in the number of edges, the cost is expensive given the large number of candidate anchors.

To the best of our knowledge, we are the first to study the anchored k-truss problem to prevent unraveling of strong tie communities. We prove the problem is NP-hard when $k > 3$. Consider the hardness of the problem, we also adopt a greedy strategy to find a best anchor in each iteration. However, the cost of the greedy algorithm for anchored k-truss is much more than anchored k-core. The reasons are two-fold. (1) The larger number of candidate anchors. Since anchoring a vertex in k-truss may enlarge the k-truss subgraph, the initial candidate anchors for k-truss are all the vertices in the graph. Thus, the candidate anchors for k-truss are much more. (2) Although there is a polynomial algorithm for computing k-truss, its cost is more expensive than k-core.

Our principal contributions for the problems of anchored k-core and anchored k-truss are summarized as follows.

- We develop the first two efficient algorithms, `OLAK` and `AKT`, to solve the anchored k-core problem and the anchored k-truss problem on general large graphs, respectively.

- We introduce a novel *onion layer* structure $\mathcal{L}_v$, which divides a small set of *vertices* by layers, so that we can efficiently find the best anchor in each iteration of the greedy algorithm. We show that only the vertices in $\mathcal{L}_v$ need to be considered during computation, which significantly reduces the search space.

- We introduce a novel *edge onion layer* structure $\mathcal{L}_e$, which divides a small set of *edges* by layers. The structure also contains all the endpoints of the edges inside it, so that we can efficiently find a best anchor and only the

vertices in $\mathcal{L}_e$ need to be considered as candidate anchors. It significantly reduces the search space.

- By using the well-organized structures $\mathcal{L}_v$ and $\mathcal{L}_e$, we develop efficient algorithms to compute the gain for each candidate anchor in a layer-by-layer paradigm. With the concept of support path or triangle hold path, we only need to explore a very small portion of the vertices in $\mathcal{L}_v$ or the edges in $\mathcal{L}_e$. Together with early termination and pruning techniques, we further reduce the number of anchor and the time to compute anchoring gain.

- Our comprehensive experiments on 10 real-life networks demonstrate the effectiveness and efficiency of our proposed techniques. For instance, our algorithm can prevent the unraveling of 630 vertices in k-core computation by anchoring one single vertex in the `Pokec` network. Regarding the running time, our `OLAK` and `AKT` algorithms outperform the straightforward implementations of the greedy algorithms by orders of magnitude.

1.2.3 Finding Critical Users

To the best of our knowledge, we are the first to propose and investigate the collapsed k-core problem. Solving the problem can discover the critical users from community perspective, where the users may not have large degrees or other vertex perspective properties. Finding these hidden but critical users are crucial for the stability of the communities, and can reinforce the communities. Due to the hardness of the problem, we resort to greedy heuristics where the best vertex is obtained in each iteration to avoid enumerating all possible answer sets with size b. Through theoretical analyses, we significantly reduce the number of candidate vertices to speed up the computation.

Our principal contributions are summarized as follows.

- We propose a novel subgraph concept, namely collapsed k-core, and the corresponding collapsed k-core problem to find critical users in social networks. We prove the collapsed k-core problem is NP-hard for any k value.
- We develop an efficient algorithm, namely `CKC`, to solve the collapsed k-core problem, by significantly reducing candidate vertices to visit.
- Our comprehensive experiments on 9 real-life networks demonstrate the efficiency of our proposed techniques. A real example on `DBLP` is presented in the experiments to show the effectiveness of the collapsed k-core model.

1.3 Organization

The thesis is organized as follows:

- Chapter 2 introduces the related work on cohesive subgraph models, social network components and mining attributed graphs.
- Chapter 3 presents the two (k,r)-core problems, the corresponding algorithms and the experimental results.
- Chapter 4 presents the problems of anchored k-core and anchored k-truss, the corresponding algorithms and the experimental results.
- Chapter 5 presents the collapsed k-core problem, the corresponding algorithms and the experimental results.
- Chapter 6 concludes our research and discusses several possible directions for future work.

Chapter 2

Literature Review

2.1 Cohesive Subgraph Models

There are various cohesive subgraph models to accommodate different scenarios in the literature. A cohesive subgraph is induced by a set of vertices which satisfy certain cohesiveness constrains. In the following, some popular models on cohesive subgraph are introduced.

2.1.1 Clique

The clique [63], as one of the earliest cohesive subgraph models, is the most cohesive subgraph. Given a graph, a clique is defined as a subgraph where each pair of vertices has an edge, i.e., every vertex is adjacent to every other vertex in the subgraph. A clique S is called maximal if no supergraph of S is a clique. A clique S is called maximum if there is no clique larger than S, i.e., S has the largest number of vertices than any other clique. The definition of clique ensures that (i) the perfect familiarity and reachability among the vertices in a clique; and (ii) the leave of any vertex in a clique does not weaken the cohesiveness inside the clique, i.e., the reduced clique remains a clique. However, the definition of

clique is too restrictive for many applications because every two vertices have to be connected by an edge in the clique. Consequently, some clique relaxation models are proposed to accommodate different scenarios. Pattillo *e*t al. [70] analyze the differences between some clique relaxation models by investigating six different clique-based properties: distance, diameter, domination, degree, density and connectivity. Based on these properties, it introduces a taxonomy of clique relaxation models in the literature and present the implications of choosing one model over another.

To relax the requirement of direct interaction in clique, Luce [62] proposes the s-clique which requires the distance on the graph between any two vertices within the subgraph to be at most s. Since the intermediate vertices in a shortest path between two vertices in an s-clique may not be a part of the s-clique, Alba [5] proposes a new model named s-club. Besides the distance requirement, an s-club further requires that the intermediate vertices in a shortest path between two vertices in an s-club have to be a part of the s-club. The above two models mainly focus on the diameter of a subgraph, Quasi-clique model [3] comes to the view of the edge density in a subgraph which should be not less than a given threshold. Here the edge density of a subgraph S is the ratio of existing edge number in S to the number of edges S if every two vertices have an edge.

As a fundamental graph problem, maximal clique enumeration has been extensively studied. Most clique algorithms (e.g., [16]) are based on backtracking search. Eppstein and Strash [34] further speedup maximal clique enumeration by selecting pivots with good potential to reduce the search space in backtracking. Chang *e*t al. study maximal clique enumeration in a sparse graph [19]. Recently, Wang *e*t al. [84] utilize the overlaps among cliques to speedup the maximal clique enumeration. A variety of clique computation algorithms have been proposed in the literature (e.g., [23, 84]).

2.1.2 k-Core

Regarding the quasi-clique model, Seidman [76] argues that edge density is a rather averaging property and may result in a group with highly cohesive regions coupled with very sparse regions. Therefore, Seidman [76] proposes the k-core model, a maximal subgraph where each vertex has at least k neighbors in the subgraph. The k-cores are subsets of a network whose cohesion monotonically increases and size monotonically decreases, by the increase of k. For small and nontrivial values of k, the k-cores tend to be large which can be regarded as seedbeds for other highly cohesive subgraph models. The core number of a vertex is the largest value of k such that the k-core contains the vertex. Core decomposition is to compute the core number for every vertex in a graph [12, 14], which can be used to define the network unraveling sequence. This sequence considers both of the k-cores and their complements, which means both regions of strong ties and regions of weak ties are taken into account. The unraveling sequence is shown to be useful in global network comparisons with easy and efficient computations.

The model of k-core has been widely used in many applications such as social contagion [82], community detection [96], event detection [66], network analysis [7], network visualization [98, 100], internet topology [8, 18], dense subgraph problems [9], influence study [54], graph clustering [41], graph model validation [45], structure analysis of software system [97], protein function prediction [6], and user engagement in social networks [14, 65, 94, 95]. Besides the various applications, the concept of k-core has enjoyed much popularity in the theory because it can be used as subroutine for harder problems [53], such as computing cliques of size k, or k-truss subgraphs. The k-core can also provide approximations for the densest subgraph problem, and the densest at-least-k subgraph problem [58].

The k-core becomes more and more popular in social studies, because its degeneration property can be used to quantify engagement dynamics in real social networks [65]. Bhawalkar et al. [14] propose the problem of anchored k-core to prevent unraveling of social networks by retaining some users. Specifically, given a graph G, it is 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 and inapproximate. Chitnis et al. further prove that the anchored k-core problem is NP-hard even on a planar graph [25]. Considering the possibility that a user in the k-core community may still choose to leave for unexpected reasons, Zhang et al. [95] propose the collapsed k-core problem to find the critical users who can break the k-core community significantly. Specifically, given a graph G, it is to remove b vertices in k-core to decrease the largest possible number of vertices in the k-core of G. The problem is proven to be NP-hard for any valid value of k.

From the algorithmic perspective, Batagelj and Zaversnik [12] present a linear-time in-memory algorithm for core decomposition of a graph, which iteratively deletes a vertex with the smallest degree in the remaining graph, with time complexity of $O(m+n)$. Wen et al. [86] and Cheng et al. [22] propose I/O efficient algorithms for core number computation on graphs that cannot fit in the main memory of a machine. Locally computing and estimating core numbers are studied in [30] and [68] respectively. Khaouid et al. [53] evaluate several implementations for core decomposition of large graphs and conclude that it is affordable to perform core decomposition for large graphs in a consumer-grade PC. Core maintenance is to maintain the k-core for any k after inserting or deleting edges in the graph. The algorithms for core number maintenance are proposed by [4, 60, 72, 99]. Zhang et al. [99] present the state-of-the-art algorithm to maintain the core numbers based on k-order which is an instance of all the possible vertex deletion sequences produced by a core decomposition algorithm.

For the anchored k-core problem, we do not need to decompose or estimate core numbers and we aim to propose an in-memory solution, the only applicable existing technique is core maintenance, which can be used by setting the core number of a candidate anchor as infinite and inserting its edges. By this way, we evaluated the core maintenance algorithm [99] for the anchored k-core problem in the experiments. For the collapsed k-core problem, we do not need to consider the core maintenance algorithms since the computation is limited on the k-core. A polynomial-time algorithm [14] for the anchored k-core problem on graphs with bounded tree-width has been proposed. However, this assumption usually does not hold in real-life social networks. Motivated by this, Zhang *et* al. [94] present an efficient algorithm to solve the anchored k-core problem by utilizing the vertex deletion order in k-core computation. For the anchored k-core problem, we need to consider the vertices *not* in k-core because it is useless to anchor vertices already in k-core. This is different from the problem of collapsed k-core, where the deletion of a vertex *not* in k-core will not affect the resulting k-core. To the best of our knowledge, we are the first to propose an algorithm on general graphs to solve the anchored k-core problem and the first to study the collapsed k-core problem to find critical users for social network engagement.

2.1.3 k-Truss

Cohen [29] proposes the model of k-truss which is a maximal subgraph where every edge exists in at least $k-2$ triangles in the subgraph, i.e., every edge has a support (number of common neighbors for two endpoints) of at least $k-2$ in the subgraph. The k-truss as an edge triangle based model, not only captures users with high engagement but also ensures the tie strength between users inside the subgroup. The truss number of an edge is the largest value of k such that the k-truss contains the edge. Truss decomposition is to compute the truss number

for every edge in a graph. Cohen [29] also presents the first algorithm for truss decomposition, which iteratively deletes an edge with the smallest support in the remaining graph, with time complexity of $O(\sum_{v \in V(G)}(\deg(v)^2))$. Wang and Cheng [83] reduce the time complexity of truss decomposition to $O(m^{1.5})$ by using a subtle trick to find the edges whose supports should decrease after the removing an edge. They also study the I/O efficient truss decomposition in large graphs which cannot fit in memory.

Huang *e*t al. [50] extend the model of k-truss to probabilistic graphs such that each edge in the k-truss has at least γ probability to be contained in at least $k-2$ triangles, where γ is a given threshold. Shao *e*t al. [79] study the k-truss detection problem on distributed systems. They propose a parallel and efficient algorithm based on the devised triangle complete subgraph for each computing node. Zhao and Tung [100] introduce the concept of k-truss as k-mutual-friend to capture the cohesion in social interactions, and propose an I/O efficient approach to discover cohesive subgraphs. They also propose a system to visualize the communities based on k-truss, which can quickly locate active social groups and interactions in a unified view. Huang *e*t al. [47] present a k-truss based community model which further requires edge connectivity inside the community, and design a compact tree-shape index to search the k-truss based community. The index preserves the truss number and triangle adjacency relationship to support the query of k-truss community in linear time with respect to the community size. They further identify the scope in a graph that is affected by edge insertion or deletion to apply the k-truss community query in dynamic graphs.

To the best of our knowledge, we are the first to propose the anchored k-truss problem to prevent unraveling of strong tie communities. There are some major differences between the algorithms for anchored k-core and anchored k-truss, such as the vertex support conditions are inherently different.

2.1.4 Other Models

Lee *et* al. [57] proposed an extension of k-core, namely (k,d)-core, where both k-core and d-truss structures are enforced on the same graph. Seidman and Foster [77] introduce the k-plex model which is the largest subgraph where each node is missing no more than $k-1$ edges to its neighbors in the subgraph. It is NP-hard to find all maximal k-plexes and the maximum k-plex. Wu and Pei [88] propose the first algorithm for maximal k-plex enumeration. Berlowitz *et* al. [13] present an algorithm which focuses on the efficiency guarantees in terms of input-output complexity. The algorithm runs with polynomial delay when k is a constant and in FPT incremental polynomial time when k is a parameter.

In the literature, connectivity is another widely used metric in subgraph cohesiveness. A connected graph is k-edge-connected if it remains connected after removing any $k-1$ edges. Given a graph, a k-edge-connected component is a maximal subgraph which is k-edge-connected. Zhou *et* al. [101] propose an efficient algorithm to find all k-edge connected components. Chang *et* al. [20] solve the k-edge connected component computation problem based on a graph decomposition paradigm.

A generalized model, namely k-(r,s)-nucleus, is proposed by [74] to understand the distribution of existing models in the graph, and ideally determine relationships among them. The authors [74] introduce the following definitions:

Definition 1. *Let $r < s$ be positive integers and $\mathcal{S}$ be a set of K_ss in G.*

- *$K_r(\mathcal{S})$ is the set of K_rs contained in some $S \in \mathcal{S}$.*
- *The number of $S \in \mathcal{S}$ containing $R \in K_r(\mathcal{S})$ is the $\mathcal{S}$-degree of that K_r.*
- *Two K_rs R, R' are $\mathcal{S}$-connected if there exists a sequence $R = R_1, R_2, \ldots, R_k = R'$ in $K_r(\mathcal{S})$ such that for each i, some $S \in \mathcal{S}$ contains $R_i \cup R_{i+1}$.*

Definition 2. *Let k, r, and s be positive integers such that $r < s$. A k-(r, s) nucleus is a maximal union $\mathcal{S}$ of K_ss such that:*

- *The $\mathcal{S}$-degree of any $R \in K_r(\mathcal{S})$ is at least k.*
- *Any $R, R' \in K_s(\mathcal{S})$ are $\mathcal{S}$-connected.*

The authors show that the (r, s)-nucleus form a hierarchical decomposition of a graph. Based on the above definition, a k-(r, s) nucleus is a k-core when $r = 1, s = 2$, and is a k-truss when $r = 2, s = 3$. An algorithm similar to core decomposition is proposed to perform the nucleus decomposition using a peeling process. Their recent work [73] leverages the disjoint-set forest data structure to efficiently construct the hierarchy during traversal. It tracks the disconnected substructures which occurs in the same node of the hierarchy nucleus tree, and shows the technique works for any peeling algorithms.

2.2 Social Network Components

A large body of social networks components have been studied in sociology community. In this thesis, the objective is to consider the popular components in cohesive subgraph models to better accommodate real-life applications. In the following, some fundamental social components are introduced.

2.2.1 User Engagement

Evaluating the activity of user engagement is crucial for social networking sites to improve user stickiness and avoid the collapse of network. Malliaros and Vazirgiannis [65] verify that the degeneration property of k-core can be used to quantify engagement dynamics in real social networks. Garcia *e*t al. [38] use core decomposition to explain the decline of Friendster, which was popular mainly in

America and Asia at early 21st century. They assert that the network collapse of Friendster is due to the core number threshold, which determines the engagement of a user, steadily increased. Under this assumption, their model reasonably explain the time evolution of the active user number in Friendster. Seki and Nakamura [78] explain that the mechanism in the collapse of Friendster by use of an individual-level model from Cannarella *e*t al. [17], and show the collapse starts from the center of the core structure. Ugander *e*t al. [82] emphasize that the neighborhood structure hypothesis has formed the underpinning of essentially all current models for social contagion. They find that the probability of contagion is tightly controlled by the number of friends in current subgraph, like k-core or k-truss, rather than by the actual number of the friends in the graph. Bhawalkar and Kleinberg *e*t al. [14] use the game-theory to show the network unraveling process stops when the remaining engaged individuals correspond to the k-core of the network.

2.2.2 User Similarity

The similarity among users has been widely used in many applications, such as item recommendation [75, 64], link prediction [32, 91] and classification [56]. Anderson *e*t al. [10] show that considering the similarity in characteristics of users can significantly enhance the analysis of user-to-user evaluations. They identify some fundamental rules for how similarity affects evaluations, and how similarity acts together with relative status. Then they propose new methods to demonstrate how community judgments can be more accurately retrieved from a small set of evaluations together with the similarity levels among users. Li *e*t al. [59] propose a graph-based framework to geographically mine the similarity between users based on their location histories. The framework considers both the sequence property of user movement behaviors and the hierarchy property

of geographic spaces to effectively and consistently model each user's location history and mine user similarities. Liu *et* al. [61] present a new user similarity model to improve the recommendation performance when only a few ratings are available for computing the similarities among users. Their model combines the local context for common ratings of each user pair and global preference of each user rating. Cheung and Tian [24] use machine learning techniques to retrieve the optimal user similarity measure and user rating styles for improving the accuracy of collaborative recommendation. By minimizing a prediction error term, they propose several transformation functions for modeling the rating styles, and present the corresponding learning algorithms.

2.2.3 Tie Strength

Tie strength, introduced by Granovetter [44], is a fundamental social network characteristic and has been well studied in sociology communities. Recently, many social network researchers show that the strength of ties is still a tenable theory on social networks nowadays, such as Facebook and Twitter [100]. Onnela *et* al. [69] give a counterintuitive consequence that social networks are robust to the removal of the strong ties but collapsed on the phase when some weak ties have been removed. They observe a coupling between tie strengths and the local structure, and show that this coupling significantly slows down the information propagation process. Bakshy *et* al. [11] examine the relative roles between strong and weak ties in information diffusion. They show that although strong ties may be more influential towards individuals, the effect of strong ties is not large enough to substitute the abundant information from weak ties in social networks.

Gilbert and Karahalios [42] present a predictive model that maps social media data to tie strength for distinguishing strong and weak ties. They show

that the model can enhance social media design elements, including friend recommendation, message routing, privacy controls and information prioritization. Sintos and Sintos [81] use the principle of strong triadic closure to characterize the tie strength in social networks. They study the problem of labeling the ties by strong or weak so as to enforce the strong triadic closure property. Rotabi *et* al. [71] show that most existing methods for strong tie detection are based on structural information, especially on triangles, following the ideas from sociology. They experimentally demonstrate that using only structural network features is sufficient for strong tie detection with high precision. With the minimum triangle number of $k-2$ for each edge, the k-truss [29] can be regarded as a strong tie community where each edge in the community is a strong tie.

2.3 Mining Attributed Graphs

It is common to use attributed graphs under various scenarios for real-world social network studies on both research and industry [37, 51]. A large amount of classical graph queries have been investigated on attributed graphs such as clustering [92], community detection [93], and network modeling [51]. Huang *et* al. [49] survey the state-of-the-art of community search on various kinds of graphs including attributed graphs. To the best of our knowledge, the (k,r)-core work [96] in this thesis is the first work to advocate a general cohesive subgraph model to consider both user engagement and attribute similarity on various kinds of attributed graphs.

Dang *et* al. [31] and Yang *et* al. [93] devised clustering methods which considers both structural and attribute similarities to detect communities. Because their detected communities are from clustering functions and do not have obvious structure and similarity characteristics, their method cannot solve our problem.

Recently, there are some investigations on the problem of cohesive subgraph computation on attributed graphs. Wu *et* al. [90] developed efficient algorithms to find dense and connected subgraphs in dual networks. Nevertheless, their model is inherently different to our (k,r)-core model because they only consider the cohesiveness of the dual graph (i.e., the attribute similarity in our problem) based on densest subgraph model. The connectivity constraint on graph structure alone cannot reflect the structure cohesiveness of the graph. Since their result only contains the subgraph which is connected on the original graph and is densest on the dual graph, their result excludes some cohesive subgraphs where vertices have high engagement and similarity. Consequently, their approach cannot be applied to solving our problem. Zhu *et* al. [103] studied finding the k-core within a given spatial region and contains a query point. Their approach is designed for community search on geo-social networks while we detect communities considering similarity of various attributes instead of a specific region. Fang *et* al. [36] proposed algorithms to find a subgraph related to a query point considering cohesiveness on both structure and keyword similarity, while it focuses on maximizing the number of common keywords of the vertices in the subgraph. Their recent work [35] aims to find a connected cohesive subgraph which satisfies a minimum degree of k for every vertex in the subgraph and has the minimum spatial radius. Huang and Lakshmanan [48] study the query of attributed truss communities which are connected and close k-truss subgraphs containing the query points, with the largest attribute relevance score.

The techniques developed in related papers have not been found applicable to solving our (k,r)-core problems. In this thesis, we show that the computation of (k,r)-core is much more challenging than the individual computation of k-core and clique. Moreover, our empirical study shows that it is less efficient to compute (k,r)-core by sequentially applying the k-core and clique techniques.

Chapter 3

Cohesive Subgraph Discovery

3.1 Introduction

Mining cohesive subgraphs is one of the most fundamental graph problems to find communities with high interaction among users. In the chapter, we model the social networks as *attributed graphs* to discover cohesive subgraphs on real-life social data considering various user attributes. In the attributed graphs, users are represented by vertices, friendship are represented by edges, and vertex attribute is associated with specific properties, such as locations or keywords. We move beyond the simple structure-based cohesive subgraph models and advocate a complicated but more realistic cohesive subgraph model on attributed graphs, namely (k,r)-core, to accommodate two intuitive and important criteria: *engagement* and *similarity*.

We adopt the k-core model on the graph structure, where each vertex in the subgraph has at least k neighbors (*structure constraint*). The similarity of two users can be derived from a given set of attributes (e.g., location, interests, and user generated content), which varies in different scenarios. By connecting two users (vertices) whose similarity exceeds a given threshold r, a *similarity*

graph is derived to capture the similarities among users. Specifically, we adopt the well-known *clique* model to capture the cohesiveness of users from similarity perspective; that is, the vertices of a cohesive subgraph in this chapter form a clique on the similarity graph, which can ensure pairwise similarity among users (*similarity constraint*).

To capture both engagement and similarity, we introduce the (k,r)-core model which is defined as follows. We say a connected subgraph of G is a (k,r)-core if and only if it satisfies both structure and similarity constraints. More specifically, given an attributed graph G, a (k,r)-core is a k-core of G (*structure constraint*) and the vertex set of the (k,r)-core induces a clique on the corresponding similarity graph (*similarity constraint*). A (k,r)-core is maximal if none of its supergraphs is a (k,r)-core. Usually, the groups with larger size have more social impact and influence, and hence attract more attention from the social network companies. Consequently, we focus on efficiently enumerating the maximal (k,r)-cores and finding the maximum (k,r)-core. Using the model of (k,r)-core can discover interesting groups, which are promising to become stable and active, to significantly enhance user stickiness and experience. In the following, we present two examples to motivate the (k,r)-core problems.

Example 1 (**Interest-Based Social Groups**). *In social networks such as Facebook and Weibo, the friendship information and mutual interests are widely used to recommend existing groups to users. The model (k,r)-core can be used for group discovery, which recommends new promising groups to relevant users. In Figure 3.1, we use a set of keywords to describe the interests of each user (e.g., Facebook user). Jaccard similarity metric can be employed to measure the user similarity. Suppose $k = 3$ and the similarity threshold $r = 0.5$, the group within red circle is a maximal (k,r)-core* [1]*. In this group, each user has at least*

[1]Note that the values of k and r can be tuned or learned for different requirements.

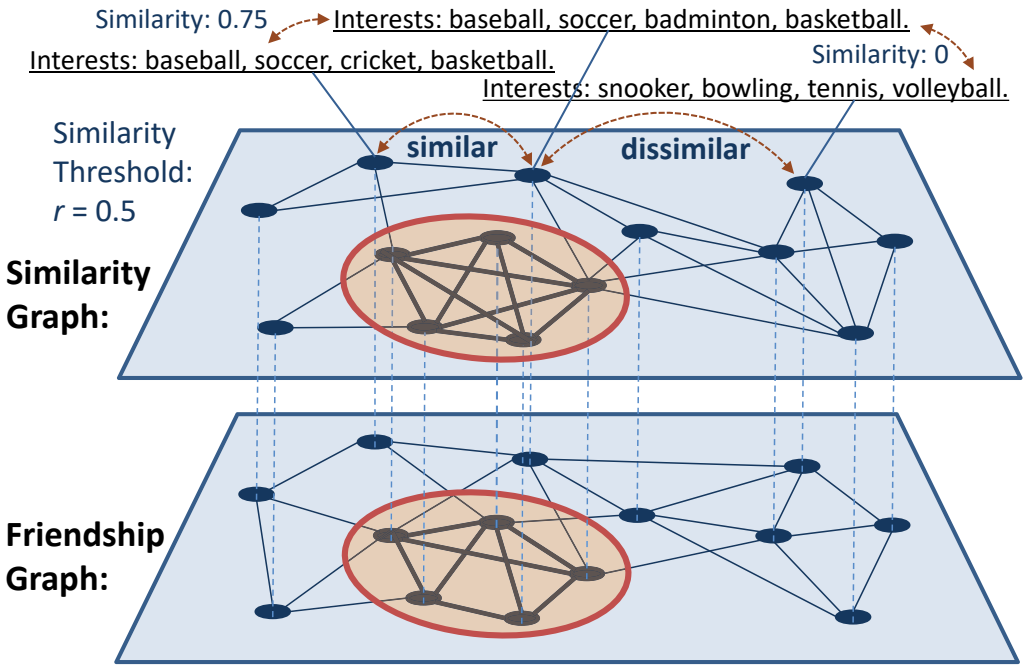


Figure 3.1: Group by Friendship and Interests

3 *friends and their interests are similar to others. This group is likely to remain stable, become active and enhance the interactions among users.*

Example 2 (Location-Based Game Teams). *By recommending game teams (groups) with potential to become sustainable and active, the game companies can greatly enhance user stickiness and improve game experience [21]. In many location-based online games such as Pokemon Go and Ingress, users play the games based on their locations and surroundings. For Pokemon Go, players would like to play the game with a group of people, in which there are some friends, to catch pokemons and attack gyms together. Because it is a location-based mobile game, players usually play it within a geographical range of frequently visited places such as their homes. Due to the diverse distribution of pokemons and pokestops, the places near the home of every player are likely to be visited, which naturally requires pairwise closeness (i.e., similarity) among group*

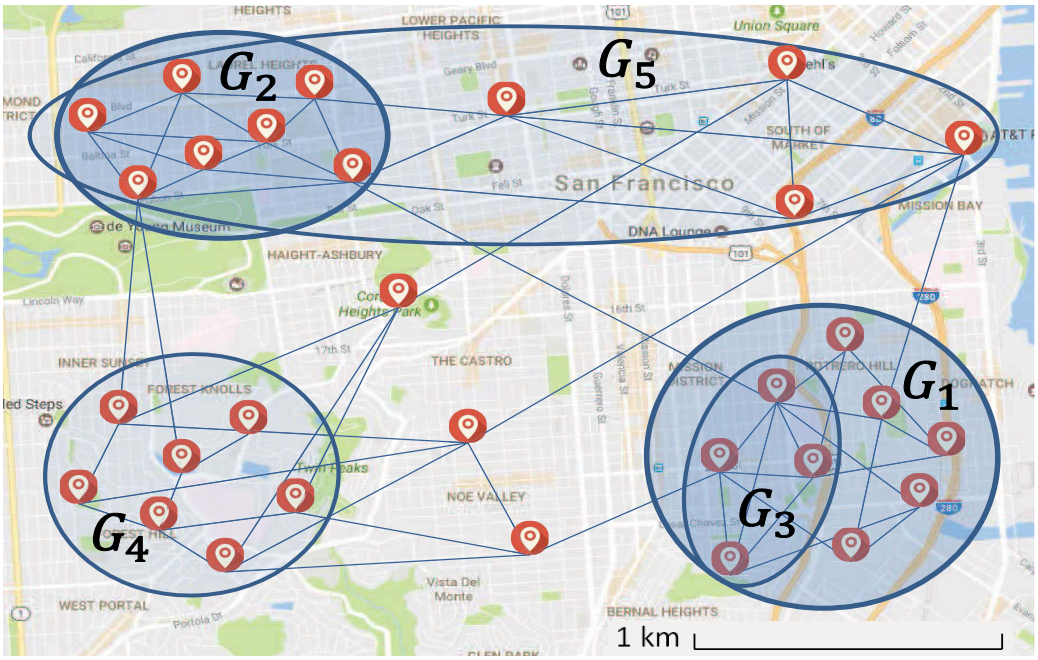


Figure 3.2: Group by Friendship and Locations

members. Consequently, (k,r)-cores can be good candidate groups where user engagement and similarity are guaranteed. With similar rationale, (k,r)-cores are useful to party games (e.g., Werewolf) which may be hosted at the homes of the team players.

As illustrated in Figure 3.2, we can model the players, their friendship and locations as a graph on the map. Suppose $k = 3$ *and the distance (similarity) threshold* $r = 1$ *km,* G_4 *and* G_5 *are not good candidate groups. Although each player pair in* G_4 *has close distance, their friendship is weak. Likewise, although each player in* G_5 *has at least k friends, some players cannot conveniently play with others because they are far away from others. However, maximal (k,r)-cores (i.e.,* G_1 *and* G_2*) can effectively identify good candidate groups because each player has at least k friends in the same group, and the distance for every two players is at most r (i.e., they are similar). Note that although* G_3 *is also a (k,r)-core, it is less interesting because it is contained by a larger group,* G_1*.*

In this chapter, we prove that the problem of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core are both NP-hard because of the similarity constraint involved. A straightforward solution is the combination of the existing k-core and clique algorithms, e.g., enumerating the cliques on similarity graph and then checking the structure constraint on the graph. We show that this is not promising because of the isolated processing of structure and similarity constraints. The performance can be immediately improved by considering two constraints (i.e., two pruning rules) at the same time without explicitly materializing the similarity graph. Then our technique contributions focus on further reducing the search space of two mining algorithms from the following three aspects: (i) effective pruning, early termination and maximal check techniques. (ii) (k,k')-core based approach to derive tight upper bound for the problem of finding the maximum (k,r)-core. and (iii) good search orders in two mining algorithms.

Road Map. Section 3.2 formally defines the (k,r)-core problems and analyze the complexity. Section 3.3 presents a baseline solution. Section 3.4 warms up our solution. Section 3.5 presents the algorithms for enumerating the maximal (k,r)-cores. Section 3.6 presents the algorithms for finding the maximum (k,r)-core. Section 3.7 introduces the search orders in proposed algorithms. Section 3.8 experimentally evaluate all the algorithms. Section 3.9 concludes the chapter.

3.2 Preliminaries

In this section, we first formally introduce the concept of (k,r)-core, then show that the two problems are NP-hard. Table 3.1 summarizes the mathematical notations used throughout this chapter.

Table 3.1: The summary of notations

Notation	Definition
G	a simple attributed graph
S,J,R	induced subgraphs
u, v	vertices in the attributed graph
$sim(u, v)$	similarity between u and v
$deg(u, S)$	number of adjacent vertex of u in S
$deg_{min}(S)$	minimal degree of the vertices in S
$DP(u, S)$	number of dissimilar vertices of u w.r.t S
$DP(S)$	number of dissimilar pairs of S
$SP(u, S)$	number of similar vertices of u w.r.t S
M	vertices chosen so far in the search
C	candidate vertices set in the search
E	relevant exclusive vertices set in the search
$\mathcal{R}(M,C)$	maximal (k,r)-cores derived from $M \cup C$
$SF(S)$ (i.e.,$SF_C(S)$)	every u in S with $DP(u, C) = 0$
$SF_{C\cup E}(S)$	every u in S with $DP(u, C \cup E) = 0$

3.2.1 Problem Definition

We consider an undirected, unweighted, and attributed graph $G = (V, \mathcal{E}, A)$, where $V(G)$ (resp. $\mathcal{E}(G)$) represents the set of vertices (resp. edges) in G, and $A(G)$ denotes the attributes of the vertices. By $sim(u, v)$, we denote the similarity of two vertices u, v in $V(G)$ which is derived from their corresponding attribute values (e.g., users' geo-locations and interests) such as Jaccard similarity or Euclidean distance. For a given similarity threshold r, we say two vertices are dissimilar (resp. similar) if $sim(u, v) < r$ (resp. $sim(u, v) \geq r$) [2].

For a vertex u and a set S of vertices, $DP(u, S)$ (resp. $SP(u, S)$) denotes the number of other vertices in S which are dissimilar (resp. similar) to u regarding the given similarity threshold r. We use $DP(S)$ to denote the number of dissimilar pairs in S. We use $S \subseteq G$ to denote that S is a subgraph of G where

[2]Following the convention, when the distance metric (e.g., Euclidean distance) is employed, we say two vertices are similar if their distance is not larger than the given distance threshold.

$\mathcal{E}(S) \subseteq \mathcal{E}(G)$ and $A(S) \subseteq A(G)$. By $deg(u, S)$, we denote the number of adjacent vertices of u in $V(S)$. Then, $deg_{min}(S)$ is the minimal degree of the vertices in $V(S)$. Now we formally introduce two constraints, namely *structure constraint* and *similarity constraint*, which describe the cohesiveness of the vertices of an attributed subgraph from graph structure and vertices attribute perspectives, respectively.

Definition 3. ***Structure Constraint.*** *Given a positive integer k, a subgraph S satisfies the structure constraint if $deg(u, S) \geq k$ for each vertex $u \in V(S)$, i.e., $deg_{min}(S) \geq k$.*

Definition 4. ***Similarity Constraint.*** *Given a similarity threshold r, a subgraph S satisfies the similarity constraint if $DP(u, S) = 0$ for each vertex $u \in V(S)$, i.e., $DP(S) = 0$.*

We then formally define the (k,r)-core based on structure and similarity constraints.

Definition 5. (**k**,**r**)-***core****. Given a connected subgraph $S \subseteq G$, S is a (k,r)-core if S satisfies both structure and similarity constraints.*

In this chapter, we aim to find all maximal (k,r)-cores and the maximum (k,r)-core, which are defined as follows.

Definition 6. ***Maximal*** (**k**,**r**)-***core****. Given a connected subgraph $S \subseteq G$, S is a maximal (k,r)-core if S is a (k,r)-core of G and there exists no (k,r)-core S' of G such that $S \subset S'$.*

Definition 7. ***Maximum*** (**k**,**r**)-***core****. Let $\mathcal{R}$ denote all (k,r)-cores of an attributed graph G, a (k,r)-core $S \subseteq G$ is maximum if $|V(S)| \geq |V(S')|$ for every (k,r)-core $S' \in \mathcal{R}$.*

Problem Statement. Given an attributed graph G, a positive integer k and a similarity threshold r, we aim to develop efficient algorithms for the following two fundamental problems: (i) enumerating all maximal (k,r)-cores in G; (ii) finding the maximum (k,r)-core in G.

Example 3. *In Figure 3.2, all vertices are from the k-core where $k = 3$. G_1, G_2 and G_3 are the three (k,r)-cores. G_1 and G_2 are maximal (k,r)-cores while G_3 is fully contained by G_1. G_1 is the maximum (k,r)-core.*

3.2.2 Problem Complexity

We can compute k-core in linear time by recursively removing the vertices with a degree of less than k [12]. Nevertheless, the two problems studied in this chapter are NP-hard due to the additional similarity constraint.

Theorem 1. *Given a graph $G(V, \mathcal{E})$, the problems of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core are NP-hard.*

Proof. Given a graph $G(V, \mathcal{E})$, we construct an attributed graph $G'(V', \mathcal{E}', A')$ as follows. Let $V(G') = V(G)$ and $\mathcal{E}(G') = \{(u, v) \mid u \in V(G'), v \in V(G'), u \neq v\}$, i.e., G' is a complete graph. For each $u \in V(G')$, we let $A(u) = adj(u, G)$ where $adj(u, G)$ is the set of adjacent vertices of u in G. Suppose a Jaccard similarity is employed, i.e., $sim(u, v) = \frac{|A(u) \cap A(v)|}{|A(u) \cup A(v)|}$ for any pair of vertices u and v in $V(G')$, and let the similarity threshold $r = \epsilon$ where ϵ is an infinite small positive number (e.g., $\epsilon = \frac{1}{2|V(G')|}$). We have $sim(u, v) \geq r$ if the edge $(u, v) \in \mathcal{E}(G)$, and otherwise $sim(u, v) = 0 < r$. Since G' is a complete graph, i.e., every subgraph $S \subseteq G'$ with $|S| \geq k$ satisfies the structure constraint of a (k,r)-core, the problem of deciding whether there is a k-clique on G can be reduced to the problem of finding a (k,r)-core on G' with $r = \epsilon$, and hence can be solved by the problem of enumerating all maximal (k,r)-cores or finding the

maximum (k,r)-core. Theorem 1 holds due to the NP-hardness of the k-clique problem [40]. □

Following theorem indicates that the maximal (k,r)-core enumeration problem studied in this chapter is harder than the maximal clique enumeration problem in the sense that there does not exist a polynomial delay or polynomial total algorithm for the maximal (k,r)-core enumeration problem while exist for the maximal clique enumeration [19].

Theorem 2. *There does not exist a polynomial delay or polynomial total algorithm for the problem of enumerating all maximal (k,r)-cores unless P=NP.*

Proof. First, according to Theorem 1, it is impossible to find a (k,r)-core in polynomial time of input size during enumeration unless P=NP. Consequently, there is no polynomial delay algorithm for maximal (k,r)-core enumeration. Second, suppose on the contrary that there is a polynomial total (total time polynomial to input + output size) algorithm for the maximal (k,r)-core enumeration problem, we can derive that when there is no (k,r)-core, the algorithm can terminate in time polynomial to input size. This contradicts Theorem 1. Therefore, Theorem 2 holds. Because the size of a (k,r)-core is at most the input size, time complexity towards output size can not be polynomial. There is also no polynomial total (total time polynomial to input + output size) algorithm. □

Remark 1. *It might be confusing that k-clique decision problem is NP-complete while maximal clique enumeration has a polynomial delay algorithm. Actually, the NP-completeness of k-clique decision problem holds if and only if the size of clique is at least k. However, the clique size in maximal clique enumeration can be arbitrary. It is to say, deciding whether there is a maximal clique in maximal clique enumeration can be solved in polynomial time. Consequently, our complexity proof does not conflict with the hardness properties of clique problems.*

3.3 The Clique-based Approach

Let G' denote a new graph named *similarity graph* with $V(G') = V(G)$ and $\mathcal{E}(G') = \{(u, v) \mid sim(u, v) \geq r \ \& \ u, v \in V(G)\}$, i.e., G' connects the similar vertices in $V(G)$. Then, the set of vertices in a (k,r)-core satisfies the structure constraint on G and is a *clique* (i.e., a complete subgraph) on the similarity graph G' (because every vertex pair is similar in a (k,r)-core). This implies that we can use the existing clique algorithms on the similarity graph to enumerate the (k,r)-core candidates, followed by a structure constraint check. More specifically, we may first construct the similarity graph G' by computing the pairwise similarity of the vertices. Then we enumerate the cliques in G', and compute the k-core on each induced subgraph of G for each clique. We can find the maximal (k,r)-cores after the maximal check. We may further improve the performance of this clique-based approach in the following three ways.

- Instead of enumerating cliques on the similarity graph G', we can first compute the k-core of G, denoted by S. Then, we apply the clique-based method on the similarity graph of each connected subgraph in S.
- An edge in S can be deleted if its corresponding vertices are *dissimilar*, i.e., there is no edge between these two vertices in similarity graph S'.
- We only need to compute k-core for each maximal clique because any maximal (k,r)-core derived from a non-maximal clique can be obtained from the maximal cliques.

The above three methods substantially improve the performance of the clique-based approach. Nevertheless, our experiments later will demonstrate that the improved clique-based approach is substantially outperformed by our baseline algorithm (Section 3.8.3), although the state-of-the-art k-core and clique computation methods have been applied [12, 84].

Algorithm 1: **EnumerateMKRC**(G, k, r)

Input : G : attributed graph, k : degree threshold, r : similarity threshold
Output: $\mathcal{M}$: Maximal (k,r)-cores

```
1 for each edge (u, v) in E(G) do
2     Remove edge (u, v) from G If sim(u, v) < r;
3 S ← k-core(G); R := ∅;
4 for each connected subgraph S in S do
5     R := R ∪ NaiveEnum(∅, S);
6 for each (k,r)-core R in R do
7     if there is a (k,r)-core R' ∈ R s.t. R ⊂ R' then
8         R := R \ R;
9 return R
```

Algorithm 2: **NaiveEnum**(M, C)

Input : M : chosen vertices, C : candidate vertices
Output: $\mathcal{R}$: (k,r)-cores

```
1 if C = ∅ and deg_min(M) ≥ k and DP(M) = 0 then
2     R := R ∪ R for every connected subgraph R ∈ M;
3 else
4     u ← choose a vertex in C;
5     NaiveEnum(M ∪ u, C \ u);          /* Expand */;
6     NaiveEnum(M, C \ u );             /* Shrink */;
```

3.4 Warming Up for Our Approach

For ease of understanding, we start with a straightforward set enumeration approach . The pseudo-code is given in Algorithm 1. At the initial stage (Line 1-2), we remove the edges in $\mathcal{E}(G)$ whose corresponding vertices are dissimilar, and then compute the k-core $\mathcal{S}$ of the graph G. For each connected subgraph $S \in \mathcal{S}$, the procedure NaiveEnum (Line 5) identifies all possible (k,r)-cores by enumerating and validating all the induced subgraphs of S. By $\mathcal{R}$, we record the (k,r)-cores seen so far. Lines 6-8 eliminate the non-maximal (k, r)-cores.

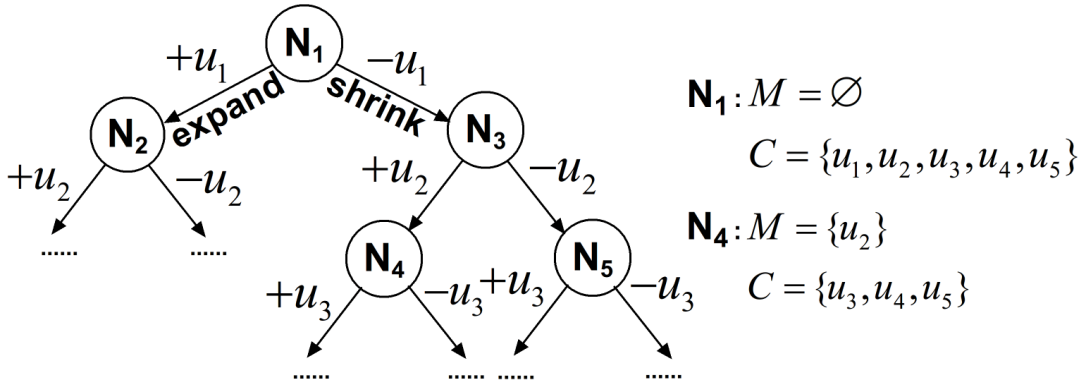


Figure 3.3: Example of the Search Tree

During the NaiveEnum search procedure (Algorithm 2), the vertex set M incrementally retains the chosen vertices, and C retains the candidate vertices. As shown in Figure 3.3, the enumeration process corresponds to a *binary search tree* in which each leaf node represents a subset of S. In each non-leaf node, there are two branches. The chosen vertex will be moved to M from C in *expand branch*, and will be deleted from C in *shrink branch*, respectively. In order to enumerate all possible solutions, for each chosen vertex u from C at Line 4, we will try to extend M with u (Line 5) or explicitly discard u (Line 6). Then each subset $R \subseteq S$ is validated according to the structure, similarity and connectivity constraints at Line 1. Algorithm 1 immediately finds the maximum (k,r)-core by returning the maximal (k,r)-core with the largest size.

Algorithm Correctness. We can safely remove dissimilar edges (i.e., edges whose corresponding vertices are dissimilar) at Line 1 and 2, since they will not be considered in any (k,r)-core due to the similarity constraint. For every (k,r)-core R in G, there is one and only one connected k-core subgraph S from $\mathcal{S}$ with $R \subseteq S$. Since all possible subsets of S (i.e., $2^{|S|}$ leaf nodes) are enumerated in the corresponding search tree, every (k,r)-core R can be accessed *exactly once* during the search. Together with the structure/similarity constraints and maximal property validation, we can output all maximal (k,r)-cores.

3.5 Finding All Maximal (k,r)-Cores

In this section, we propose pruning techniques for the enumeration algorithm including candidate reduction, early termination, and maximal check techniques, respectively. Note that we defer discussion on search orders to Section 3.7.

3.5.1 Reducing Candidate Size

We present pruning techniques to explicitly/implicitly exclude some vertices from C.

Eliminating Candidates

Intuitively, when a vertex in C is assigned to (i.e., expand branch) M or discarded (i.e., shrink branch), we shall recursively remove some non-promising vertices from C due to structure and similarity constraints. The following two pruning rules are based on the definition of (k,r)-core.

Theorem 3. ***Structure based Pruning****. We can discard a vertex u in C if $deg(u, M \cup C) < k$.*

Theorem 4. ***Similarity based Pruning****. We can discard a vertex u in C if $DP(u, M) > 0$.*

Candidate Pruning Algorithm. If a chosen vertex u is extended to M (i.e., to the expand branch), we first apply the similarity pruning rule to exclude vertices in C which are dissimilar to u. Otherwise, none of the vertices will be discarded by the similarity constraint when we follow the shrink branch. Due to the removal of the vertices from C (expand branch) or u (shrink branch), we conduct structure based pruning by computing the k-core for vertices in $M \cup C$. Note that the search terminates if any vertex in M is discarded.

It takes at most $O(|C|)$ time to find dissimilar vertices of u from C. Due to the k-core computation, the structure based pruning takes linear time to the number of edges in the induced graph of $M \cup C$.

After applying the candidate pruning, following two important invariants always hold at each search node.

Similarity Invariant. We have

$$DP(u, M \cup C) = 0 \text{ for every vertex } u \in M \tag{3.1}$$

That is, M satisfies similarity constraint regarding $M \cup C$.

Degree Invariant. We have

$$deg_{min}(M \cup C) \geq k \tag{3.2}$$

That is, M and C together satisfy the structure constraint.

Retaining Candidates

In addition to explicitly pruning some non-promising vertices, we may implicitly reduce the candidate size by not choosing some vertices from C. In this chapter, we say a vertex u is *similarity free* w.r.t C if u is similar to all vertices in C, i.e., $DP(u, C) = 0$. By $SF(C)$ we denote the set of similarity free vertices in C.

Theorem 5. *Given that the pruning techniques are applied in each search step, we do not need to choose vertices from $SF(C)$ on both expand and shrink branches. Moreover, $M \cup C$ is a (k,r)-core if we have $C = SF(C)$.*

Proof. For every vertex $u \in SF(C)$, we have $DP(u, M \cup C) = 0$ due to the similarity invariant of M (Equation 3.1) and the definition of $SF(C)$. Let M_1 and C_1 denote the corresponding chosen set and candidate after u is chosen for

expansion. Similarly, we have M_2 and C_2 if u is moved to the shrink branch. We have $M_2 \subset M_1$ and $C_2 \subseteq C_1$, because there are no discarded vertices when u is extended to M while some vertices may be eliminated due to the removal of u in the shrink branch. This implies that $\mathcal{R}(M_2, C_2) \subseteq \mathcal{R}(M_1, C_1)$. Consequently, we do not need to explicitly discard u as the shrink branch of u is useless. Hence, we can simply retain u in C in the following computation.

However, $C = SF(C)$ implies every vertex u in $M \cup C$ satisfies the similarity constraint. Moreover, u also satisfies the structure constraint due to the degree invariant (Equation 3.2) of $M \cup C$. Consequently, $M \cup C$ is a (k,r)-core. □

Note that a vertex $u \in SF(C)$ may be discarded in the following search due to the structural constraint. Otherwise, it is moved to M when the condition $SF(C) = C$ holds. For each vertex u in C, we can update $DP(u, C)$ in a passive way when its dissimilar vertices are eliminated from the computation. Thus, it takes $O(n_d)$ time in the worst case where n_d denotes the number of dissimilar vertex pairs in C.

Remark 2. *With similar rationale, we can move a vertex u directly from C to M if it is similarity free (i.e., $u \in SF(C)$) and is adjacent to at least k vertices in M. As this validation rule is trivial, it will be used in this chapter without further mention.*

Example 4. *In Figure 3.4, initially we have $M = \{u_6\}$ and $C = \{u_1, \ldots, u_5, u_7,$ $\ldots, u_{10}\}$. We use the spatial distance of two vertices as their similarity, and the only dissimilar pair is u_4 and u_7. Suppose u_7 is chosen from C, following the expand branch, u_7 will be moved from C to M and then u_4 will be pruned due to the similarity constraint. Then we need to prune u_8 as $deg(u_8, M \cup C) < 3$. Thus, $M = \{u_6, u_7\}, E = \{u_4, u_8\}$. Since we have $SF(C) = C$, this search branch is terminated and we get a (k,r)-core of $M \cup C$. Regarding the shrink branch, u_7 is*

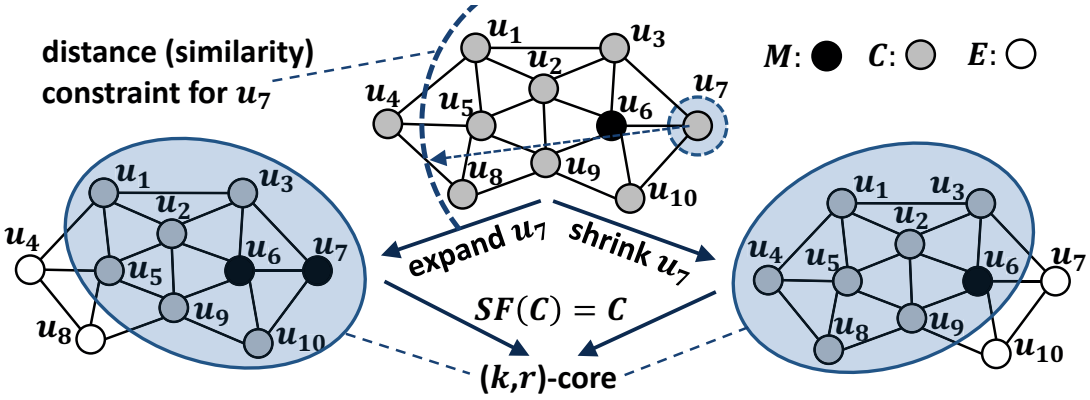


Figure 3.4: Pruning and Retaining Candidates

moved from C to E, which leads to the deletion of u_{10} due to structure constraint. Thus, $M = \{u_6\}, E = \{u_7, u_{10}\}$. Since we have $SF(C) = C$, this search branch is terminated and we get a (k,r)-core of $M \cup C$.

3.5.2 Early Termination

Trivial Early Termination. There are two trivial early termination rules. As discussed in Section 3.5.1, we immediately terminate the search if any vertex in M is discarded due to the structure constraint. We also terminate the search if M is disconnected to C. Both these stipulations will be applied in the remainder of this chapter without further mention.

In addition to identifying the subtree that cannot derive any (k,r)-core, we further reduce the search space by identifying the subtrees that cannot lead to any *maximal* (k,r)-core. By E, we denote the related excluded vertices set for a search node of the tree, where the discarded vertices during the search are retained if they are similar to M, i.e., $DP(v, M) = 0$ for every $v \in E$ and $E \cap (M \cup C) = \emptyset$. We use $SF_C(E)$ to denote the similarity free vertices in E w.r.t the set C; that is, $DP(u, C) = 0$ for every $u \in SF_C(E)$. Similarly, by $SF_{C \cup E}(E)$ we denote the similarity free vertices in E w.r.t the set $E \cup C$.

Theorem 6. ***Early Termination.*** *We can safely terminate the current search if one of the following two conditions hold:*
(i) there is a vertex $u \in SF_C(E)$ *with* $deg(u, M) \geq k$*;*
(ii) there is a set $U \subseteq SF_{C \cup E}(E)$*, such that* $deg(u, M \cup U) \geq k$ *for every vertex* $u \in U$*.*

Proof. (i) We show that every (k,r)-core R derived from current M and C (i.e., $R \subseteq \mathcal{R}(M, C)$) can reveal a larger (k,r)-core by attaching the vertex u. For any $R \in \mathcal{R}$, we have $deg(u, R) \geq k$ because $deg(u, M) \geq k$ and $M \subseteq V(R)$. u also satisfies the similarity constraint based on the facts that $u \in SF_C(E)$ and $R \subseteq M \cup C$. Consequently, $V(R) \cup \{u\}$ forms a (k,r)-core. (ii) The correctness of condition (ii) has a similar rationale. The key idea is that for every $u \in U$, u satisfies the structure constraint because $deg(u, M \cup U) \geq k$; and u also satisfies the similarity constraint because $U \subseteq SF_{E \cup C}(E)$ implies that $DP(u, U \cup R) = 0$. □

Early Termination Check. It takes $O(|E|)$ time to check the condition (i) of Theorem 6 with one scan of the vertices in $SF_C(E)$. Regarding condition (ii), we may conduct k-core computation on $M \cup SF_{C \cup E}(E)$ to see if a subset of $SF_{C \cup E}(E)$ is included in the k-core. The time complexity is $O(n_e)$ where n_e is the number of edges in the induced graph of $M \cup C \cup E$.

3.5.3 Checking Maximal

In Algorithm 1 (Lines 6-8), we need to validate the maximal property based on all (k,r)-cores of G. The cost increases with both the number and the average size of the (k,r)-cores. Similar to the early termination technique, we use the following rule to check the maximal property.

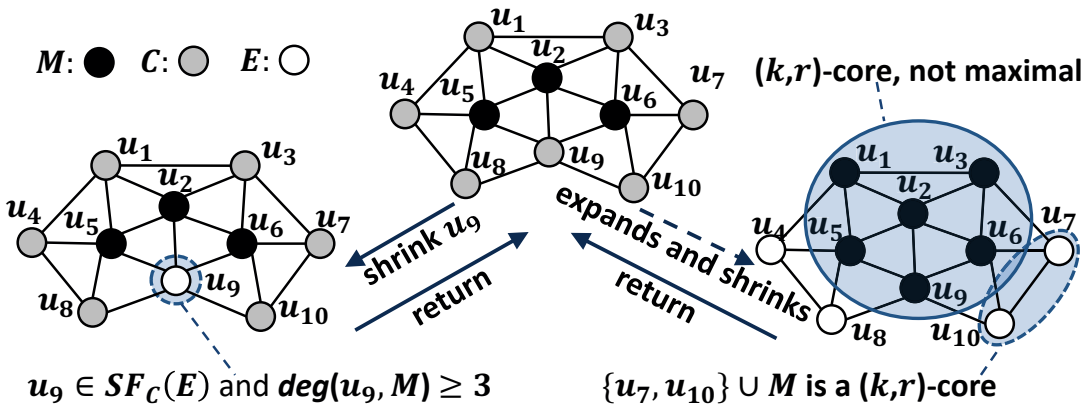


Figure 3.5: Early Termination and Check Maximal

Theorem 7. ***Checking Maximal.*** *Given a (k,r)-core R, R is a maximal (k,r)-core if there doesn't exist a non-empty set $U \subseteq E$ such that $R \cup U$ is a (k,r)-core, where E is the excluded vertices set when R is generated.*

Proof. E contains all discarded vertices that are similar to M according to the definition of the excluded vertices set. For any (k,r)-core R' which fully contains R, we have $R' \subseteq E \cup R$ because $R = M$ and $C = \emptyset$, i.e., the vertices outside of $E \cup R$ cannot contribute to R'. Therefore, we can safely claim that R is maximal if we cannot find R' among $E \cup R$. □

Example 5. *In Figure 3.5, we have $M = \{u_2, u_5, u_6\}$, $C = \{u_1, u_3, u_4, u_7, u_8, u_9, u_{10}\}$. u_4 and u_7 is the only dissimilar pair. If u_9 is chosen from C on the shrink branch, u_9 is moved from C to E. Because u_9 is similar to every vertex in C and has 3 neighbors in M, the search is terminated for $u_9 \in SF_C(E)$ and $deg(u_9, M) \geq 3$ according to Theorem 6. If we expand and shrink the initial graph several times, the graph becomes $M = \{u_1, u_2, u_3, u_5, u_6, u_9\}$ and $E = \{u_4, u_7, u_8, u_{10}\}$. Here M is a (k,r)-core, but we can further extend u_7 and u_{10} to M and get a larger (k,r)-core. According to Theorem 7, M is not a maximal (k,r)-core.*

Algorithm 3: **AdvancedEnum**(M, C, E)

Input : M : chosen vertices set, C : candidate vertices set, E : relevant excluded vertices set
Output: $\mathcal{R}$: maximal (k,r)-cores

```
1 Update C and E based on candidate pruning techniques (Theorem 3 and
  Theorem 4);
2 Return If current search can be terminated (Theorem 6);
3 if C = SF(C) (Theorem 5) then
4 |   M := M ∪ C;
5 |   R := R ∪ M If CheckMaximal(M, E) (Theorem 7);
6 else
7 |   u ← a vertex in C \ SF(C) (Theorem 5);
8 |   AdvancedEnum(M ∪ u, C \ u, E);
9 |   AdvancedEnum(M, C \ u, E ∪ u);
```

Since the maximal check algorithm is similar to our advanced enumeration algorithm, we delay providing the details of this algorithm to Section 3.5.4.

Remark 3. *The early termination technique can be regarded as a lightweight version of the maximal check, which attempts to terminate the search before a (k,r)-core is constructed.*

3.5.4 Advanced Enumeration Method

In Algorithm 3, we present the pseudo code for our advanced enumeration algorithm which integrates the techniques proposed in previous sections. We first apply the candidate pruning algorithm outlined in Section 3.5.1 to eliminate some vertices based on structure/similarity constraints. Along with C, we also update E by including discarded vertices and removing the ones that are not similar to M. Line 2 may then terminate the search based on our early termination rules. If the condition $C = SF(C)$ holds, $M \cup C$ is a (k,r)-core according to Theorem 5, and we can conduct the maximal check (Lines 3-5). Otherwise,

Algorithm 4: **CheckMaximal**(M, C)

Input : M : chosen vertices, C : candidate vertices
Output: $isMax$: true if M is a maximal (k,r)-core
1 Update C based on similarity and structure constraint;
2 **if** M is a (k,r)-core **then**
3 Exit the algorithm with $isMax = false$ *If* $|\mathbf{M}^*| < |M|$;
4 **else if** $|C| > 0$ **then**
5 $u \leftarrow$ a vertex in C;
6 **CheckMaximal**($M \cup u$, $C \setminus u$);
7 **CheckMaximal**(M, $C \setminus u$);

Lines 7-9 choose one vertex from $C \setminus SF(C)$ and continue the search following two branches, where the three sets M, C and E are updated accordingly.

Checking Maximal Algorithm. According to Theorem 7, we need to check whether some of the vertices in E can be included in the current (k,r)-core, denoted by M^*.This can be regarded as the process of further exploring the search tree by treating E as candidate C (Line 5 of Algorithm 3). Algorithm 4 presents the pseudo code for our maximal check algorithm.

To enumerate all the maximal (k,r)-cores of G, we need to replace the NaiveEnum procedure (Line 5) in Algorithm 1 using our advanced enumeration method (Algorithm 3). Moreover, the naive checking maximals process (Line 6-8) is not necessary since checking maximals is already conducted by our enumeration procedure (Algorithm 3). Since the search order for vertices does not affect the correctness, the algorithm correctness can be immediately guaranteed based on above analyses. It takes $O(n_e + n_d)$ times for each search node in the worst case, where n_e and n_d denote the total number of edges and dissimilar pairs in $M \cup C \cup E$.

Algorithm Correctness. Section 3.4 shows the correctness of the Algorithm 1 when none of our pruning techniques have been applied. Section 3.5 confirms

that our candidate size reduction and early termination techniques can safely exclude some vertices from further computation, i.e., we can reach all maximal (k,r)-cores on the pruned search tree. Each non-maximal (k,r)-core will be discarded by either the maximal check technique or the early termination technique. Our early termination and checking maximals techniques will remove all of the non-maximal (k,r)-cores. For every non-maximal (k,r)-core generated in the search tree node, our early termination and checking maximals techniques ensure that there is a (k,r)-core R with larger size according to the definition of maximal (k,r)-core. Since the search order for vertices does not affect the correctness. The algorithm correctness can be immediately guaranteed. Thus, the correctness of our enumeration algorithm follows.

Time Complexity. Let n_e and n_d denote the total number of edges and dissimilar pairs in $M \cup C \cup E$. According to the analysis of candidate size reduction and early termination techniques, it takes $O(n_e + n_d)$ times for each search node in the worst case. [3]

Another issue is the computation of the dissimilar pairs. To support general similarity metrics, we do not consider the indexing of the vertices attribute (e.g., R-tree and M-tree). Thus, we need to materialize the dissimilar pairs based on pair-wise computation. In our implementation, we compute the pair-wise similarity of the vertices for the following search when the first vertex u is inserted into M (i.e., $|M| = 1$) with a time complexity $O(n_v + n_p^2)$ where n_v is the number of vertices in S (Line 4 of Algorithm 1) and n_p is the number of similar pairs of u in S.

[3] We can regard the checking maximals processing as the continue of the search by attaching vertices of E to C.

3.6 Finding the Maximum (k,r)-core

In this section, we first introduce the upper bound based algorithm to find the maximum (k,r)-core. Then a novel (k, k')-core approach is proposed to derive tight upper bound of the (k,r)-core size. Finally, we show the proposed upper bound and algorithm can be applied to finding the top-m maximal (k,r)-cores.

3.6.1 Algorithm for Finding the Maximum One

Algorithm 5 presents the pseudo code for finding the maximum (k,r)-core, where R denotes the largest (k,r)-core seen so far. There are three main differences compared to the enumeration algorithm (Algorithm 3). (i) Line 2 terminates the search if we find the current search is non-promising based on the upper bound of the core size, denoted by $KRCoreSizeUB(M,C)$. (ii) We do not need to validate the maximal property. (iii) Along with the order of visiting the vertices, the order of the two branches also matters for quickly identifying large (k,r)-cores (Lines 6-12), which is discussed in Section 3.7.

To find the maximum (k,r)-core in G, we need to replace the NaiveEnum procedure (Line 5) in Algorithm 1 with the method in Algorithm 5, and remove the naive maximal check section of Algorithm 1 (Line 6-8). To quickly find a (k,r)-core with a large size, we start the algorithm from the subgraph S which holds the vertex with the highest degree. The maximum (k,r)-core is identified when Algorithm 1 terminates.

Algorithm Correctness. Since Algorithm 5 is essentially an enumeration algorithm with an upper bound based pruning technique, the correctness of this algorithm is clear if the $KRCoreSizeUB(M, C)$ at Line 2 is calculated correctly.

Time Complexity. As shown in Section 3.6.2, we can efficiently compute the upper bound of core size in $O(n_e + n_s)$ time where n_s is the number of similar

Algorithm 5: **FindMaximum**(M, C, E)

Input : M : chosen vertices set , C : candidate vertices set, E : relevant excluded vertices set

Output: R : the largest (k,r)-core seen so far

```
1  Update C and E; Early terminate if possible;
2  if KRCoreSizeUB(M, C) > |R| then
3      if C = SF(C) then
4          R := M ∪ C;
5      else
6          u ← choose a vertex in C \ SF(C);
7          if Expansion is preferred then
8              FindMaximum(M ∪ u, C \ u, E);
9              FindMaximum(M, C \ u, E ∪ u);
10         else
11             FindMaximum(M, C \ u, E ∪ u);
12             FindMaximum(M ∪ u, C \ u, E);
```

pairs w.r.t $M \cup C \cup E$. For each search node the time complexity of the maximum algorithm is same as that of the enumeration algorithm. In practice, the number of search nodes in the maximum algorithm is much less than the enumeration algorithm, because the maximum algorithm further prunes the subtrees by the upper bound of core size. It also avoids the checking maximal process.

3.6.2 Size Upper Bound of (k,r)-Core

We use R to denote the (k,r)-core derived from $M \cup C$. In this way, $|M| + |C|$ is clearly an upper bound of $|R|$. However, it is very loose because it does not consider the similarity constraint.

Recall that G' denotes a new graph that connects the similar vertices of $V(G)$, called *similarity graph*. By J and J', we denote the induced subgraph of vertices $M \cup C$ from graph G and the similarity graph G', respectively. Clearly, we have $V(J) = V(J')$. Because R is a *clique* on the similarity graph J' and the size of

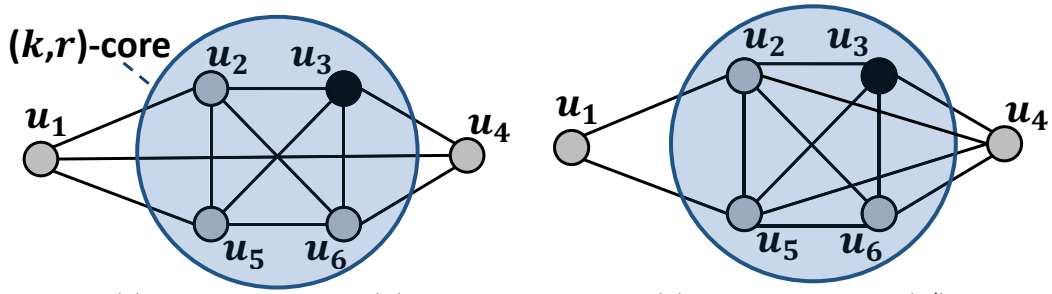
a k-clique is k, we can apply the maximum clique size estimation techniques to J' to derive the upper bound of $|R|$. Color [39] and k-core based methods [12] are two state-of-the-art techniques for maximum clique size estimation.

Color based Upper Bound. Let c_{min} denote the minimum number of colors to *color* the vertices in the similarity graph J' such that every two adjacent vertices in J' have different colors. Since a k-clique needs k number of colors to be *colored*, we have $|R| \leq c_{min}$. Therefore, we can apply graph coloring algorithms to estimate a small c_{min} [39].

***k*-Core based Upper Bound.** Let k_{max} denote the maximum k value such that k-core of J' is not empty. Since a k-clique is also a (k-1)-core, this implies that we have $|R| \leq k_{max} + 1$. Therefore, we may apply the existing k-core decomposition approach [12] to compute the maximal core number (i.e., k_{max}) on the similarity subgraph J'.

At the first glance, both the structure and similarity constraints are used in the above method because J itself is a k-core (structure constraint) and we consider the k_{max}-core of J' (similarity constraint). The upper bound could be tighter by choosing the smaller one from color based upper bound and k-core based upper bound. Nevertheless, we observe that the vertices in k_{max}-core of J' may not form a k-core on J since we only have J itself as a k-core. If so, we can consider k_{max}-1 as a tighter upper bound of R. Repeatedly, we have the largest k_{max}-i as the upper bound such that the corresponding vertices form a k-core on J and a (k_{max}-i)-core on J'. We formally introduce this (k, k')-core based upper bound in the following.

(*k,k'*)-Core based Upper Bound. We first introduce the concept of (k,k')-core to produce a tight upper bound of $|R|$. Theorem 8 shows that we can derive the upper bound for any possible (k,r)-core R based on the largest possible k' value, denoted by k'_{max}, from the corresponding (k,k')-core.



(a) Friendship Graph (J) (b) Similarity Graph (J')

Figure 3.6: Upper Bound Examples

Definition 8. ***(k, k′)-core.*** *Given a set of vertices U, the graph J and the corresponding similarity graph J', let J_U and J'_U denote the induced subgraph by U on J and J', respectively. If $deg_{min}(J_U) \geq k$ and $deg_{min}(J'_U) = k'$, U is a (k,k')-core of J and J'.*

Theorem 8. *Given the graph J, the corresponding similarity graph J', and the maximum (k,r)-core R derived from J and J', if there is a (k,k')-core on J and J' with the largest k', i.e., k'_{max}, we have $|R| \leq k'_{max} + 1$.*

Proof. Based on the fact that a (k,r)-core R is also a (k,k')-core with $k' = |R| - 1$ according to the definition of (k,r)-core, the theorem is proven immediately. □

Example 6. *In Figure 3.6, we have $k = 3$, $M = \{u_3\}$ and $C = \{u_1, u_2, u_4, u_5, u_6\}$. Figure 3.6(a) shows the induced subgraph J from $M \cup C$ on G and Figure 3.6(b) shows the similarity graph J' from $M \cup C$ on the similarity graph G'. We need at least 5 colors to color J', so the color based upper bound is 5. By core decomposition on similarity graph J', we get that the k-core based upper bound is 5 since $k_{max} = 4$ with 4-core $\{u_2, u_3, u_4, u_5, u_6\}$. Note that the vertices of this 4-core do not form a 3-core on J. Regarding the (k, k')-core based upper bound, we can find $k'_{max} = 3$ because there is a $(3, 3)$-core on J and J' with four vertices $\{u_2, u_3, u_5, u_6\}$, and there is no other (k, k')-core with a larger k' than k'_{max}. Consequently, the (k, k')-core based upper bound is 4, which is tighter than 5.*

Algorithm 6: **KK'coreBound**(M, C)

Input : M : vertices chosen, C : candidate vertices
Output: k'_{max} : the upper bound for the size of the maximum (k,r)-core in $M \cup C$

```
1  H := vertices in M ∪ C with increasing order of their similarity degrees;
2  for each u ∈ H do
3      k' := deg_sim(u);
4      KK'coreUpdate(u, k', H);
5      reorder H accordingly;
6  return k' + 1
7  KK'coreUpdate(u, k', H)
8  Remove u from H;
9  for each v ∈ NB_sim[u] ∩ H do
10     if deg_sim[v] > k' then
11         deg_sim[v] := deg_sim[v] − 1;
12 for each v ∈ NB[u] ∩ H do
13     deg[v] := deg[v] − 1;
14     if deg[v] < k then
15         KK'coreUpdate(v, k', H);
```

3.6.3 Algorithm for (k,k')-Core Upper Bound

Algorithm 6 shows the details of the (k,k')-core based upper bound (i.e.,k'_{max}) computation, which conducts core decomposition [12] on J' with additional update which ensures the corresponding subgraph on J is a k-core. We use $deg[u]$ and $deg_{sim}[u]$ to denote the degree and similarity degree (i.e., the number of similar pairs from u) of u w.r.t $M \cup C$, respectively. Meanwhile, $NB[u]$ (resp. $NB_{sim}[u]$) denotes the set of adjacent (resp. similar) vertices of u. The key idea is to recursively mark the k' value of the vertices until we reach the maximal possible value. Line 1 sorts all vertices based on the increasing order of their similarity degrees. In each iteration, the vertex u with the lowest similarity degree has already reached its maximal possible k' (Line 3). Then Line 4 invokes

the procedure KK'coreUpdate to remove u and decrease the degree (resp. similarity degree) of its neighbors (resp. similarity neighbors) at Lines 9-11 (resp. Lines 12-15). Note that we need to recursively remove vertices with degree smaller than k (Line 15) in the procedure. At Line 5, we need to reorder the vertices in H since their similarity degree values may be updated. According to Theorem 8, $k' + 1$ is returned at Line 6 as the upper bound of the maximum (k,r)-core size.

Time Complexity. We can use an array H to maintain the vertices where $H[i]$ keeps the vertices with similarity degree i. Then the sorting of the vertices can be done in $O(|J|)$ time. Similarly, the re-location cost of each individual vertex is bounded by the number of its adjacent edges in J. The time complexity of the algorithm is $O(n_e + n_s)$, where n_e and n_s denote the number of edges in the graph J and the similarity graph J', respectively. In Algorithm 6, each edge in J or J' will be visited at most once for the update of degrees and similarity degrees.

Algorithm Correctness. Let $k'_{max}(u)$ denote the largest k' value u can contribute to (k,k')-core of J. By H_j, we represent the vertices $\{u\}$ with $k'_{max}(u) \geq j$ according to the definition of (k,k')-core. We then have $H_j \subseteq H_i$ for any $i < j$. This implies that a vertex u on H_i with $k'_{max}(u) = i$ will not contribute to H_j with $i < j$. Thus, we can prove correctness by induction.

3.6.4 Finding the Top-m Maximal (k,r)-Cores

Besides the maximum (k,r)-core, social network service providers would also like to see the top-m maximal (k,r)-cores whose activeness reflects the hotness of the network. Users may be interested in these top-m communities which are most representative and appealing groups in the network. To find the top-m maximal (k,r)-cores, we can record current top-m largest maximal (k,r)-cores

in Algorithm 5. In Line 2, the size upper bound is compared with the size of the minimum maximal (k,r)-core recorded. And in Line 4, we replace the minimum maximal (k,r)-core with current larger one if it is maximal. More specifically, the output of Algorithm 5 should be "$\{R_j \mid j \in N^+ \ \& \ j \leq m\}$: the top-$m$ largest maximal (k,r)-cores seen so far". Line 2 should be replaced by "If $\{KRCoreSizeUB(M, C) > |min(R_j)|\}$ then" where $min(R_j)$ is the minimum maximal (k,r)-core in $\{R_j \mid j \in N^+ \ \& \ j \leq m\}$. Line 4 should be replaced by "$min(R_j) := M \cup C$ If CheckMaximal(M, E);". Algorithm 6 keeps same since there is no difference towards finding the maximum and the top-m.

3.7 Search Order

Section 3.7.1 briefly introduces some important measurements that should be considered for an appropriate visiting order. Then we investigate the visiting orders in three algorithms: finding the maximum (k,r)-core (Algorithm 5), advanced maximal (k,r)-core enumeration (Algorithm 3) and maximal check (Algorithm 4) at Section 3.7.2, Section 3.7.3, and Section 3.7.4, respectively.

3.7.1 Important Measurements

In this chapter, we need to consider two kinds of search orders: (i) the vertex visiting order: the order of which vertex is chosen from candidate set C and (ii) the branch visiting order: the order of which branch goes first (expand first or shrink first). It is difficult to find simple heuristics or cost functions for two problems studied in this chapter because, generally speaking, finding a maximal/maximum (k,r)-core can be regarded as an optimization problem with two constraints. On one hand, we need to reduce the number of dissimilar pairs to satisfy the similarity constraint, which implies eliminating a considerable

number of vertices from C. On the other hand, the structure constraint and the maximal/maximum property favors a larger number of edges (vertices) in $M \cup C$; that is, we prefer to eliminate fewer vertices from C.

To accommodate this, we propose three measurements where M' and $\mathbf{C}'$ denote the updated M and C after a chosen vertex is extended to M or discarded.

- Δ_1 : the change of number of dissimilar pairs, where

$$\Delta_1 = \frac{DP(C) - DP(\mathbf{C}')}{DP(C)} \tag{3.3}$$

  Note that we have $DP(u, M \cup C) = 0$ for every $u \in M$ according to the similarity invariant (Equation 3.1).

- Δ_2 : the change of the number of edges, where

$$\Delta_2 = \frac{|\mathcal{E}(M \cup C)| - |\mathcal{E}(M' \cup \mathbf{C}')|}{|\mathcal{E}(M \cup C)|} \tag{3.4}$$

  Recall that $|\mathcal{E}(V)|$ denote the number of edges in the induced graph from the vertices set V.

- $deg(u, M \cup C)$: Degree. We also consider the degree of the vertex as it may reflect its importance. In our implementation, we choose the vertex with highest degree at the initial stage (i.e., $M = \emptyset$).

3.7.2 Finding the Maximum (k,r)-Core

Since the size of the largest (k,r)-core seen so far is critical to reduce the search space, we aim to quickly identify the (k,r)-core with larger size. One may choose to carefully discard vertices such that the number of edges in M is reduced slowly (i.e., only prefer smaller Δ_2 value). However, as shown in our empirical study,

this may result in poor performance because it usually takes many search steps to satisfy the structure constraint. Conversely, we may easily fall into the trap of finding (k,r)-cores with small size if we only insist on removing dissimilar pairs (i.e., only favor larger Δ_1 value).

In our implementation, we use a cautious greedy strategy where a parameter λ is used to make the trade-off. In particular, we use $\lambda\Delta_1 - \Delta_2$ to measure the suitability of a branch for each vertex in $C \setminus SF(C)$. In this way, each candidate has two scores. The vertex with the highest score is then chosen and its branch with higher score is explored first (Line 6-12 in Algorithm 5).

For time efficiency, we only explore vertices within two hops from the candidate vertex when we compute its Δ_1 and Δ_2 values. It takes $O(n_c \times (d_1^2 + d_2^2))$ time where n_c denote the number of vertices in $C \setminus SF(C)$, and d_1 (resp. d_2) stands for the average degree of the vertices in J (resp. J').

3.7.3 Enumerating All Maximal (k,r)-Core

The ordering strategy in this section differs from the maximum in two ways.

(i) We observe that Δ_1 has much higher impact than Δ_2 in the enumeration problem, so we adopt the Δ_1-then-Δ_2 strategy; that is, we prefer the larger Δ_1, and the smaller Δ_2 is considered if there is a tie. This is because the enumeration algorithm does not prefer (k,r)-core with very large size since it eventually needs to enumerate all maximal (k,r)-cores. Moreover, by the early termination technique proposed in Section 3.5.2, we can avoid exploring many non-promising subtrees that were misled by the greedy heuristic.

(ii) We do not need to consider the search order of two branches because both must be explored eventually. Thus, we use the score summation of the two branches to evaluate the suitability of a vertex. The complexity of this ordering strategy is the same as that in Section 3.7.2.

3.7.4 Checking Maximal

The search order for checking maximals is rather different than the enumeration and maximum algorithms. Towards the checking maximals algorithm, it is cost-effective to find a *small* (k,r)-core which fully contains the candidate (k,r)-core. To this end, we adopt a short-sighted greedy heuristic. In particular, we choose the vertex with the largest degree and the expand branch is always preferred as shown in Algorithm 4. By continuously maintaining a priority queue, we fetch the vertex with the highest degree in $O(\log |C|)$ time.

3.8 Performance Evaluation

This section evaluates the effectiveness and efficiency of our algorithms through comprehensive experiments.

3.8.1 Experimental Setting

Datasets. Four real datasets are used in our experiments. The original data of `DBLP` was downloaded from `http://dblp.uni-trier.de` and the remaining three datasets were downloaded from `http://snap.stanford.edu`. In `DBLP`, we consider each author as a vertex with attribute of counted "attended conferences" and "published journals" list. There is an edge for a pair of authors if they have at least one co-authored paper. We use *Weighted Jaccard Similarity* between the corresponding attributes (counted conferences and journals) to measure the similarity between two authors. Thus, we can detect some groups in which people have cohesive co-authorship and similar research background. In `Pokec`, we consider each user to be a vertex with personal interests. We use *Weighted Jaccard Similarity* as the similarity metric. And there is an edge between two users if they are friends. Then, we can find groups of people who have at least

Table 3.2: Statistics of Datasets

Dataset	**Nodes**	**Edges**	d_{avg}	d_{max}
`Brightkite`	58,228	194,090	6.67	1098
`Gowalla`	196,591	456,830	4.65	9967
`DBLP`	1,566,919	6,461,300	8.25	2023
`Pokec`	1,632,803	8,320,605	10.19	7266

k friends in the same group and share similar personal interests. In `Gowalla` and `Brightkite`, we consider each user as a vertex along with his/her location information. The graph is constructed based on friendship information. We use *Euclidean Distance* between two locations to measure the similarity between two users. Thus, we may detect subgroups in which people have cohesive friendship and close mutual distance. Table 3.2 shows the statistics of the four datasets.

Algorithms. To the best of our knowledge, there are no existing works that investigate the problem of (k,r)-core. In this chapter, we implement and evaluate 2 baseline algorithms, 2 advanced algorithms and the clique-based algorithm which are described in Table 3.3. Since the naive method in Section 3.4 is extremely slow even on a small graph, we employ `BasEnum` and `BasMax` as the baseline algorithms in the empirical study for the problem of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core, respectively. In Table 3.3, we also show the name for each technique, which may be equipped or unloaded for the mentioned algorithms. Note that we do not specifically evaluate the structure/similarity based candidate pruning techniques because they are indispensable for the baseline algorithm.

Remark 4. *Since the naive method in Section 3.4 is extremely slow even on a small graph, we employ* `BasEnum` *and* `BasMax` *as the baseline algorithms in the empirical study for the problem of enumerating all maximal (k,r)-cores and finding the maximum (k,r)-core, respectively.*

Table 3.3: Summary of Algorithms

Technique	**Description**
CR	The candidate retaining technique (Theorem 5).
ET	The early termination technique (Theorem 6).
CM	The checking maximal technique (Theorem 7).
CK	A tighter upper bound from the color based and the k-core based upper bound. (Section 3.6.2).
UB	The (k,k')-core upper bound technique (Theorem 8).
SO	The best search order is applied (Section 3.7).
Algorithm	**Description**
Clique+	The advanced clique-based algorithm proposed in Section 3.3, using the clique and k-core computation algorithms in [84] and [12], respectively. The source code for maximal clique enumeration was downloaded from http://www.cse.cuhk.edu.hk/~jcheng/publications.html.
BasEnum	The basic enumeration method proposed in Algorithm 1 including the structure and similarity constraints based pruning techniques (Theorems 3 and 4 in Section 3.5.1). The best search order (Δ_1-then-Δ_2, in Section 3.7.3) is applied.
AdvEnum	AdvEnum = BasEnum+CR+ET+CM. The advanced enumeration algorithm proposed in Section 3.5.4 that applies all advanced pruning techniques including: candidate size reduction (Theorems 3, 4 and 5 in Section 3.5.1), early termination (Theorem 6 in Section 3.5.2) and checking maximals (Theorem 7 in Section 3.5.3). Moreover, the best search order is used (Δ_1-then-Δ_2, in Section 3.7.3).
BasMax	The algorithm proposed in Section 3.6.1 with the upper bound replaced by a naive one: $\|M\|+\|C\|$. The best search order is applied ($\lambda\Delta_1 - \Delta_2$, in Section 3.7.2).
AdvMax	AdvMax = BasMax+UB. The advanced finding maximum (k,r)-core algorithm proposed in Section 3.6.1 including (k,k')-core based upper bound technique (Algorithm 6). Again, the best search order is applied ($\lambda\Delta_1 - \Delta_2$, in Section 3.7.2).

Parameters. We conducted experiments using different settings of k and r. We set reasonable positive integers for k, which varied from 3 to 15. In Gowalla and

`Brightkite`, we used Euclidean distance as the distance threshold r, ranging from 1 km to 500 km. The pairwise similarity distributions are highly skewed in `DBLP` and `Pokec`. Thus, we used the thousandth of the pairwise similarity distribution in decreasing order which grows from top 1‰ to top 15‰ (i.e., the similarity threshold value drops). Regarding the search orders of the `AdvMax` and `BasMax` algorithms, we set λ to 5 by default.

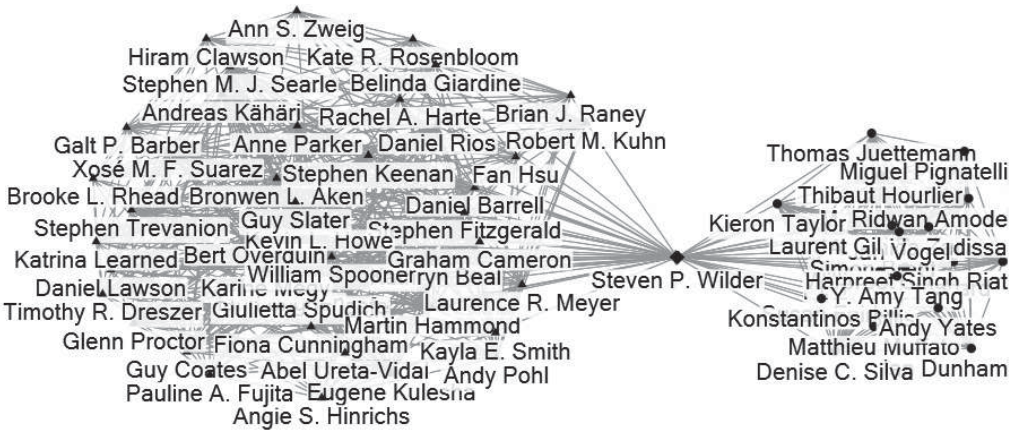
All programs were implemented in standard C++ and compiled with G++ in Linux. All experiments were performed with Intel Xeon 2.3GHz CPUs and a Redhat Linux system. The time cost is set to INF if an algorithm did not terminate within one hour. The source code is available at `https://sites.google.com/view/fanzhang`. We evaluate the performance of an algorithm by its running time. To better evaluate the algorithm differences, we set the time cost to INF if an algorithm did not terminate within one hour. We also report the number of maximal (k,r)-cores, and their average/maximum sizes.

3.8.2 Effectiveness

We conducted case studies on `DBLP` and `Gowalla` to demonstrate the effectiveness of our (k,r)-core model. Compared to k-core, (k,r)-core enables us to find more valuable information with the additional similarity constraint on the vertex attribute.

DBLP. Figure 3.7 shows a case of `DBLP` with $k = 15$ and r= 3‰[4]. In Figure 3.7(a), all authors come from the same k-core based on their co-authorship information alone (their structure constraint). While there are two (k,r)-cores with one common author named Steven P. Wilder, if we also consider their research background (their similarity constraint). We find the result of (k,r)-cores

[4]To avoid the noise, we enforce that there are at least three co-authored papers between two connected authors in the case study.



(a) Enumeration

(b) Maximum

Figure 3.7: Case Study on DBLP (k=15, r=top 3‰)

is consistent with reality that there are two groups of people with Dr. Wilder from both sides. A large number of the authors in the left (k,r)-core are bioinformaticians from the European Bioinformatics Institute (EBI), while many of the authors in the right (k,r)-core are from the Wellcome Trust Centre (WTC). Dr. Wilder got his Ph.D. from the WTC, University of Oxford in 2007, and has worked at EBI ever since. Currently, he is a bioinformatician with research fo-

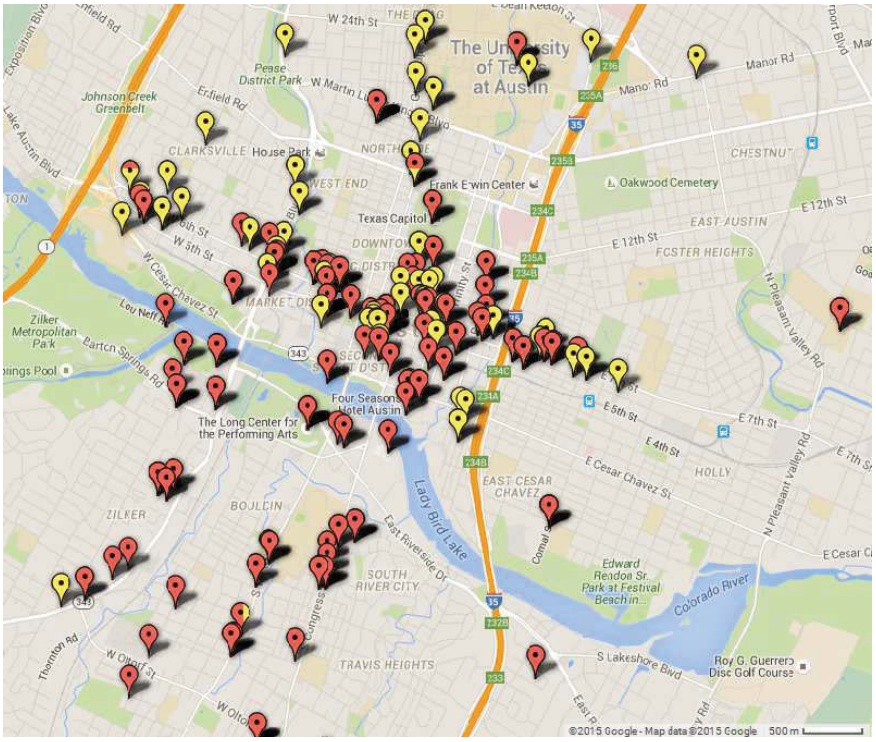
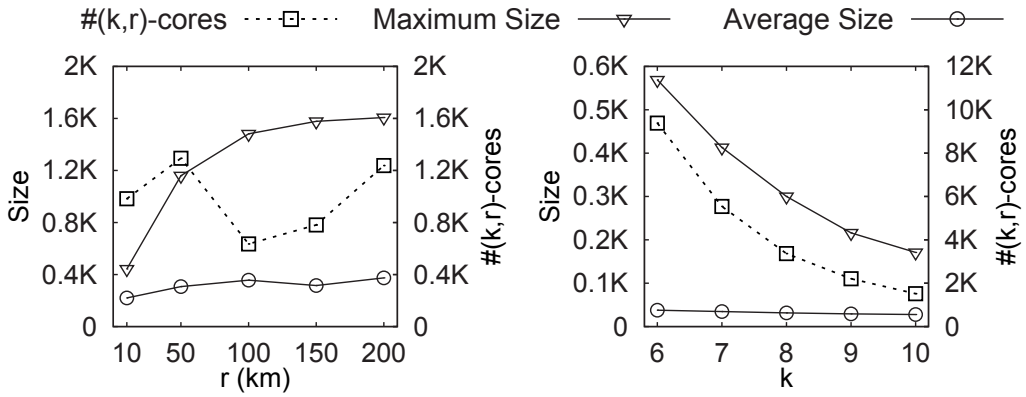


Figure 3.8: Case Study on Gowalla (k=10, r=10km)

cus on genome analysis. Figure 3.7(b) depicts the maximum (k,r)-core of DBLP with 49 authors. We find that they have intensively co-authored many papers related to a project named *Ensembl* (`http://www.ensembl.org/index.html`), which is one of the well known genome browsers. It is very interesting that, although the size of maximum (k,r)-core changes when we vary the values of k and r, the authors remaining in the maximum (k,r)-core are closely related to the project.

Gowalla. Figure 3.8 illustrates a set of `Gowalla` users who are from the same k-core with $k = 10$. By setting r to 10 km, two groups of users emerge, each of which is a maximal (k,r)-core, and we cannot identify them by structure constraint or similarity constraint alone. We observe that the maximum (k,r)-core in `Gowalla` always appears at Austin when $k \geq 6$. Then we realize that this



(a) Gowalla, k=5 (b) DBLP, r=top 3‰

Figure 3.9: (k,r)-core Statistics

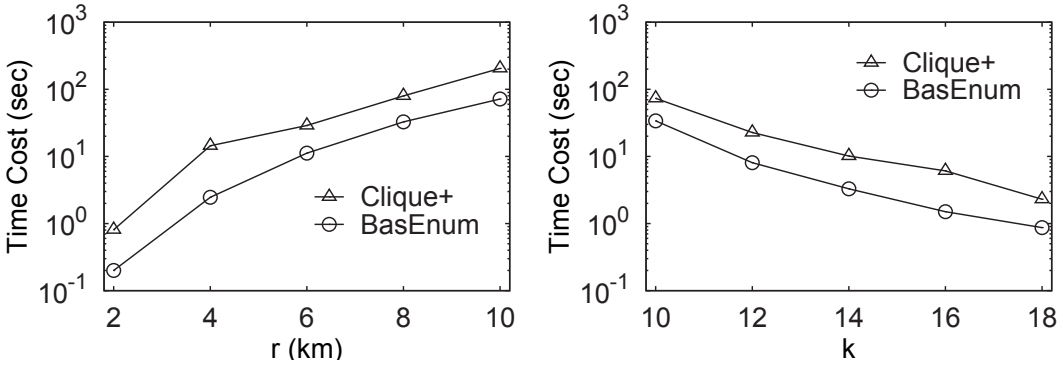
is because the headquarters of `Gowalla` is located in Austin.

We also report the number of (k,r)-cores, the average size and maximum size of (k,r)-cores on `Gowalla` and `DBLP`. Figure 3.9(a) and (b) show that both maximum size of (k,r)-cores and the number of (k,r)-cores are much more sensitive to the change of r or k on the two datasets, compared to the average size.

3.8.3 Efficiency

In this section, we evaluate the efficiency of the techniques proposed in this chapter and report the time costs of the algorithms.

Evaluating the Clique-based Method. In Figure 3.10, we evaluate the time cost of the maximal (k,r)-core enumeration for `Clique+` and `BasEnum` on the `Gowalla` and `DBLP` datasets. In Figure 3.11(a), we fix the structure constraint k to 5 and vary the similarity threshold r from 2 km to 10 km. Correspondingly in Figure 3.11(b), we fix r to 3‰ and vary k from 18 to 10. In the experiments, `BasEnum` always outperform `Clique+` by a stable margin because we apply pruning rules in `BasEnum` with the best search order and a large number of of cliques are materialized in the similarity graphs for `Clique+`. Consequently, we exclude



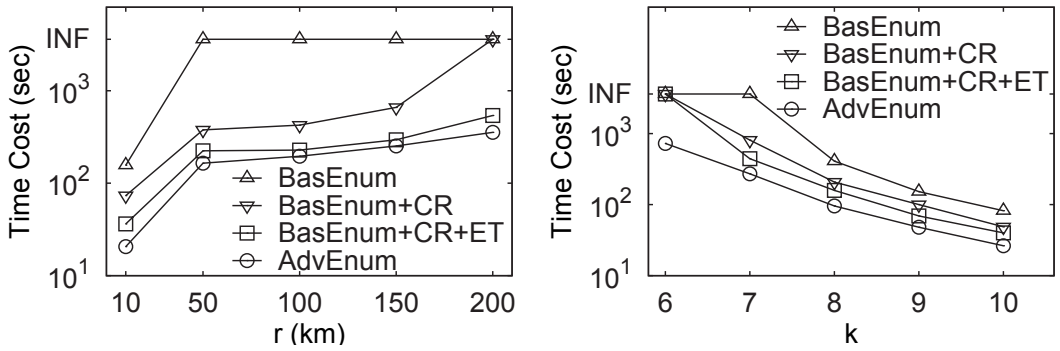
(a) Gowalla, k=5 (b) DBLP, r=top 3‰

Figure 3.10: Evaluate Clique-based Method

`Clique+` from the following experiments. This supports the insight that for a problem of computing cohesive subgraphs on dual graphs, a careful integration of existing cohesive subgraph computations at each search step (e.g., `BasEnum`) is better than computing the two kinds of cohesive subgraphs sequentially (e.g., `Clique+`).

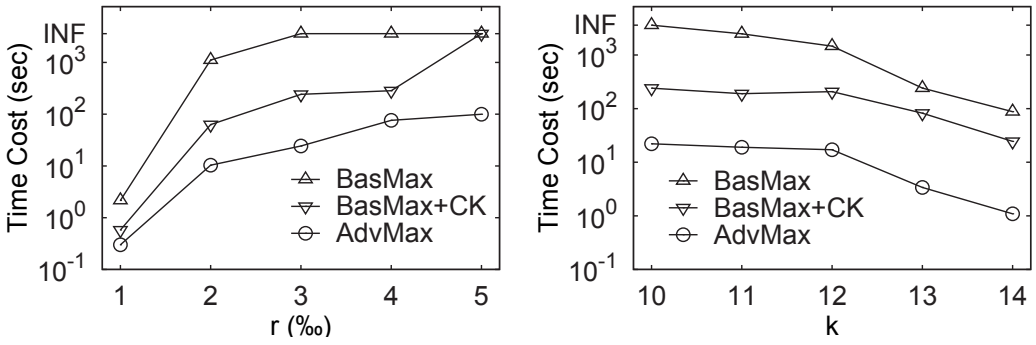
Evaluating the Pruning Techniques. In Figure 3.11, we evaluate the efficiency of our pruning techniques on `Gowalla` and `DBLP` by incrementally integrating these techniques from `BasEnum`, `BasEnum+CR`, `BasEnum+CR+ET` to `AdvEnum` (`BasEnum+CR+ET+CM`). Note that the best search order is used for all algorithms. Among these techniques, Theorem 5 achieves the best speedup because the search on $SF(C)$ is skipped and $SF(C)$ may be large. The results in Figure 3.11 confirm that all techniques contribute to enhance the performance of `AdvEnum`.

Evaluating the Upper Bound Technique. Figure 3.12 demonstrates the effectiveness of the (k,k')-core based upper bound technique (Algorithm 6) on DBLP by varying the values of r and k. In `BasMax+CK`, we used the better upper bound from color and k-core based upper bound techniques (Section 3.6.2) [39, 12]. Studies show that `BasMax+CK` greatly enhances performance compared to



(a) Gowalla, k=5 (b) DBLP, r=top 3‰

Figure 3.11: Evaluate Pruning Techniques

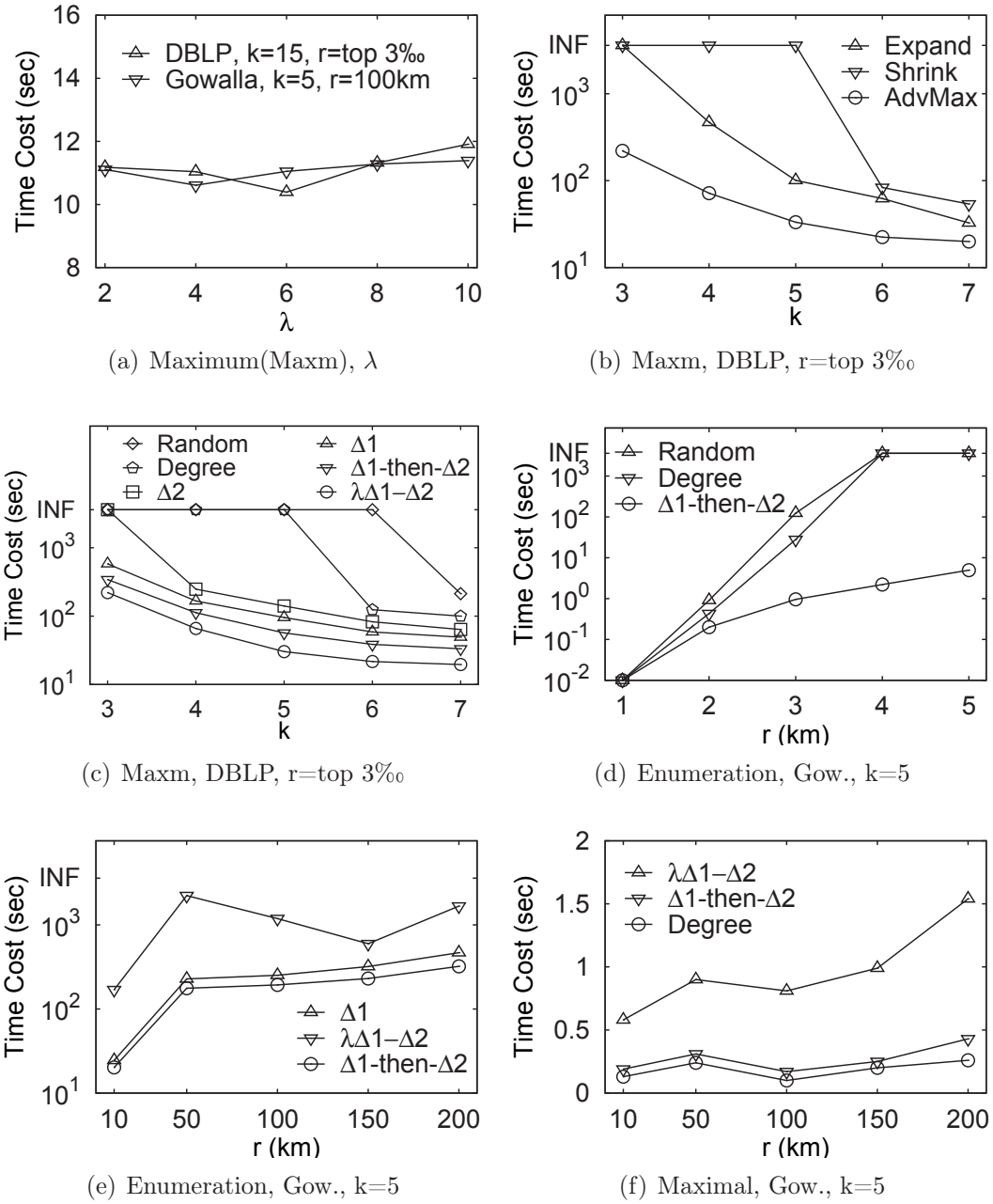


(a) DBLP, k=10 (b) DBLP, r=top 3‰

Figure 3.12: Evaluate Upper Bounds

the naive upper bound $|M| + |C|$ (used in `BasMax`). Nevertheless, our (k,k')-core based upper bound technique (`AdvMax`) outperforms `BasMax+CK` by a large margin because it can better exploit the structure/similarity constraints.

Evaluating the Search Orders. In this experiment, we evaluate the effectiveness of the three search orders proposed for the maximum algorithm (Section 3.7.2, Figure 3.13(a)-(c)), enumeration algorithm (Section 3.7.3, Figure 3.13(d)-(e)) and the checking maximal algorithm (Section 3.7.4, Figure 3.13(f)). We first tune λ value for the search order of `AdvMax` in Figure 3.13(a) against



(a) Maximum(Maxm), λ

(b) Maxm, DBLP, r=top 3‰

(c) Maxm, DBLP, r=top 3‰

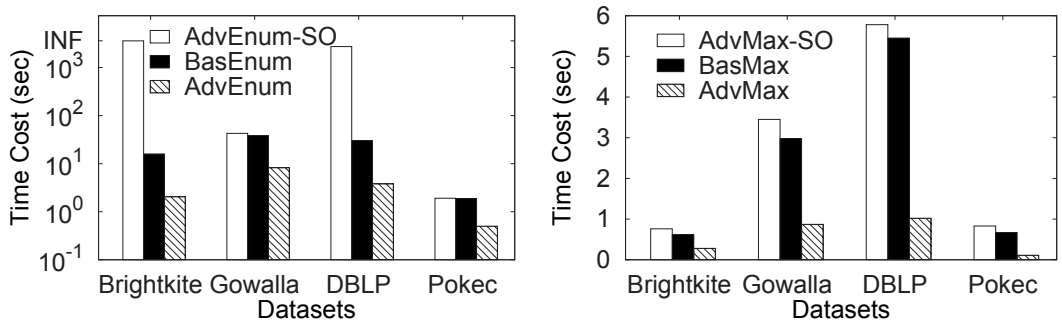
(d) Enumeration, Gow., k=5

(e) Enumeration, Gow., k=5

(f) Maximal, Gow., k=5

Figure 3.13: Evaluate Search Orders

DBLP and Gowalla. In the following experiments, we set λ to 5 for maximum algorithms. Figure 3.13(b) verifies the importance of the adaptive order for the two branches on DBLP where Expand (resp. Shrink) means the expand (resp.

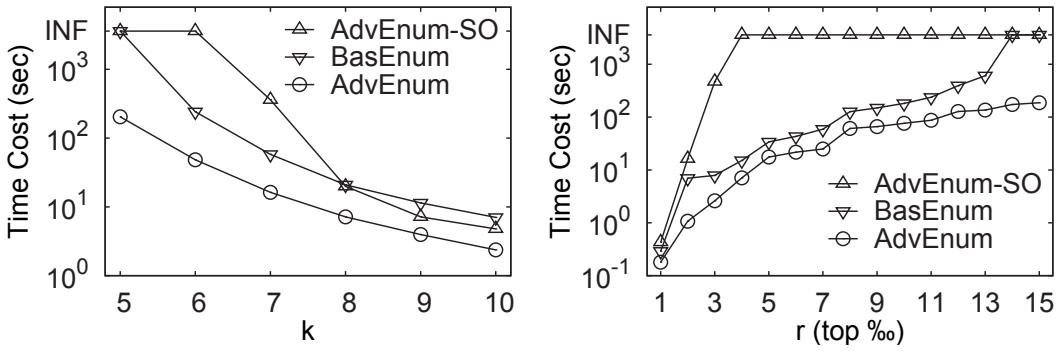


(a) Enumeration (b) Maximum

Figure 3.14: Performance on Four Datasets

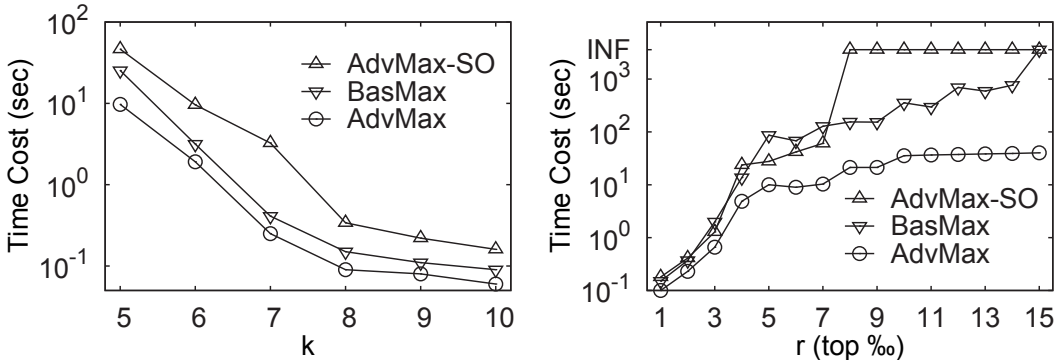
shrink) branch is always preferred in `AdvMax`. In Figure 3.13(c), we investigate a set of possible order strategies for `AdvMax`. As expected, the $\lambda\Delta_1 - \Delta_2$ order proposed in Section 3.7.2 outperforms the other alternatives including random order, degree based order (Section 3.7.4, used for checking maximals), Δ_1 order, Δ_2 order and Δ_1-then-Δ_2 order (Section 3.7.3, used by `AdvEnum`). Similarly, Figure 3.13(d) and (e) confirm that the Δ_1-then-Δ_2 order is the best choice for `AdvEnum` compared to the alternatives. Figure 3.13(f) shows that the degree order achieves the best performance for the checking maximal algorithm (Algorithm 4) compared to the two orders used by `AdvEnum` and `AdvMax`.

Effect of Different Datasets. Figure 3.14 evaluates the performance of the enumeration and maximum algorithms on four datasets with $k = 10$. We set r to 500 km, 300 km, 3‰ and 5‰ in `Brightkite`, `Gowalla`, `DBLP` and `Pokec`, respectively. We use `AdvEnum-SO` to denote the `AdvEnum` algorithm *without* the best search order while all other advanced techniques applied (degree order is used instead). Figure 3.14(a) demonstrates the efficiency of those techniques and search orders on four datasets. We also demonstrate the efficiency of the upper bound and search order for the maximum algorithm in Figure 3.14(b), where three algorithms are evaluated (`AdvMax-SO`, `BasMax`, and `AdvMax`).



(a) Gowalla, r=100 (b) DBLP, k=15

Figure 3.15: Effect of k and r for Enumeration



(a) Gowalla, r=100 (b) DBLP, k=15

Figure 3.16: Effect of k and r for Maximum

Effect of k and r. Figure 3.15 studies the impact of k and r for the three enumeration algorithms on `Gowalla` and `DBLP`. As expected, Figure 3.15(a) shows that the time cost drops when k grows because many more vertices are pruned by the structure constraint. In Figure 3.15(b), the time costs grow when r increases because more vertices will be included in (k,r)-cores when the similarity threshold drops. Similar trends are also observed in Figure 3.16 for three maximum algorithms. Moreover, Figure 3.15 and 3.16 further confirm the effectiveness of proposed techniques. `AdvMax` greatly outperforms `AdvEnum` under the same setting because `AdvMax` can further cut-off the search tree based on the derived upper bound of the (k,r)-core size and does not need maximal check.

3.9 Conclusion

In this chapter, we propose a novel cohesive subgraph model, called (k,r)-core, which considers the cohesiveness of a subgraph from the perspective of both graph structure and vertex attribute. We show that the problem of enumerating the maximal (k,r)-cores and finding the maximum (k,r)-core are both NP-hard. Several novel pruning techniques are proposed to improve algorithm efficiency, including candidate size reduction, early termination, checking maximals and upper bound estimation techniques. We also devise effective search orders for enumeration, maximum and maximal check algorithms. Extensive experiments on real-life networks demonstrate the effectiveness of the (k,r)-core model, as well as the efficiency of our techniques.

Chapter 4

Prevent Network Unraveling

4.1 Introduction

Prevent network unraveling is a crucial task for social networking sites to avoid the churn of users, as in the collapse of Friendster [65, 78]. Bhawalkar and Kleinberg *e*t al. [14] show that each user in a social network spends a cost to remain engaged but obtains a benefit which is proportional to the number of engaged friends. The equilibrium leads to the well-known model k-core [76], in which each vertex has at least k neighbors. In this basic model, a user will remain engaged if, and only if, at least k of his/her friends are engaged. A user with less than k friends engaged will leave. His/her departure may be contagious and form a cascade of departures in the network. This procedure is called *network unraveling* in which active individuals may leave by the negative influence of his/her friends.

To prevent unraveling in social networks, Bhawalkar and Kleinberg *e*t al. [14] introduce the problem of anchored k-core. The aim is to retain (anchor) some users with incentives to ensure they will not leave regardless of the behavior of others, so that the largest number of users will remain engaged when the

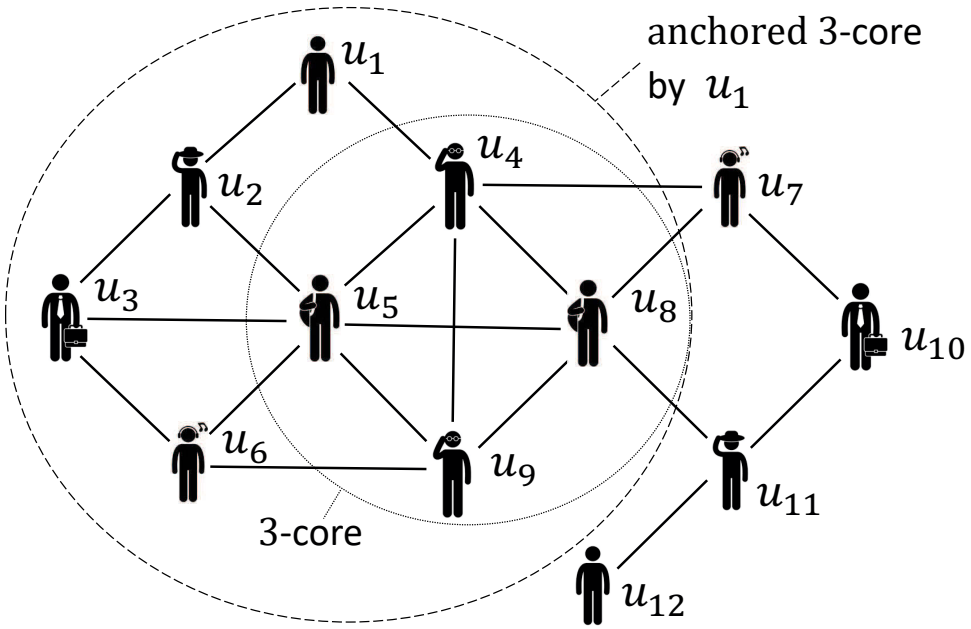


Figure 4.1: Motivating Example for Anchored k-Core

unraveling stops. Formally speaking, it is to anchor a set of b vertices such that the induced k-core is the largest one. This problem has a wide range of applications, and can identify users whose participation is critical to overall engagement of the networks. Below is a motivating example.

Example 7. *Suppose there is a computer science study group, and the number of friends in the group represents the willingness of a member to engage in the collaborative learning. If one leaves, he/she will weaken the willingness of his/her friends to remain engaged, which may incur the unraveling of the study group. As illustrated in Figure 4.1, we model 12 members in a study group and their relationships as a network. According to the above engagement model with k=3, i.e., a person will leave if there are less than three friends, four members will remain engaged eventually; that is, 3-core of the network includes u_4, u_5, u_8 and u_9. To prevent unraveling, we may persuade the member u_1 not to leave through additional incentives, such as a regular personal tutoring or priority booking of*

the study room . As a result, members u_2, u_3 and u_6 will also remain engaged since each of them now has three friends in the study group. This motivates us to find the most cost effective way to "anchor" a set of b members so that the size of the resulting k-core is maximized. It turns out that the optimal solution is $\{u_1\}$ and $\{u_1, u_{10}\}$ for $b = 1$ and $b = 2$, respectively.

Besides ensuring high user engagement, tie strength should be considered in some scenarios of social network unraveling [42]. Most existing methods for strong tie detection are based on structural information, especially on triangles, which has been shown to be effective in social networks [71, 81]. The model of k-core fits well with the communities which do not require tie strength, such as the study group mentioned in Example 7, where students may discuss with other students in the group, even though their relationships are not quite strong. Nevertheless, some community types need to consider the strength of ties, e.g., a programming team in a company, where every member would like enough familiar colleagues in this team to conduct better collaboration and avoid unexpected troubles. These communities are called strong tie communities which correspond to the model of k-truss [29] where each edge is contained in at least $k-2$ triangles in the subgraph. Besides ensuring the strength of ties, the k-truss is also a subgraph of (k-1)-core and a subgraph of (k-1)-edge-connected subgraph where the latter remains connected whenever fewer than k-1 edges are removed in the subgraph. It motivates us to study the problem of anchored k-truss to prevent collapse of strong tie communities, which is to retain (anchor) some users in the network such that the k-truss community has the largest number of users.

Example 8. *Suppose there is a programming group, and the number of common friends in the group for two friends represents the strength of their relationship. The weak friendships cannot contribute to this group, i.e., the edges involving in fewer than $k-2$ triangles are removed in the group. The deletion of edges will*

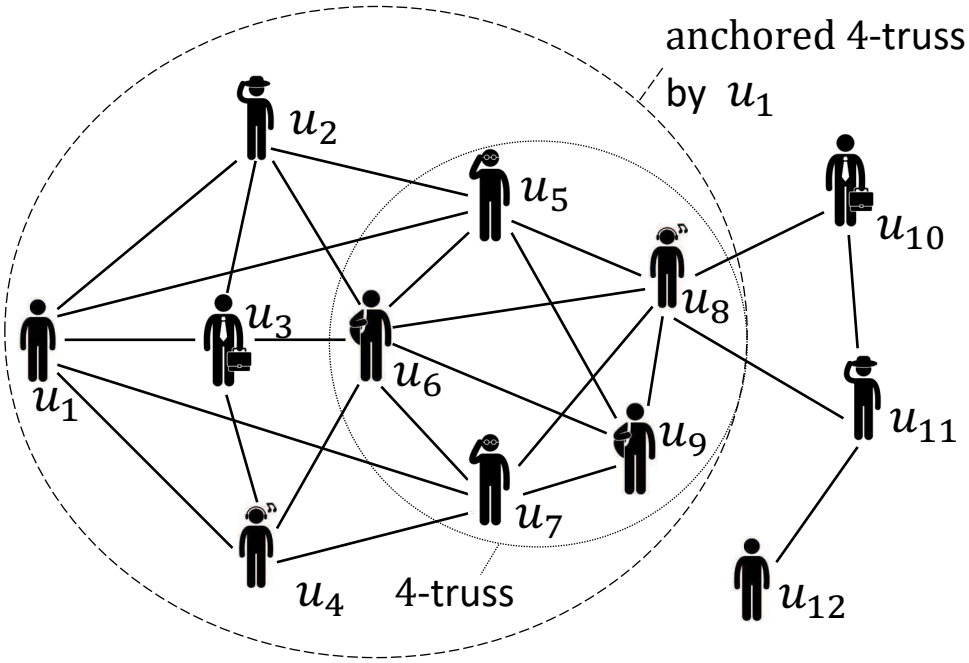


Figure 4.2: Motivating Example for Anchored k-Truss

reduce the number of triangles for some other edges and incur the unraveling of the group. As illustrated in Figure 4.2, we model 12 members in a programming group and their relationship as a network. According to the k-truss model with k=4, i.e., an edge will be removed if it involves in less than two triangles, five members will remain engaged eventually; that is, 4-truss of the network includes u_5, u_6, u_7, u_8 *and* u_9. *Note that the edges* (u_1, u_5) *and* (u_1, u_7) *start the unraveling of the left part users. To prevent unraveling, we may persuade the member* u_1 *not to leave through additional incentives, such as a salary bonus. Thus the edges of* u_1 *can exist in the group as long as involving in a triangle. As a result, members* u_2, u_3 *and* u_4 *will also remain engaged since the corresponding edges now involve in enough triangles in the group. This motivates us to "anchor" a set of b members so that the size of the resulting k-truss is maximized. The optimal solution for* $b = 1$ *is* $\{u_1\}$.

The anchored k-core problem [14] is NP-hard and inapproximable when $k \geq 3$. A polynomial-time algorithm in graphs with bounded tree-width was proposed in [14], but it cannot handle general large-scale graphs. Consider the hardness of the problem, we choose a greedy strategy, where a best anchor is found in each iteration by computing the k-core for each possible anchor vertex. As demonstrated in our empirical study, a straightforward implementation of the greedy algorithm is very time consuming due to the large number of candidate anchors. To the best of our knowledge, we are the first to study the anchored k-truss problem to prevent unraveling of strong tie communities. We prove the problem is NP-hard when $k > 3$. We also adopt a greedy strategy to find a best anchor in each iteration. However, the cost of the greedy algorithm for anchored k-truss is much more than anchored k-core for two reasons: (1) the larger number of candidate anchors. (2) the polynomial algorithm for computing k-truss is more costly than the linear algorithm for computing k-core.

To address the above issues, we design two auxiliary structures $\mathcal{L}$, *onion layers* for k-core and *edge onion layers* for k-truss. The former divides a small set of *vertices* by layers and the latter divides a small set of *edges* by layers. Based on the two structures, we develop corresponding efficient techniques to significantly reduce the search space.

Due to the existence of anchor vertices, some new vertices will join k-core (resp. k-truss), which are termed **followers** in this chapter. The number of followers is the *gain* of the anchoring activity. As we adopt the greedy heuristics, our research focuses on finding the best anchor in the graph, i.e., the vertex with the largest number of followers. For the anchored k-core problem, we observe that when we only consider one anchor, all followers must reside on the (k-1)-shell, i.e., the vertices in (k-1)-core but not in k-core. We put these vertices, their neighbors and corresponding edges into *onion layers* $\mathcal{L}$ and order the vertices

by layers. For the anchored k-truss problem, we observe that when we only consider one anchor, all followers must reside on the (k-1)-hull, i.e., the vertices in (k-1)-truss but not in k-truss. We put the vertices and edges, which involve in a triangle where there is an edge from (k-1)-hull, into *edge onion layers* $\mathcal{L}$ and order the edges by layers.

For both problems, we only need to consider the vertices in the corresponding $\mathcal{L}$ as the candidate anchors to find the best anchor vertex. By doing so, the number of candidate anchors is significantly reduced. More importantly, all of the follower computations for finding the best anchor are restricted to a small part of $\mathcal{L}$, which significantly reduces the search space. We enhance the computation for a given anchor by imposing the layer structure on $\mathcal{L}$ and develop efficient algorithms to quickly find its followers. The key idea is that, considering we will try a large number of candidate anchors, it is worthwhile to partition $\mathcal{L}$ into several layers in each iteration of the greedy algorithm so that the unraveling procedure can be conducted following a layer-by-layer paradigm. We formally prove that our layer-by-layer computation can always produce the correct results. By using the well-organized layer structure $\mathcal{L}$, we can effectively identify the candidate followers, and develop early termination and candidate anchor pruning techniques to eliminate non-promising followers and anchors at an early stage.

Note that although the ideas in *onion layers* and *edge onion layers* are similar, there are some major differences for the two problems: (i) a vertex in k-truss can be a candidate anchor (i.e., have some followers) in the anchored k-truss problem, while a vertex in k-core cannot in the anchored k-core problem; (ii) the *onion layers* is based on vertices while the *edge onion layers* is based on edges, although both the problems focus on anchoring vertices and computing vertex followers; and (iii) the vertex support conditions in k-core and k-truss computations are inherently different.

Road Map. Section 4.2 introduces k-core and the anchored k-core problem. Section 4.3 presents our solution for the anchored k-core problem. Section 4.4 studies the anchored k-truss problem and presents our solution. Section 4.5 evaluates the proposed algorithms. Section 4.6 concludes the chapter.

4.2 Preliminaries

In this section, we first give some necessary notations and introduce the concept of k-core and its corresponding algorithm. Then, we formally define the anchored k-core problem and show its hardness. Table 4.1 summarizes the mathematical notations used throughout the anchored k-core problem in this chapter.

4.2.1 Problem Definition

We consider an unweighted and undirected graph $G = (V, E)$, where V (resp. E) represents the set of vertices (resp. edges) in G. We denote $n = |V|$, $m = |E|$ and assume $m > n$. $NB(u, G)$ is the set of adjacent vertices of u in G, which is also called the neighbor set of u in G. We use $deg(u, G)$, the degree of u in G, to represent the number of adjacent vertices of u in G if $u \notin A$. $NB(u, G)$ (resp. $deg(u, G)$) is also written as $NB(u)$ (resp. $deg(u)$) when the context is clear. We also use G to represent the vertices in G. Given a subgraph $J \subseteq G$, $NB(J)$ denotes the neighbor set of the vertices in J, i.e., $NB(J) = \{u \mid NB(u, J) \neq \emptyset \ \& \ u \in G\}$.

The concept of k-core has been widely used to describe cohesive subgraphs, which is formally defined as follows.

Definition 9. ***k-core****. Given a graph G, a subgraph J is the k-core of G, denoted by $C_k(G)$, if (i) J satisfies degree constraint, i.e., $deg(u, J) \geq k$ for every $u \in J$; and (ii) J is maximal, i.e., any subgraph $J' \supset J$ is not a k-core.*

Table 4.1: Summary of Notations

Notation	Definition
G	an unweighted and undirected graph
u, v, x	a vertex in the graph
n, m	the number of vertices and edges in G
A	a set of anchor vertices
$NB(u, G)$	the set of adjacent vertices of u in G
$deg(u, G)$	$\|NB(u, G)\|$ if $u \notin A$; $+\infty$ if $u \in A$
G_A (G_x)	graph G anchored by A (x)
k	the degree constraint
b	the budget for the number of anchors
$C_k(G)$, $S_k(G)$	k-core and k-shell of G
$\mathcal{L}$ (i.e., L_0^s)	*onion layers* of G (with $s+1$ layers)
L_i	vertices on i-th layer of $\mathcal{L}$
L_i^j	$\bigcup_{i \le k \le j} L_k$
$l(u)$	layer index of the vertex u in $\mathcal{L}$
$\mathcal{F}(x)$ ($\mathcal{F}(A)$)	followers of an anchor x (A)
$CF(x)$	the candidate followers of an anchor x
$d^+(u)$	degree upper bound of u in $C_k(G_x)$

According to the above definition, we have $C_{k+1}(G) \subseteq C_k(G)$ for any valid value of k [12]. As shown in Algorithm 7, the k-core of a graph G can be obtained by recursively removing the vertices whose degrees are less than k, with a time complexity of $\mathcal{O}(m)$. The *core number* of a vertex $u \in G$ is the highest core where u appears. In this chapter, we use k-shell to denote the vertices with the core number k.

Definition 10. ***k-shell.*** *Given a graph G, the k-shell of G, denoted by $S_k(G)$, is the set of vertices with the core number of k; that is, $S_k(G) = C_k(G) \setminus C_{k+1}(G)$.*

Example 9. *In Figure 4.1, we have 3-core $C_3(G)$ induced by $\{u_4, u_5, u_8, u_9\}$, 2-core $C_2(G)$ induced by $\{u_1, u_2, ..., u_{11}\}$, and 2-shell $S_2(G) = \{u_1, u_2, u_3, u_6, u_7, u_{10}, u_{11}\}$.*

Algorithm 7: **ComputeCore**(G, k)

Input : G : a social network, k : degree constraint
Output: $C_k(G)$

```
while exists u ∈ G with deg(u, G) < k do
    G:= G \ {u};
return G
```

In this chapter, once a vertex u in G is **anchored**, it is always retained in k-core regardless of the number of neighbors, i.e., $deg(u, G) = +\infty$ if $u \in A$.

Definition 11. ***anchored k-core****. Given a graph G and a vertex set $A \subseteq G$, the anchored k-core, denoted by $C_k(G_A)$, is the corresponding k-core of G with vertices in A anchored.*

According to the definition of vertex degree, the computation of k-core with anchors is exactly the same as the k-core computation without anchors.

In addition to the anchored vertices in A and vertices in $C_k(G)$, more vertices might be retained in the $C_k(G_A)$ due to the contagious nature of the k-core computation. These vertices are called **followers** of the anchor vertices A, denoted by $\mathcal{F}(A, G)$, because they will not appear in k-core without the underpinning of A. The size of the followers reflects the effectiveness of the anchor vertices, where $\mathcal{F}(A, G) = C_k(G_A) \setminus \{C_k(G) \cup A\}$. In the following, we may use *anchor* to represent the *anchor vertex*, and we use $\mathcal{F}(A)$ to denote $\mathcal{F}(A, G)$ when the context is clear.

Problem Statement. Given a graph G, a degree constraint k and a budget b, the **anchored k-core problem** aims to find a set A of b vertices in G such that the size of the resulting anchored k-core, $C_k(G_A)$, is maximized; that is, $\mathcal{F}(A, G)$ is maximized.

Example 10. *In Figure 4.1, we have 3-core* $C_3(G) = \{u_4, u_5, u_8, u_9\}$. *If we set* $A = \{u_1\}$, *we have* $C_3(G_A) = \{u_1, u_2, ..., u_9\}$ *and* $\mathcal{F}(A) = \{u_2, u_3, u_6\}$. u_1 *is the best anchor if* $b = 1$, *and* $\{u_1, u_{10}\}$ *is the set of best anchors if* $b = 2$.

4.2.2 Problem Complexity

Given a set A of anchor vertices, we can immediately use a linear algorithm to compute $C_k(G_A)$ by not considering any vertex in A at Line 1 of Algorithm 7. However, it is very challenging to find the optimal A. As shown in [14], when $k \geq 3$ the problem of anchored k-core is NP-hard and W[2]-hard w.r.t the budget b. This implies that there is no non-trivial polynomial-time approximation algorithm even for $k > 2$, not mentioning the exact solution. In this chapter, we adopt the greedy heuristic. Not surprisingly, the greedy algorithm may fail in some particular cases. For example, a graph G consists of two separate subgraphs G_1 and G_2. Specifically, G_1 is a chain of $\lceil n/2 \rceil + 1$ vertices with $V(G_1) = \{v_1, v_2, ..., v_{\lceil n/2 \rceil + 1}\}$ and $E(G_1) = \{(v_i, v_{i+1}) \mid v_i, v_{i+1} \in V(G_1) \ \& \ 1 \leq i \leq \lceil n/2 \rceil\}$. When $k = 2$ and $b = 2$, the greedy algorithm will never choose an anchor x from $V(G_1)$ because $\mathcal{F}(x, G) = 0$. However, anchoring v_1 and $v_{\lceil n/2 \rceil + 1}$ can immediately get $\lceil n/2 \rceil - 1$ followers. We can infer that if there is a large subgraph in which most vertices can only becomes followers by anchoring a set U of vertices simultaneously, and anchoring a single vertex in U can not get enough followers, then the greedy algorithm will fail.

The inapproximability of the problem motivated the authors in [14] to develop a polynomial-time algorithm in graphs with bounded tree-width. However, this assumption does not hold in many real-life graphs (e.g., social networks). This motivated us to develop efficient heuristic algorithms to tackle the problem of anchored k-core on general graphs, and significantly improve performance by imposing an *onion layer* structure.

4.3 Our Approach

This section presents our onion-layer based anchored k-core (OLAK) algorithm to effectively and efficiently find a set of anchors for the anchored k-core problem. In Section 4.3.1, we briefly introduce the motivation behind our *onion layer* based techniques. Section 4.3.2 shows how to limit the number of candidate anchors, and Section 4.3.3 presents efficient algorithms to compute the number of followers for a given anchor. Section 4.3.4 further develops new pruning techniques to reduce the number of candidate anchors, and present our OLAK algorithm by integrating these new techniques.

4.3.1 Motivation

A straightforward solution for the anchored k-core problem is to exhaustively enumerate all possible set A with size b, and compute the resulting anchored k-core for each possible A. The time complexity of $\mathcal{O}(\binom{n}{b}m))$ is cost-prohibitive. Considering the hardness of the problem, we resort to a greedy heuristic which iteratively finds the best anchor vertex, i.e., the vertex with the largest number of followers. A straightforward implementation of the greedy algorithm is shown in Algorithm 8. The time complexity is $\mathcal{O}(bnm)$, where n and m correspond to the number of candidate anchors in each iteration (Line 3) and the cost of follower computation (Line 4). Note that we exclude the vertices in $C_k(G)$ at Line 3 because they are already in k-core.

Although the greedy algorithm does not have the sub-modular property due to the inapproximability of the problem, our empirical study shows its resulting anchor vertices have similar numbers of followers compared to that of the exact solution. However, a simple implementation of the greedy algorithm is still unscalable on large-scale networks. In this chapter, we aim to significantly

Algorithm 8: **GreedyAK**(G, k, b)

Input : G : a social network, k : degree constraint,
b : number of anchor vertices
Output: A : the set of anchor vertices

```
1 A := ∅; i := 0;
2 while i < b do
3     for each u ∈ G \ {A ∪ C_k(G)} do
4         Compute F(A ∪ u,G);
5     u* ← the best anchor vertex in this iteration;
6     A := A ∪ u*; i := i + 1;
7 return A
```

improve the two components of the greedy algorithm: (i) the number of candidate anchors in each iteration (Line 3); and (ii) the computation cost of finding followers (Line 4), which is determined by the number of candidate followers.

Motivated by this, we propose an auxiliary structure $\mathcal{L}$, namely *onion layers*, to facilitate the computation such that we can significantly reduce the number of candidate anchors and candidate followers. At a high level, $\mathcal{L}$ consists of a subset of vertices such that we only need to consider the vertices within $\mathcal{L}$ as the candidate anchors. Moreover, the vertices within $\mathcal{L}$ are organized by different layers. Another nice property of the layer structure is that, by exploiting the layer structure, we can effectively bound the region (i.e., candidate followers) influenced by an anchor. We also develop early termination techniques based on the layer structure to eliminate non-promising candidate followers.

In this section, we will focus on the problem of anchored k-core with $b = 1$, i.e., finding the *best* anchor which has the largest number of followers. In Section 4.3.4 we show the proposed algorithm can be immediately used in each iteration of Algorithm 8 by considering the previously anchored vertices. Note that the enhanced greedy algorithm produces the same result as Algorithm 8 because they follow the same greedy heuristic.

4.3.2 Reducing the Number of Candidate Anchors

The *onion layers* of G, denoted by $\mathcal{L}$, consists of the vertices in (k-1)-shell and their neighbors that are not in k-core; that is, $\mathcal{L} := S_{k-1}(G) \cup \{NB(S_{k-1}(G), G) \setminus C_k(G)\}$. Below, we show that only the vertices in $\mathcal{L}$ need to be considered, and that $F(u)$ is empty for every $u \notin \mathcal{L}$.

Theorem 9. *Given a graph G and its (k-1)-shell $S_{k-1}(G)$, if a vertex x is anchored, all of its followers come from (k-1)-shell; that is, $u \in F(x, G)$ implies $u \in S_{k-1}(G)$.*

Proof. We prove correctness by contradiction. The intuition is that if a follower comes from a k'-shell with $k' < k - 1$, we show that it belongs to (k-1)-shell instead.

Let M and N be the k-core and (k-1)-core of G respectively before anchoring the vertex x. As a follower u cannot come from $C_k(G)$, u has a core number k' with $k' < k - 1$ if $u \notin S_{k-1}(G)$. 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 $deg(v, M' \setminus \{x\}) \geq k - 1$ for every vertex $v \in NB(x, M')$ because $deg(v, M') \geq k$ and only one edge is removed from v. This means the deletion of x will not be cascaded since all of its neighbors in M' stay in the computation of $C_{k-1}(M' \setminus \{x\})$. Consequently, all vertices in $M' \setminus \{x\}$ satisfy the $k - 1$ degree constraint and hence $M' \setminus \{x\} \subseteq C_{k-1}(G)$. As $u \in M'$ and $u \neq x$, we have u which belongs to $C_{k-1}(G)$ and this contradicts with the fact that the core number of u is smaller than $k - 1$. □

The following theorem significantly reduces the size of the candidate anchors.

Theorem 10. *Given a graph G, if an anchored vertex x has at least one follower, x is from $\mathcal{L}$; that is, $|F(x, G)| > 0$ implies that $x \in S_{k-1}(G) \cup \{NB(S_{k-1}(G), G) \setminus C_k(G)\}$.*

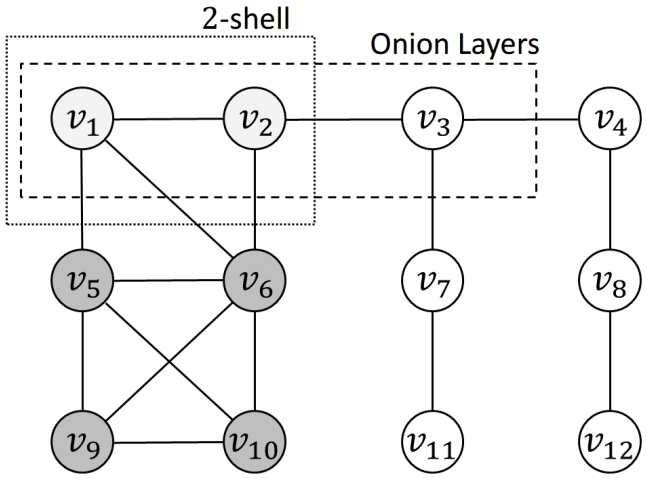


Figure 4.3: Bounded Anchors and Followers

Proof. $F(x)$ is the follower set of x and suppose $F(x) \neq \emptyset$. Additionally, $NB(x) \cap F(x) \neq \emptyset$, otherwise, for every x's neighbor u and $u \notin C_k(G)$, we have $u \notin C_k(G_x)$, which leads to $C_k(G) = C_k(G_x)$ and thus $F(x) = \emptyset$. Since $F(x) \subseteq S_{k-1}(G)$ by Theorem 9, x is in $S_{k-1}(G)$ or at least one of its neighbors in $S_{k-1}(G)$, i.e., $x \in S_{k-1}(G) \cup \{NB(S_{k-1}(G), G) \setminus C_k(G)\}$. □

Example 11. *In Figure 4.3 with $k = 3$, we have 3-core $C_3(G) = \{v_5, v_6, v_9, v_{10}\}$, 2-shell $S_2(G) = \{v_1, v_2\}$ and onion layers $\mathcal{L} = \{v_1, v_2, v_3\}$. By Theorem 9, followers of any vertex are inside $S_2(G)$. Promising anchors are inside $\mathcal{L}$ by Theorem 10. Consequently, to find the best anchor, we only consider the vertices in $\mathcal{L}$ as candidate anchors, which are $\{v_1, v_2, v_3\}$ in this example. Once a vertex u is anchored, only vertices $\{v_1, v_2\}$ may become followers.*

4.3.3 Efficiently Finding Followers

In this section, we develop efficient algorithms to compute followers for a chosen anchor vertex. A straightforward implementation is to directly apply the k-core computation algorithm (Algorithm 7) with the existence of the anchor. As an

alternative, one may extend the continuous k-core maintenance algorithms [60, 72, 99] by setting the core number of the anchor vertex as infinite and then update core numbers for other vertices. A vertex with core number increased to k is a follower. This greatly improves the computational cost. Nevertheless, we show the performance can be significantly enhanced by using the well-organized structure of *onion layers*.

In the experiments, we observe that the size of candidate anchors in Theorem 10 is still considerably large and there are many unavoidable attempts to find the best anchor vertex. This implies that it is worthwhile to carefully build an auxiliary data structure to facilitate the computation of followers for all candidate vertices. Specifically, we revisit k-core computation and design the *onion layer* structure of $\mathcal{L}$ such that the computation of the followers in each attempt can be greatly enhanced.

The Onion Layer Structure

We notice that the k-core computation (Algorithm 7) does not explicitly consider the deletion (i.e., leave) order of the non k-core vertices. We say the deletion order of an instance of k-core computation is *valid* if (1) the vertex violates the degree constraint at the time it is deleted; and (2) all remaining vertices satisfy the degree constraint when the deletion stops. Theorem 11 below shows that any *valid order* will come up with the k-core.

Theorem 11. *Algorithm 7 always returns the same $C_k(G)$ w.r.t any valid deletion order of non k-core vertices.*

Proof. Suppose there are two different valid deletion orders, O_1 and O_2, leading to two different k-cores C_1 and C_2, respectively. Let $M = C_1 \setminus C_2$ and $M \neq \emptyset$. This implies that all vertices in M are discarded in the access order O_2. Suppose u_1 is the first removed vertex in M, we have $deg(u_1, M \cup C_2) \geq k$ because

Algorithm 9: **OnionPeeling**(G, k)

Input : G : a social network, k : degree constraint
Output: *onion layers* $\mathcal{L}$ (i.e., L_0^s)

```
1 N := C_{k-1}(G); i := 0;
2 P := {u | deg(u, N) < k & u ∈ N};
3 while P ≠ ∅ do
4 |   i := i + 1; L_i := P;
5 |   N := N \ P;
6 |   P := {u | deg(u, N) < k & u ∈ N};
7 L_0 := {u | u ∈ NB(L_1^i, G) \ {N ∪ L_1^i}};
8 return L_0^i
```

$deg(u_1, C_1) \geq k$ and none of the vertices in C_2 or M are removed when u_1 is accessed. This implies O_2 is not a valid order. □

Theorem 11 motivates us to impose an *onion layer* structure on $\mathcal{L}$ to facilitate computation of the followers. $\mathcal{L}$ consists of $s+1$ layers, $\{L_0, L_1, \ldots, L_s\}$ ($L_0^s = \mathcal{L}$), produced by an *onion-peeling-like* algorithm. The pseudo-code is shown in Algorithm 9. We first compute $C_{k-1}(G)$ at Line 1, then start to peel the (k-1)-shell by removing *all* vertices not satisfying the degree constraint at the same time (Lines 2 and 6), which are kept in the same layer (Line 4). When the peeling process terminates, we have $i = s$, $\mathcal{L} = L_0^s = S_{k-1}(G)$ and $N = C_k(G)$. Then we put the neighbors of $S_{k-1}(G)$ (excluding the ones in $C_k(G)$ and L_1^s) to L_0 as the highest layer (Line 7). In this chapter, we use L_i^j ($i < j$) to denote the vertices between layer i and layer j (inclusive), and $l(u)$ to denote the layer index of a vertex u in $\mathcal{L}$.

Algorithm Correctness. We show that $S_{i-1}(G)$, obtained in Algorithm 9, is correct; that is, N is $C_k(G)$ when the peeling process (Lines 3-6) terminates. We can choose an arbitrary deletion order for the vertices in the same layer since they do not satisfy the degree constraint. On the other hand, when the peeling process terminates (i.e., $V = \emptyset$ at Line 6), we have $deg(u, N) \geq k$ for every vertex

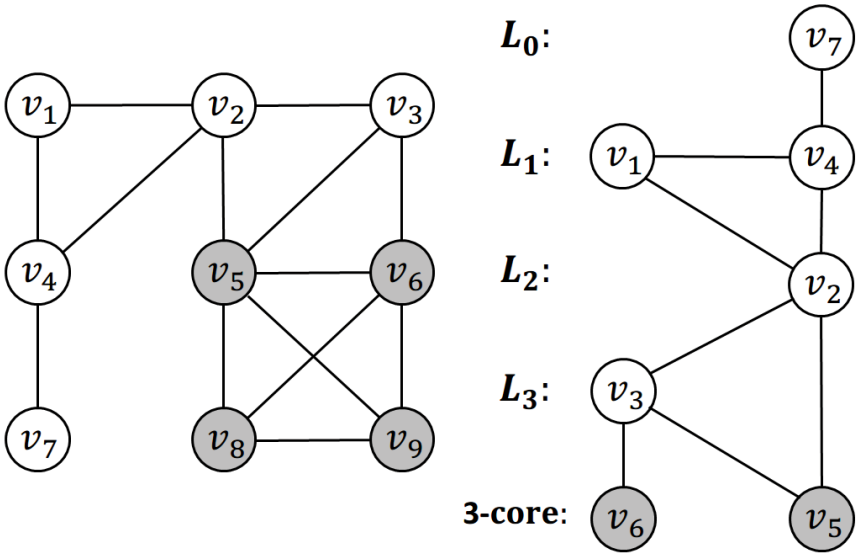


Figure 4.4: Onion Layer Structure (L_0^3)

$u \in N$. Therefore, we can get a valid deletion order and hence N is $C_k(G)$ when the peeling process terminates.

The time complexity of Algorithm 9 is $\mathcal{O}(m)$ in the worst case. Although we need to update the *onion layer* structure in each iteration of the greedy algorithm, this cost is greatly amortized by a considerably large number of follower computations.

Example 12. *In Figure 4.4 with k=3, we have 3-core* $C_3 = \{v_5, v_6, v_8, v_9\}$ *and 2-core* $C_2 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_8, v_9\}$. *By checking degree constraint against the vertices in* C_2, *we have* $deg(v_1, C_2) < 3$ *and* $deg(v_4, C_2) < 3$. *So the layer* $L_1 = \{v_1, v_4\}$. *After deleting* v_1 *and* v_4 *from* C_2, *we have* $deg(v_2, C_2) < 3$ *and* $L_2 = \{v_2\}$. *Iteratively, after deleting* v_2 *from* C_2, *we have* $deg(v_3, C_2) < 3$ *and* $L_3 = \{v_3\}$. *After deleting* v_3 *from* C_2, *3-core computation is finalised. Note that for original 2-core* C_2 *we have* $v_7 \in NB(C_2) \setminus C_2$. *Thus,* $L_0 = \{v_7\}$ *and all layers are generated.*

Onion Layer based Follower Computation

Now, we present an algorithm to efficiently compute the followers for a given anchor x based on the *onion layer* structure. The algorithm has two key techniques: (1) *finding candidate followers*, which explores the candidate followers for the anchor x; and (2) *early termination*, which recursively discards the non-promising candidates during the computation.

(1) Finding candidate followers. We first introduce the concept of a *support path*, and show how to find the candidate followers of an anchor x, denoted by $CF(x)$.

Definition 12. ***Support Path.*** *We say there is a support path for a vertex $u \in L_1^s$ w.r.t a given anchor vertex x if there is a path $x \rightsquigarrow u$ such that all vertices are from L_1^s and we have $l(y) < l(z)$ for every two consecutive vertices y and z along this path. Note that $l(u)$ is the layer index of the vertex $u \in \mathcal{L}$.*

Theorem 12. *A vertex $u \in S_{k-1}(G)$ is a follower of the anchor x (i.e., $u \in CF(x)$) implies that there is a support path $x \rightsquigarrow u$.*

Proof. According to the definition of vertex degree (note that $deg(u, G) = +\infty$ if $u \in A$), we can immediately employ the onion-peeling algorithm (Lines 1-6 in Algorithm 9) to compute $C_k(G_x)$ in which the vertex x is anchored. In the computation of $C_k(G)$ (without any anchors), all vertices in L_1^s are removed in Algorithm 9, while some of them (i.e., followers) may survive the computation of $C_k(G_x)$ (with anchoring x). Let i denote the layer index of x ($i = l(x)$). In the computation of $C_k(G_x)$, for a vertex $u \in L_1^{i-1}$, when u is accessed, the degree of u is less than k because the degree at current time is exactly the same as u is accessed in the computation of $C_k(G)$. So all vertices in L_1^{i-1} are deleted. Then, when vertices in L_1^{i-1} have been deleted and no vertex in L_i has been deleted, for every vertex $v \in L_i$, the degree of v is less than k because the degree at current

time is the same as in the computation of $C_k(G)$. Consequently, all vertices in $L_1^i \setminus \{x\}$ cannot follow x and are deleted. At this point, only neighbors of x in L_{i+1}^s become candidate followers and clearly they have support paths. For a candidate follower y, only its neighbors in L_{j+1}^s $(j = l(y))$ become candidate followers because y cannot save other vertices in the computation of $C_k(G_x)$, i.e., the degrees of other vertices are still less than k with the existence of y. Consequently, the candidate spread from x is strictly a top-down search through x's edges and candidates' edges, which constitute support paths. For a vertex z without any support paths, when non-candidate vertices in L_1^{p-1} $(p = l(z))$ have been deleted and no vertex in L_p has been deleted, the degree of z is less than k because it is same as z is accessed in the computation of $C_k(G)$. Consequently, every candidate follower of x has at least one support path which implies there is always a support path for a follower of x. □

According to the above theorem, we may generate candidate followers by iteratively activating the neighbors at lower level of $\mathcal{L}$. In this chapter, we use $CF(x)$ to denote all candidate followers of an anchor x obtained based on Theorem 12.

Example 13. *In Figure 4.4 with k=3, we have 3-core* $C_3 = \{v_5, v_6, v_8, v_9\}$ *and* L_0^3 *shown on the right side. If* v_1 *is anchored, only* v_2 *will be activated as a candidate follower, because* v_2 *is a neighbor of* v_1 *and is on a lower layer. Although* v_4 *is also a neighbor of* v_1*,* v_4 *will not be a candidate follower because they are on the same layer, i.e.,* $l(v_1) = l(v_4)$*. Similarly, after* v_2 *becomes a candidate follower,* v_3 *will be a candidate follower as well, while* v_4 *will not. Thus, we only need to consider the vertices on* $\{v_1, v_2, v_3\}$ *for follower computation.*

(2) Early termination. We remark that existing a *support path* is a necessary condition for a follower, and hence we need to conduct k-core computation on

$CF(x) \cup C_k(G) \cup \{x\}$ to identify the true followers. To avoid this, we introduce an early termination technique to prune the search space. In this technique, we find the candidate followers of an anchor x in a layer-by-layer fashion, i.e., for all the vertices which have been found to be inside of $CF(x)$ and are waiting to be explored, we explore the vertex with the smallest layer number first (ties are broken by the vertices' IDs).

In the layer-by-layer seach, each vertex in $\mathcal{L}$ has **three statuses**. We say a vertex u is **unexplored** if it has not been checked with the degree constraint in our layer-by-layer traversal. A vertex is **survived** if it has survived the degree check, otherwise it becomes **discarded**. For a given anchor, a *discarded* vertex will never be involved in the following computation, and a *survived* vertex may become *discarded* later due to the deletion cascade. Note that some vertices are *implicitly* marked as *discarded* since they are never accessed due to the candidate followers pruning technique.

We use $d^+(u)$ to denote the degree upper bound of a vertex u in $C_k(G_x)$. Specifically, $d^+(u) = d_s^+(u) + d_u^+(u) + d_c(u)$ where $d_s^+(u)$ (resp. $d_u^+(u)$) is the number of *survived* (resp. *unexplored*) neighbors in $\mathcal{L}$ and $d_c(u)$ is the number of neighbors in $C_k(G)$. The following theorem indicates that we can safely exclude a candidate follower u if $d^+(u) < k$. The removal of a vertex may invoke the deletion of other vertices, where details are described in Algorithm 10. When the shrink function terminates, all of the vertices affected by the removal of u will be correctly updated.

Theorem 13. *A vertex $u \in L_1^s$ cannot be a follower if $d^+(u) < k$.*

Proof. Neighbors of a vertex u can be classified into four disjoint sets N_0, N_1, N_2 and N_3. N_0 denotes the set of neighbors *not in* $C_{k-1}(G)$. N_1 (resp. N_2) denotes the explored (resp. *unexplored*) neighbors in $\mathcal{L}$, and N_3 denotes the neighbors from $C_k(G)$. Clearly, none of the neighbors in N_0 contributes degree support to

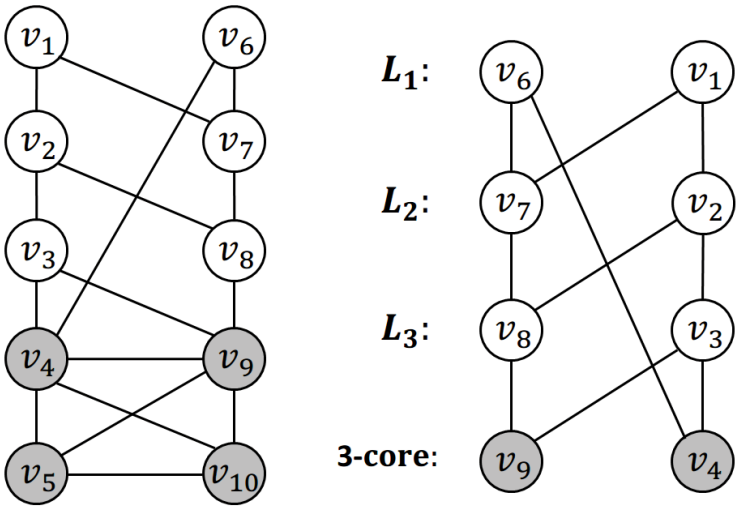


Figure 4.5: Early Termination

u because they have been discarded during the computation of $C_{k-1}(G)$. Once a vertex in $\mathcal{L}$ is *explored*, it will be marked as either *survived* or *discarded*, and a *discarded* vertex cannot provide degree support to u w.r.t $C_k(G_x)$. Therefore, $d_s^+(\mathrm{u})$ is correct. Moreover, we have $d_u^+(u) = |N_2|$ and $d_c(u) = |N_3|$. Consequently, the degree of u in $C_k(G_x)$ is bounded by $d^+(u)$. □

Example 14. *In Figure 4.5 with k=3, we have 3-core* $C_3 = \{v_4, v_5, v_9, v_{10}\}$ *and* L_1^3 *shown on the right side. If* v_1 *is anchored, the candidate follower set without early termination technique is* $\{v_2, v_3, v_7, v_8\}$*. The candidates starts from* v_1 *which puts* v_7 *and* v_2 *in the waiting list for future exploration. Then* v_7 *is explored and we have* $d_s^+(v_7) = 1$*,* $d_u^+(v_7) = 1$ *and* $d_c(v_7) = 0$*. Since* $d^+(v_7) < 3$*,* v_7 *is discarded. Then* v_2 *is explored and we have* $d^+(v_2) = 3$ *which means* v_8 *and* v_3 *can be put in the waiting list. Similarly,* v_8 *is explored and discarded which leads to the deletion of* v_2 *and* v_3*. Thus,* v_1 *does not have any followers.*

(3) Finding Followers. Algorithm 11 lists the pseudo-code of the follower computation for a chosen anchor x. A min heap H is used to keep the candidate

Algorithm 10: **Shrink**(u)

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

Algorithm 11: **FindFollowers**(x, $\mathcal{L}$)

```
   Input : x : the anchor; L : onion layers
   Output: F : the followers of x
1  H := ∅; H.push(x);
2  while H ≠ ∅ do
3      u ← H.pop();
4      Compute d+(u);
5      if d+(u) ≥ k then
6          u is set survived;
7          for each v ∈ NB(u) ∪ L and l(v) > l(u)
8          and v ∉ H do
9              H.push(v);
10     else
11         u is set discarded ;
12         Shrink(u);
13 return survived vertices in L \ {x}
```

followers, and the key of a vertex u is $l(u)$ with ties broken by the vertices' IDs. In this way, we explore the candidates in a layer-by-layer fashion and it is easy to check whether a vertex u has been *explored* based on its ID and layer index $l(u)$. For each popped vertex u, Line 4 computes its degree upper bound $d^+(u)$. If u survives the degree check or u is the anchor x, u will be set to *survived* (Line 6) and its neighbors in lower layers (i.e., unexplored candidate followers) will be pushed into H if they are not already in H (Lines 8-9). Otherwise, u is

set to *discarded* and the early termination process is invoked (Lines 11-12). The deletion may be cascaded and some *survived* vertices may be set to *discarded* during the process. When the algorithm terminates, all *survived* vertices in $\mathcal{L} \setminus \{x\}$ are the followers of x. The time complexity of the algorithm is $\mathcal{O}(m)$ in the worst case because each edge is at most accessed three times: to push the neighbors into H, compute the degree upper bound and compute the cascade of the deletion.

Algorithm Correctness. We show the deletion of the vertices in Algorithm 11 has a *valid order* O for the computation of $C_k(G_x)$. A vertex u may be *implicitly* deleted if (1) $u \notin CF(x)$; or (2) $u \in CF(x)$, but u is not pushed into H because some of vertices on its support path has been set to *discarded*. We assume all these vertices on the layer i are deleted in O right before the first vertex on this layer is popped from H. The correctness of case (1) is immediate since $u \notin CF(x)$. In case (2), we conclude that there does not exist a support path for u in which all vertices are followers. Using similar rationale to Theorem 12, u cannot be supported by x and hence can be safely discarded. A vertex u may also be *explicitly* deleted in O if (3) u is set *discarded* at Line 11 because it fails the degree check when it is popped; or (4) $d^+(u)$ deceases below k due to the deletions of the other vertices (Line 5 of Algorithm 10). Because $d^+(u)$ is correctly computed (Line 4) and maintained (Algorithm 10), u does not satisfy the degree constraint when u is deleted in cases (3) and (4). Let M denote $C_k(G)$ and L' denote the remaining *survived* vertices when Algorithm 11 terminates. Now, we show that none of the vertices in L' can be discarded. As all of the vertices in $\mathcal{L}$ have been *explored* explicitly or implicitly, we have $d^+(u) = deg(u, L' \cup M)$ for every vertex $u \in L'$ since $d_s^+(u) = 0$, $d_u^+(u) = deg(u, L')$ and $d_c(u) = deg(u, M)$. As $d^+(u) \geq k$ for every vertex $u \in L'$, we have $deg(u, L' \cup M) \geq k$ and none of the vertices in $L' \cup M$ can be discarded. So O is a *valid order* and $L' \cup M = C_k(G_x)$.

Remark 5. *Note that we can also apply the onion layer structure to facilitate continuous core maintenance [99]. In this problem, when a new edge is inserted, we can set the corresponding vertex, whose core number increases to k, as an anchor. With the similar rationale, the layer structure can be used to reduce the search region of the candidate followers, whose core values may be updated. However, it is not cost effective because the onion layer structure must be updated after every insertion of an edge, and the new edges may arrive in a streaming fashion.*

4.3.4 The OLAK Algorithm

In this section, we introduce two pruning techniques to further reduce the number of candidate anchors. Then we present our `OLAK` algorithm to find the best anchor in graph G and show how to handle the case with multiple anchors.

Follower based Pruning

The following theorem indicates that, to find the best anchor, we do not need to consider an anchor if it is a follower of another anchor; that is, an anchor u is shadowed by x if $u \in \mathcal{F}(x)$.

Theorem 14. *Given two vertices x and u in $\mathcal{L}$, we have $|\mathcal{F}(x)| > |\mathcal{F}(u)|$ if $u \in \mathcal{F}(x)$.*

Proof. $u \in \mathcal{F}(x)$ implies that there is a support path $x \rightsquigarrow u$, and hence we have $CF(u) \subset CF(x)$ where $CF(u)$ and $CF(x)$ are candidate followers of u and x obtained by Theorem 12, respectively. $u \in \mathcal{F}(x)$ also implies that u has enough degree support when u is not anchored and x is anchored, i.e., $deg(u, \mathcal{F}(x) \cup \{x\} \cup C_k(G)) \geq k$. Consequently, every vertex v in $\mathcal{F}(u)$ will not be discarded in

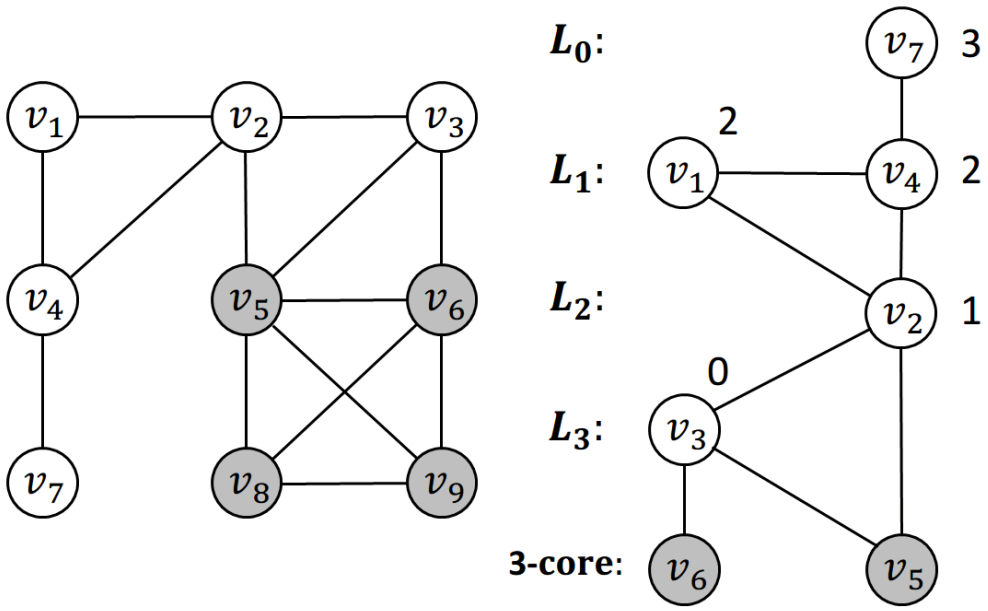


Figure 4.6: Followers Pruning and Upper Bound Pruning

the computation of $\mathcal{F}(x)$ since $deg(v, \mathcal{F}(u) \cup \{u\} \cup C_k(G)) \geq k$. So $\mathcal{F}(u) \subset \mathcal{F}(x)$ and then $|\mathcal{F}(x)| > |\mathcal{F}(u)|$. □

Example 15. *In Figure 4.6 with k=3, we have 3-core* $C_3 = \{v_5, v_6, v_8, v_9\}$ *and* L_0^3 *shown on the right side. In the procedure of finding a best anchor, if* v_1 *has been tried as an anchor, we have* $\mathcal{F}(v_1) = \{v_2, v_3\}$*. So neither* v_2 *nor* v_3 *can be a best anchor.*

Upper Bound based Pruning

Let $W(x)$ denote the neighbors of a vertex x in lower layers, i.e., $W(x) = \{u \mid u \in NB(x) \cap \mathcal{L}$ and $l(u) > l(x)\}$. We use $UB(x)$ to denote the upper bound of $|\mathcal{F}(x)|$, where

$$UB(x) = \begin{cases} \sum_{u \in W(x)} (UB(u) + 1) & \text{if } |W(x)| > 0; \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

The following theorem shows that we can accumulatively compute the upper bound of the number of followers for vertices in L_1^s. The correctness is evident since $|CF(x)| \leq UB(x)$ and $|\mathcal{F}(x)| \leq |CF(x)|$ for every vertex x.

Theorem 15. *Let λ denote the number of followers of the best anchor seen so far. We can exclude any candidate anchor x if $UB(x) < \lambda$.*

Example 16. *In Figure 4.6 with k=3, we have the 3-core $C_3 = \{v_5, v_6, v_8, v_9\}$ and L_0^3 shown on the right side. By Equation (4.1), we have $UB(v_3) = 0$, $UB(v_2) = 1$, $UB(v_1) = 2$, $UB(v_4) = 2$ and $UB(v_7) = 3$. In the procedure to find a best anchor, if v_1 is tried for anchoring and it becomes current best anchor with $|\mathcal{F}(v_1)| = 2$, we don't need to consider anchoring any vertices except for v_7 because their upper bounds do not exceed 2.*

Based on Equation (4.1), our implementation computes the upper bound of the follower size for each vertex in $\mathcal{L}$ in a bottom-up fashion with a time complexity of $\mathcal{O}(m)$. To get tighter upper bounds, we can replace the $UB(u)$ in Equation (4.1) with $CF(u)$ for each u whose $CF(u)$ has been computed. This does not pay off because the time complexity becomes $\mathcal{O}(nm)$ which reaches the complexity of the greedy algorithm for the anchored k-core problem with $b = 1$. It is also confirmed by initial experiments.

Combining the Elements

Algorithm 12 illustrates the details of `OLAK` which finds the best anchor vertex for a given graph G (i.e., $b = 1$). Particularly, we first apply Algorithm 9 to compute the *onion layers* of G (Line 1) and the upper bound of each vertex in $\mathcal{L}$ (Line 2). Initially, the candidate anchor set T is set to $\mathcal{L}$ according to Theorem 10. Then we sequentially access vertices in T based on their upper bounds of the number of followers in decreasing order, and compute their followers by

Algorithm 12: **OLAK**(G, k)

Input : G : a social network, k : degree constraint
Output: the best anchor vertex

```
1  Compute onion layers L (Algorithm 9);
2  Compute UB(x) for each vertex x ∈ L;
3  T ← L (Theorem 10); λ := 0;
4  for each x ∈ T with decreasing order of UB(x) do
5      if UB(x) < λ then
6          Break (Theorem 15);
7      F(x) ← FindFollowers(x, L) (Algorithm 11);
8      if F(x) ≠ ∅ then
9          T := T \ F(x) (Theorem 14);
10     if |F(x)| > λ then
11         λ := |F(x)|;
12 return the best anchor
```

Algorithm 11. According to the follower-based pruning, Line 9 excludes the followers of current accessed vertex from T. We continuously maintain the largest number of followers seen so far for one vertex, denoted by λ, which may eliminate some non-promising candidate anchors in T by upper bound based pruning (Line 6). We have the best anchor when the algorithm terminates.

To handle general cases where $b > 1$, our `OLAK` algorithm can easily fit within the greedy algorithm (Replacing Lines 3-4 of Algorithm 8) to find the best vertex in each iteration. The only difference is that we need to enforce that the anchored vertices in previous iterations remain in the k-core. Note that in order to avoid computing $C_{k-1}(G_A)$ (Line 1 of Algorithm 9) from scratch in each iteration, we adopt an existing core maintenance technique [99] to continuously maintain the (k-1)-core and the k-core after inserting a best anchor. Moreover, if the (k-1)-core consists of a set of disconnected subgraphs, we can avoid the re-computation of the followers of a subgraph in the next iteration unless there is a new anchor in this subgraph. The time complexity of the algorithm remains $\mathcal{O}(bnm)$ in the

worst case. Nevertheless, our empirical study shows we can significantly improve performance of the straightforward implementation (Algorithm 8) by at least 3 orders of magnitude, due to a much smaller number of candidate anchors and a more efficient follower computation algorithm.

Algorithm Correctness. (1) For the anchored k-core problem with $b = 1$ on graph G, we get the correct result immediately based on the correctness of proposed techniques. (2) Assume the algorithm is correct when $b = i$, $i \in N^+$ and returns the anchor set A. (3) Consider the problem with $b = i + 1$, now the k-core of G is $C_k(G_A)$ since we have $deg(u, G) \geq k$ for any $u \in C_k(G_A)$ (note that $deg(v, G) = +\infty$ for any $v \in A$) and $C_k(G_A)$ is maximal (according to Definition 21). Then the (k-1)-core is updated correctly by the core maintenance algorithm. Thus, we get the updated *onion layers* $\mathcal{L}$ correctly by Algorithm 9. Since all the techniques are based on $\mathcal{L}$, after running `OLAK` on G with $b = 1$ again, we get the correct result $A \cup \{x\}$ for the case of $b = i + 1$ on G. Note that in the (k-1)-core N of G, for every disconnected subgraph S with $S \in N$ and $x \notin S$, S keeps same after anchoring x, thus, the previous result of anchoring any vertex in S can be reused.

4.4 Prevent Collapse of Strong Tie Communities

Tie strength among users is a fundamental component in social networks and has been well studied [33, 42, 71, 81]. The model of k-core fits well with the communities which do not require tie strength, such as the study group mentioned in Section 3.1, where students may discuss with others, even though their relationships are not strong. Nevertheless, some community types need to consider the strength of ties, e.g., a programming team in a company, where every member would like enough familiar colleagues in this team to conduct better col-

laboration and avoid unexpected troubles. These communities are called strong tie communities which correspond to the model of k-truss [29] where each edge is contained in at least $k-2$ triangles in the subgraph. Using the numbers of triangles to detect strong ties has been widely verified to be effective [33, 71, 81].

Besides ensuring the strength of ties, the k-truss is also a subgraph of (k-1)-core and a subgraph of (k-1)-edge-connected subgraph where the latter remains connected whenever fewer than k-1 edges are removed in the subgraph. Recently, solving k-truss based community search and detection problems appears in view [47, 48, 100]. The decomposition of k-truss [50, 83, 104] conducts k-truss computation from an initial value 2 for k to the largest value such that the k-truss is not empty. It corresponds to the network collapse of strong tie communities where the tie strength threshold (i.e., the required value of k) can be tuned or defined by users. Consequently, we study the problem of anchored k-truss to prevent collapse of strong tie communities, which is formally introduced in the following sections.

4.4.1 The Problem of Anchored k-Truss

We summarize some new notations in Table 4.2, where remarkably we say a vertex u directly connects to an edge e, or e directly connects to u, if u is one of the endpoints of e.

Definition 13. ***k-truss.*** *Given a graph G, a subgraph J is the k-truss of G, denoted by $T_k(G)$, if (i) J satisfies support constraint, i.e., $sup(e, J) \geq k-2$ for every edge $e \in J$; and (ii) J is maximal, i.e., any subgraph $J' \supset J$ is not a k-truss.*

Note that we have $T_k(G) \subseteq C_{k-1}(G)$ and $T_k(G) \subseteq T_{k-1}(G)$ [29], which are also required in anchored k-truss. As shown in Algorithm 13, we firstly compute

Table 4.2: Summary of New Notations

Notation	Definition
e	an edge in the graph
$sup(e, G)$	number of triangles contain e in G
$T_k(G)$	k-truss of G
$H_k(G)$, $H_k^+(G)$	k-hull of G, k-hull$^+$ of G
$\triangle_{NB}(e, G)$	the set of vertices in the triangles which contain e on G, and do not directly connect e
$\triangle_{NB}(J, G)$	the set of vertices in $\triangle_{NB}(e, G)$ for every edge $e \in J$
$\mathcal{E}(S, G)$	the edge set in G where each edge in the set directly connects to at least a vertex in S
$\mathcal{V}(S, G)$	the vertex set in G where each vertex in the set directly connects to at least an edge in S
$\mathcal{L}$	*edge onion layers* of G

Algorithm 13: **ComputeTruss(G, k)**

Input : G : a social network, k : support constraint
Output: $T_k(G)$

```
1 G' := ComputeCore(G, k − 1);
2 while exists an edge e ∈ G' with sup(e, G') < k − 2 do
3     G' := G' \ {e};
4 Delete isolated vertices in G';
5 return G'
```

(k-1)-core $C_{k-1}(G)$ of a graph G, then recursively remove every edge whose support is less than $k-2$. We can get the k-truss after removing isolated vertices.

Definition 14. ***k-hull and k-hull$^+$.*** *Given a graph G, the k-hull of G is denoted by $H_k(G)$, and $H_k(G) = T_k(G) \setminus T_{k+1}(G)$; the k-hull$^+$ of G is denoted by $H_k^+(G)$, and $H_k^+(G) = H_k(G) \cup \mathcal{V}(H_k(G), G)$.*

The only difference between k-hull and k-hull$^+$ is that k-hull$^+$ further contains some vertices in k-truss which directly connect to an edge in k-hull and these vertices do not exist in k-hull. The edges in k-hull and k-hull$^+$ are same.

In the problem for k-truss, once a vertex x in G is **anchored**, it is always retained in k-truss. For every edge e which directly connects to x (e is called an **anchor edge**), e is retained in k-truss (denoted by T) if $sup(e, T) > 0$. This definition avoids to keep those relatively isolated vertices in anchored k-truss such as the anchor x's neighbors who are only adjacent to x.

Definition 15. ***anchored k-truss.*** *Given a graph G and a vertex set $A \subseteq G$, the anchored k-truss, denoted by $T_k(G_A)$, is the corresponding k-truss of G with vertices in A anchored.*

The computation of anchored k-truss is same with the computation of k-truss except that for anchor edges, we delete them if and only if their supports less than one. In addition to the vertices in $T_k(G)$, more vertices (i.e., **followers**, denoted by $\mathcal{F}(A, G)$) might be retained in the $T_k(G_A)$ due to the contagious nature of the k-truss computation. Formally, we have $\mathcal{F}(A, G) =$ the vertices in $T_k(G_A) \setminus T_k(G)$.

Problem Statement. Given a graph G, a support constraint k and a budget b, the **anchored k-truss problem** aims to find a set A of b vertices in G such that the size of resulting anchored k-truss, $T_k(G_A)$, is maximized; that is, $\mathcal{F}(A, G)$ is maximized.

4.4.2 The Problem Hardness

Theorem 16. *Given a graph G, the anchored k-truss problem is NP-hard when $k > 3$.*

Proof. When $k < 3$, the k-truss is the graph G. When $k = 3$, an edge exists in k-truss if its support is at least 1, thus no followers exist for anchoring any vertex according to the definition of anchor in Section 4.4.1. When $k > 3$, we reduce the anchored k-truss problem to the maximum coverage problem [52]; that is

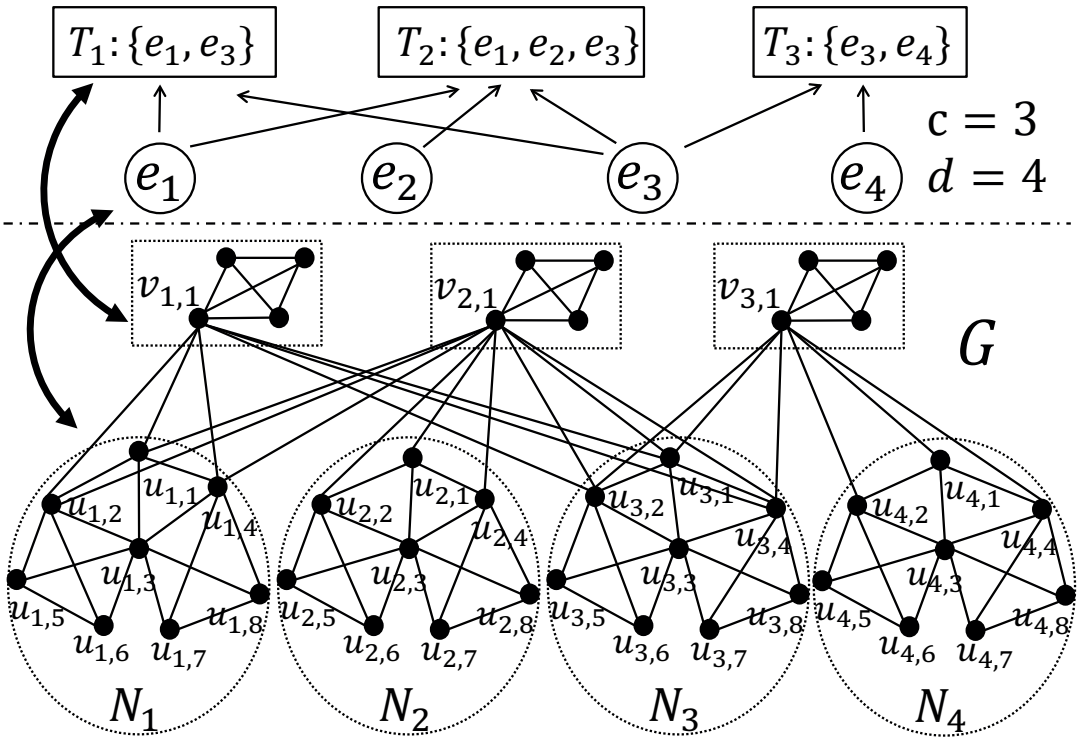


Figure 4.7: Construction Example for NP-hardness Proof, $k = 4$

finding at most b sets to cover the largest number of elements, where b is a given budget. We consider an arbitrary instance of maximum coverage problem with c sets $T_1, T_2, ..., T_c$ and d elements $\{e_1, e_2, ..., e_d\} = \cup_{1 \le i \le c} T_i$. Then we construct a corresponding instance of the anchored k-truss problem in a graph G as follows.

There are two sets of vertices in G, denoted by $\cup_{1 \le i \le c} M_i$ (M in short) and $\cup_{1 \le j \le d} N_j$ (N in short). Each subset $M_i (1 \le i \le c)$ contains k vertices, i.e., $M = \{v_{i,p} \mid 1 \le p \le k\}$, and forms a clique, every vertex pair in the set has an edge. Each subset $N_j (1 \le j \le d)$ contains $k + 4$ vertices, i.e., $N_j = \{u_{j,p} \mid 1 \le p \le k+4\}$. Particularly, vertex set $\{u_{j,p} \mid 1 \le p \le k\}$ form a lack-one-edge clique, i.e., there is an edge between each pair of vertices in the set, except for vertices $u_{j,2}$ and $u_{j,k}$. Besides, vertex set $\{u_{j,p} \mid 2 \le p \le k-1 \text{ or } k+1 \le p \le k+2\}$ and $\{u_{j,p} \mid 3 \le p \le k \text{ or } k+3 \le p \le k+4\}$ form a k-clique, respectively. For each

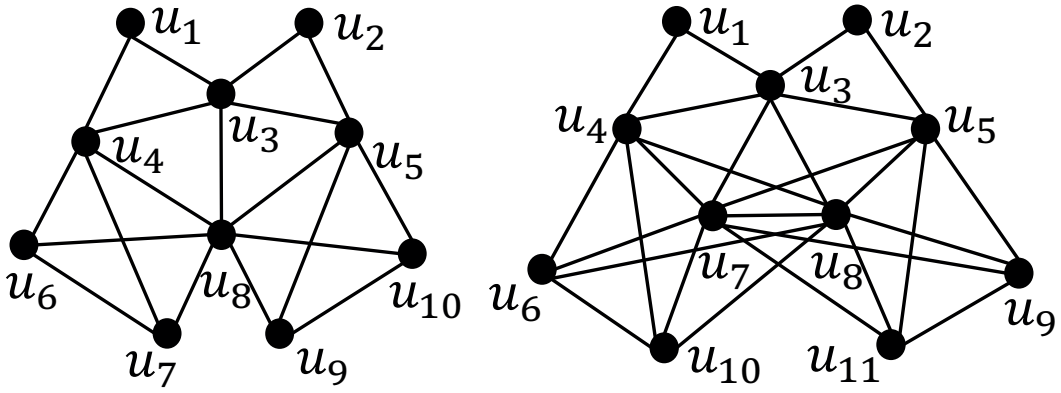
set $T_i(1 \leq i \leq c)$ and each element $e_j(1 \leq j \leq d)$, if $e_j \in T_i$, we add 1 edge from $v_{i,1}$ to $u_{j,1}, u_{j,2}$ and $u_{j,k}$, respectively.

With the construction, we ensure that for any i and j $(1 \leq i \leq c, 1 \leq j \leq d)$ (i) there is no edge between $v_{i,1}$ and $\{u_{j,p} \mid 3 \leq p \leq k-1 \text{ or } k+1 \leq p \leq k+4\}$, thus the supports of $e(v_{i,1}, u_{j,1})$, $e(v_{i,1}, u_{j,2})$ and $e(v_{i,1}, u_{j,k})$ are 2, 1 and 1, respectively; (ii) there is no edge between $u_{j,2}$ and $u_{j,k}$, thus the support of $e(u_{j,1}, u_{j,2})$ in N is $k-3$ because there are only $k-3$ common neighbors $\{u_{j,p} \mid 3 \leq p \leq k-1\}$. Similarly, the support of $e(u_{j,1}, u_{j,k})$ in N is $k-3$ for the same common neighbors with $e(u_{j,1}, u_{j,2})$; (iii) for each edge $e(u_{j,1}, u_{j,q})(3 \leq q \leq k-1)$, its support in G is $k-2$ because there are only $k-2$ common neighbors $\{u_{j,p} \mid 2 \leq p < q \text{ or } q < p \leq k\}$; (iv) all the supports of other edges are at least $k-2$ because they belong to at least 1 k-clique.

By doing this, we ensure that (i) the k-truss of G consists of $\cup_{1 \leq i \leq c} M_i$ and $\{u_{j,p} \mid 2 \leq p \leq k+4 \text{ and } 1 \leq j \leq d\}$ and corresponding edges; (ii) if $e_j \in T_i$ and $v_{i,1}$ is anchored, we have and only have $u_{j,1}$ as its follower in $T_k(G_A)$; (iii) anchoring a vertex in $G \setminus \{v_{i,1} \mid 1 \leq i \leq c\}$ cannot have any followers. Consequently, the optimal solution of anchored k-truss problem corresponds to optimal solution of the maximum coverage problem. Since the maximum coverage problem is NP-hard, the anchored k-truss problem is NP-hard for $k > 3$. □

Theorem 17. *Let $f(A) = |\mathcal{F}(A)|$. We have f is monotone but not submodular when $k > 3$.*

Proof. When $k \leq 3$, no followers exist for anchoring any vertex in G. When $k > 3$, suppose there is a set $A' \supseteq A$. For every vertex u in $\mathcal{F}(A)$, u will still exist in the anchored k-truss $T_k(G'_A)$, because anchoring more vertices in $A' \setminus A$ cannot decrease the supports of $\mathcal{E}(u)$ and the degree of u. Thus $f(A') \geq f(A)$ and f is monotone. For two arbitrary anchor sets A and B, if f is submodular, it must hold that $f(A \cup B) + f(A \cap B) \leq f(A) + f(B)$. We show that the



(a) $k = 4$ (b) $k = 5$

Figure 4.8: Examples for Non-submodular

inequality does not hold using counterexamples. As Figure 5.3 shows, we firstly construct a graph G consist of two vertices u_1, u_2 and the induced subgraph of N_1 in the proof of Theorem 16, here $N_1 = \{u_i \mid 3 \leq i \leq k+6\}$. After adding edges from u_1 to u_3 and u_4, and edges from u_2 to u_3 and u_5, the construction of G is completed. Suppose $A = \{u_1\}$ and $B = \{u_2\}$, we have $\mathcal{F}(A \cup B) = \{u_3\}$, $\mathcal{F}(A \cap B) = \mathcal{F}(A) = \mathcal{F}(B) = \emptyset$, so the inequation does not hold and thus f is not submodular. □

Considering the hardness of the problem, we resort to a greedy heuristic which iteratively finds the best anchor vertex, i.e., the vertex with the largest number of followers. Specifically, the greedy algorithm (Algorithm 14) is same as Algorithm 8 except that the input of k represents support constraint instead of degree constraint, and vertices in $T_k(G)$ are also candidate anchors since they can also have followers.

For ease of understanding, we discuss the problem on the first iteration of the greedy algorithm in the rest of the chapter. The following techniques can be directly applied to any iteration of the greedy algorithm when we substitute the k-truss with the anchored k-truss.

Algorithm 14: **GreedyAKT**(G, k, b)

Input : G : a social network, k : support constraint,
b : number of anchor vertices
Output: A : the set of anchor vertices

```
1 A := ∅; i := 0;
2 while i < b do
3     for each u ∈ G \ A do
4         Compute F(A ∪ u,G);
5     u* ← the best anchor vertex in this iteration;
6     A := A ∪ u*;i := i + 1;
7 return A
```

4.4.3 The Edge Onion Layers

The deletion order of non-k-truss edges in Algorithm 13 can be various because there can exist multiple edges with insufficient supports to wait for deletion. We say a deletion order of edges in k-truss computation is *valid* if (1) every edge in the order violates the support constraint at the time it is deleted; and (2) all the remaining edges satisfy the support constraint when the computation stops. Theorem 18 shows any valid order can arrive at the same k-truss.

Theorem 18. *Algorithm 13 always returns the same $T_k(G)$ w.r.t any valid deletion order of non k-truss edges.*

Proof. Suppose there are two different valid deletion orders, O_1 and O_2, leading to two different k-trusses T_1 and T_2, respectively. Let $M = T_1 \setminus T_2$ and $M \neq \emptyset$. This implies that all edges in M are discarded in the access order O_2. Suppose e_1 is the first removed edge in M, we have $sup(e_1, M \cup T_2) \geq k - 2$ because $sup(e_1, T_1) \geq k - 2$ and none of the edges in T_2 or M are removed when e_1 is accessed. This implies O_2 is not a valid order. □

Theorem 18 motivates us that we can utilize the inner property in a specific order to compute the anchored k-truss. An *edge onion layer* structure $\mathcal{L}$ based

Algorithm 15: **EdgeOnionPeeling**(G, k)

Input : G : a social network, k : support constraint
Output: The edge set of *edge onion layers*, i.e., L_0^s

1 $N := T_{k-1}(G)$; $i := 0$;
2 $P := \{e \mid sup(e, N) < k-2 \ \& \ e \in N\}$;
3 **while** $P \neq \emptyset$ **do**
4 $\quad i := i+1$; $L_i := P$;
5 $\quad N := N \setminus P$;
6 $\quad P := \{e \mid sup(e, N) < k-2 \ \& \ e \in N\}$;
7 $L_0 := \{(u, v) \mid u \in \triangle_{NB}(e, G) \setminus T_{k-1}(G)$ & v directly connects to e, for every $e \in H_{k-1}(G)\}$;
8 **return** L_0^i

on k-hull$^+$ is proposed to facilitate computation. $\mathcal{L}$ consists of $s+1$ layers of edges, $\{L_0, L_1, \ldots, L_s\}$ (i.e., L_0^s), and corresponding vertices which directly connect to at least one edge in L_0^s. The pseudo-code is shown in Algorithm 15. We first compute $T_{k-1}(G)$ at Line 1, then start to peel the (k-1)-hull by removing *all* edges not satisfying the support constraint at the same time (Lines 2 and 6), which are kept in the same layer (Line 4). When the peeling process terminates, we have $i = s$, the edge set of $\mathcal{L}$ is L_0^s which is same to the edge set of $H_{k-1}(G)$, and $N = T_k(G)$ after removing isolated vertices in N. Then we put the edges (excluding the ones in $T_{k-1}(G)$) between common neighbors of endpoints and the endpoints, for every edge in $H_{k-1}(G)$, to L_0 as the highest layer (Line 7). In this chapter, we use L_i^j $(i < j)$ to denote the edges between layer i and layer j (inclusive), and $l(e)$ to denote the layer index of an edge e in $\mathcal{L}$.

Definition 16. ***Edge Onion Layers: $\mathcal{L}$.*** *Given a graph G and a support constraint k, the edge set of $\mathcal{L}$ is L_0^s, the output of Algorithm 15; that is, $L_0^s = \{e \mid e \in H_{k-1}(G)\} \cup \{(u, v) \mid u \in \triangle_{NB}(e, G) \setminus T_{k-1}(G)$ & v directly connects to e, for every $e \in H_{k-1}(G)\}$. The full set $\mathcal{L}$ further contains the corresponding vertices, i.e., $\mathcal{L} = L_0^s \cup H_{k-1}^+(G) \cup \{\triangle_{NB}(H_{k-1}(G), G) \setminus T_{k-1}(G)\}$.*

4.4.4 Candidate Anchors and Followers

Theorem 19. *Given a graph G and its (k-1)-hull $H_{k-1}(G)$, if a vertex x is anchored, all of its followers except x come from (k-1)-hull; that is, $u \in F(x, G)$ implies $u \in H_{k-1}(G)$ or $u = x$.*

Proof. Let E^* be all the edges in $G \setminus T_{k-1}(G)$, We firstly prove that the edges in $E^* \setminus \mathcal{E}(x, G)$ will still be deleted in computing $T_k(G_x)$. Let O be the deletion order of edges in computing $T_{k-1}(G)$, the support of each edge is less than $k-3$ when it is to be deleted in the order of O. Then in the computation of $T_k(G_x)$, we still follow the order O. Because anchoring x can only increase the support of an edge in $E^* \setminus \mathcal{E}(x, G)$ by at most 1, the support of every edge in O is less than $k-2$ when it is just visited (i.e., deleted). Consequently, all edges in $E^* \setminus \mathcal{E}(x, G)$ cannot exist in $T_k(G_x)$. After computing $T_k(G_x)$, for every vertex u in $(G \setminus T_{k-1}(G)) \cap NB(x, G)$, the support of the anchor edge $e(u, x)$ is 0 because all of its adjacent non-anchor edges have been deleted. Consequently, all the edges in E^* are deleted in $T_k(G_x)$ and clearly the vertices in $G \setminus \{T_{k-1}(G) \cup x\}$ are also deleted. Moreover, a vertex in $T_k(G)$ cannot be a follower. □

Theorem 20. *Given a graph G, if an anchored vertex x has at least one follower besides x, x is from $\mathcal{L}$; that is, $|F(x, G)| > 0$ implies that $x \in \mathcal{L}$.*

Proof. According to the proof of Theorem 19, we can only save some $H_{k-1}(G) \cup \mathcal{E}(x, G)$ edges in $T_k(G_x)$. If the vertex x has at least one follower, x should involve in at least one triangle which contains an edge in $H_{k-1}(G)$, i.e., $x \in \triangle_{NB}(H_{k-1}(G), G)$ or $x \in H^+_{k-1}(G)$; otherwise, we cannot save any edges outside $T_k(G)$ into $T_k(G_x)$. If the anchor $x \in T_k(G)$, there is at least one edge in $\mathcal{E}(x, H_{k-1}(G))$, which implies $x \in H^+_{k-1}(G)$. So we have $x \in H^+_{k-1}(G) \cup \{\triangle_{NB}(H_{k-1}(G), G) \setminus T_k(G)\}$. Because $T_{k-1}(G) \setminus T_k(G) \subseteq H^+_{k-1}(G)$, we have $x \in H^+_{k-1}(G) \cup \{\triangle_{NB}(H_{k-1}(G), G) \setminus T_{k-1}(G)\}$, i.e., $x \in \mathcal{L}$. □

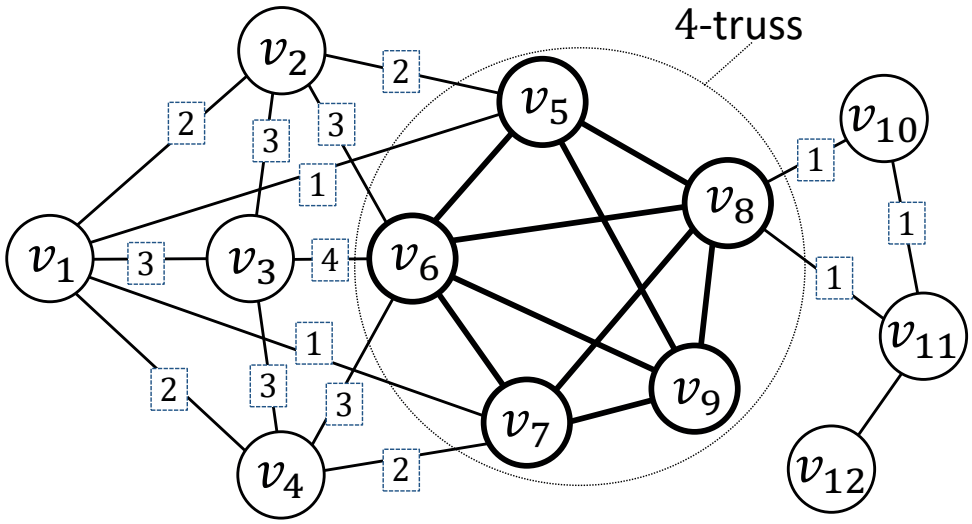


Figure 4.9: Examples for Edge Onion Layers $\mathcal{L}$, $k = 4$

Example 17. *In Figure 4.9 with k=4, we have 3-truss T_3 is the whole graph except vertex v_{12} and edge (v_{11}, v_{12}), and the 4-truss T_4 induced by $\{v_5, v_6, v_7, v_8, v_9\}$. We label every edge e in L_0^s with its layer index $l(e)$. Note that $\mathcal{L}$ also contains all the endpoints of the labeled edges. According to Theorem 19, only vertices in $T_3 \setminus T_4$ and the anchor can be followers of an anchor. By Theorem 20, if a vertex x has at least one follower besides x, x comes from $\mathcal{L}$.*

4.4.5 Efficiently Finding Candidate Followers

In this section, we define the candidate follower set for anchoring a vertex based on the *edge onion layers*. We prove that the candidate follower set is adequate for computing the followers of an anchor. We first introduce the concept of a *triangle hold path*, and show how to find the candidate followers of an anchor x, denoted by $CF(x)$.

If the vertex x is anchored, every edge in $\mathcal{E}(x, \mathcal{L})$ immediately become a candidate edge for $T_k(G_A)$.

Definition 17. ***Strong Triangle Hold.*** *We say there is a strong triangle hold from an edge e_1 to another edge e_2 if there is a triangle $\{e_1, e_2, e_3\}$ in $\mathcal{L}$ such that $l(e_1) < l(e_2)$ and $l(e_1) < l(e_3)$.*

Note that $l(e)$ is the layer index of the edge $e \in \mathcal{L}$. The triangle hold is called strong because the edge e_2 can immediately be a candidate edge in $T_k(G_x)$, once e_1 became a candidate edge in $T_k(G_x)$.

Definition 18. ***Weak Triangle Hold.*** *We say there is a weak triangle hold from an edge e_1 to another edge e_2 if there is a triangle $\{e_1, e_2, e_3\}$ in $\mathcal{L}$ such that $l(e_1) < l(e_2)$ and $l(e_1) = l(e_3)$.*

The triangle hold is called weak because the edge e_2 become a candidate edge in $T_k(G_x)$ if and only if e_1 and e_3 both became candidate edges in $T_k(G_x)$. In above two definitions, we also say e_1 *triangle holds* e_2 or e_2 is *triangle held* by e_1, if e_2 can become a candidate edge by e_1.

Definition 19. ***Triangle Hold Path.*** *We say there is a triangle hold path from a given anchor vertex x to a vertex $u \in H_{k-1}(G)$ (i.e., $x \rightsquigarrow u$) if there is an edge $e_x \in \mathcal{E}(x, \mathcal{L})$, an edge $e_u \in \mathcal{E}(u, \mathcal{L})$ and an edge set E such that each edge in $E \cup \{e_x, e_u\}$ is triangle held by at least one edge in $E \cup \{e_x, e_u\}$.*

Theorem 21. *A vertex $u \in H_{k-1}(G)$ is a follower of the anchor x implies that there is a triangle hold path $x \rightsquigarrow u$ if $x \neq u$.*

Proof. Let E_x denote the anchor edges of x in $\mathcal{L}$, i.e., $E_x = \mathcal{E}(x, \mathcal{L})$. We can use Line 1 to 6 of Algorithm 15 to compute the anchored k-truss $T_k(G_x)$ as long as (i) for each anchor edge $e_x \in E_x$, we only push e_x into P when $sup(e_x, N) < 1$ in Line 2 and 6; (ii) all isolated vertices are deleted in N. Finally, $T_k(G_x)$ is exactly $N \cup T_k(G)$. In the computation of $T_k(G)$ (without any anchors), all edges in L_1^s are removed in Algorithm 15, while some of the edges may survive the computation

of $T_k(G_x)$ (with anchoring x). The followers of x are the vertices which directly connect an survived edge and do not in $T_k(G)$. Let i denote the minimum layer index of an edge in E_x (i.e, $i = min\{l(e) \mid e \in E_x\}$). In the computation of $T_k(G_x)$, we use the same edge accessing order in computing $T_k(G)$. For an edge $e_1 \in L_1^{i-1}$, when e_1 is accessed, the support of e_1 is less than $k-2$ because the support at current time is exactly the same as e_1 is accessed in the computation of $T_k(G)$. So all edges in L_1^{i-1} are still deleted in computing $T_k(G_x)$. Then, when edges in L_1^{i-1} have been deleted and no edges in L_i have been deleted, for every edge $e_2 \in L_i \setminus E_x$, the support of e_2 is less than $k-2$ because the support at current time is the same as in the computation of $T_k(G)$. Consequently, all edges in $L_1^i \setminus E_x$ cannot be candidate edges and are deleted. At this point, only edges which are triangle held by an edge in E_x become candidate edges and clearly they have triangle hold paths. For a candidate edge e_3, only its triangle held edges in L_{j+1}^s ($j = l(e_3)$) become candidate edges because e_3 cannot save other edges in computing $T_k(G_x)$, i.e., the supports of other edges cannot increase with the existence of e_3 when the edges are accessed. Consequently, the candidate spread from x is strictly a top-down search through E_x's triangle held edges and candidate edges' triangle held edges, which constitute triangle hold paths. For an edge e_4 without any triangle support paths, when non-candidate edges in L_1^{p-1} ($p = l(e_4)$) have been deleted and no edge in L_p has been deleted, the support of e_4 is less than $k-2$ because it is same as e_4 is accessed in computing $T_k(G)$. A follower of x must directly connect to at least one candidate edge in computing $T_k(G_x)$. Consequently, there is always a triangle hold path from x to a follower u if $x \neq u$. □

The candidate follower set $CF(x)$ consists of vertices where each vertex u has at least one triangle hold path from x and $u \in H_{k-1}(G)$. According to the proof of Theorem 21, we can generate $CF(x)$ by finding *candidate edges* (these

edges may survive in $T_k(G_x)$) which is to iteratively activate the triangle held edges by candidate edges which include anchor edges.

Example 18. *In Figure 4.9 with k=4, we label every edge e in L_0^s with its layer index $l(e)$. If the vertex v_2 is anchored, all the edges in $\mathcal{E}(v_2)$ become candidate edges. There is a strong triangle hold from (v_2, v_1) to (v_1, v_3), so the latter becomes a candidate edge. For the edge (v_3, v_6), there is a triangle $\triangle_{v_2,v_3,v_6}$ with two weak triangle holds. Since (v_2, v_3) and (v_2, v_6) are already candidate edges, (v_3, v_6) becomes a candidate edge. No other edges can become candidate edges and we only need to compute the anchored k-truss on the candidate edges and the original k-truss.*

4.4.6 Finding Followers with Early Termination

To find the true followers of an anchor x, we need to conduct k-truss computation on the subgraph induced by $CF(x) \cup T_k(G) \cup \{x\}$. To further speed up the follower finding procedure, we introduce an early termination technique in generating $CF(x)$ to directly retrieve the true followers.

Layer-by-Layer Search. In the search of finding candidate edges, each edge has **three statuses**. We say an edge in $\mathcal{L}$ is **unexplored** if it has not been checked with the support constraint in our layer-by-layer traversal. An edge in $\mathcal{L}$ is **survived** if it has survived the support check, otherwise it becomes **discarded**. For a given anchor, a *discarded* edge will never be involved in the following computation, and a *survived* edge may become *discarded* later due to the deletion cascade. Note that some edges are *implicitly* marked as *discarded* since they are not involved in any triangle hold paths from x, and will never be explored.

For an edge $e \in \mathcal{L}$, the triangles containing e can be divided into six disjoint sets $\triangle_{s,s}$, $\triangle_{s,u}$, $\triangle_{s,k}$, $\triangle_{u,u}$, $\triangle_{u,t}$, $\triangle_{t,t}$ where we use subscripts s, u and t represent

that an edge is *survived*, *unexplored* and in $T_k(G)$, respectively, e.g., $\triangle_{s,u}$ represents the triangle set where each containing a *survived* edge, an *unexplored* edge and e. We use $s^+(e) = |\triangle_{s,s}| + |\triangle_{s,u}| + |\triangle_{s,k}| + |\triangle_{u,u}| + |\triangle_{u,t}| + |\triangle_{t,t}|$ to denote the upper bound of $sup(e, T_k(G_x))$. The following theorem indicates that we can safely exclude a candidate edge e if its support upper bound is insufficient. The removal of an edge may invoke the deletion of other edges, where details are described in Algorithm 16. When the shrink function terminates, all of the edge supports and edges affected by the removal of e will be correctly updated.

Theorem 22. *For an edge $e \in \mathcal{L}$, e cannot exist in $T_k(G_x)$ if one of the following conditions satisfied: (i) $e \in \mathcal{E}(x)$ and $s^+(e) < 1$; (ii) $e \notin \mathcal{E}(x)$ and $s^+(e) < k-2$.*

Proof. Since the edges in $G \setminus \{\mathcal{L} \cup T_k(G)\}$ cannot exist in $T_k(G_x)$, we only need to consider triangles in $\mathcal{L} \cup T_k(G)$ to compute the support upper bound of the edge e. Since *discarded* edges cannot provide triangle hold to any edge, the six disjoint triangle sets are adequate for computing $sup(e, T_k(G_x))$. Consequently, $s^+(e)$ is a correct upper bound of $sup(e, T_k(G_x))$. □

Example 19. *In Figure 4.9 with k=4, if the vertex v_2 is anchored, all the edges in $\mathcal{E}(v_2)$ become candidate edges and is marked unexplored. Without the support check, (v_1, v_3) can become a candidate edge. However, according to Theorem 22, (v_1, v_3) cannot become a candidate edge and is marked discarded because $s^+((v_1, v_3)) = 1 < 2$. The same for (v_3, v_6). Note that some edges are implicitly marked discarded such as (v_1, v_4). Thus no edges outside $T_k(G)$ can survive in $T_k(G_{v_2})$.*

Finding Followers. Algorithm 17 lists the pseudo-code of the follower computation for a chosen anchor x. A min heap H is used to keep the candidate edges, and the key of an edge e is $l(e)$ with ties broken by the edges' IDs. In this way, we explore the candidate edges in a layer-by-layer fashion and it is easy to

Algorithm 16: **ShrinkEdge**(e)

Input : e : the edge for support check
for each *survived* edge e_0 which forms a triangle with e and a *non-discarded* edge **do**
  $s^+(e_0) := s^+(e_0) - 1$;
  $T \leftarrow v$ **If** $s^+(e_0) < 1$ and $e_0 \in \mathcal{E}(x)$;
  $T \leftarrow v$ **If** $s^+(e_0) < k-2$ and $e_0 \notin \mathcal{E}(x)$;
for each $e_1 \in T$ **do**
  e_1 is set *discarded*;
  ShrinkEdge(e_1);

Algorithm 17: **FindFollowersTruss**(x, $\mathcal{L}$)

Input : x : the anchor; $\mathcal{L}$: *edge onion layers*
Output: F : the followers of x
$H := \emptyset$; $F := \emptyset$ //H is a min heap, key is layer index;
for each $e \in \mathcal{E}(x, \mathcal{L})$ **do**
  $H.push(e)$;
while $H \neq \emptyset$ **do**
  $e_0 \leftarrow H.pop()$;
  Compute $s^+(e_0)$;
  if $s^+(e_0) \geq k-2$ or ($s^+(e_0) \geq 1$ and $e \in \mathcal{E}(x, \mathcal{L})$) **then**
    e_0 is set *survived*;
    for each e_1 *triangle held* by e_0 and $e_1 \notin H$ **do**
      $H.push(e_1)$;
  else
    e_0 is set *discarded* ;
    ShrinkEdge(e_0);
for each *survived* edge e **do**
  $F := F \cup \mathcal{V}(e)$;
return $F \setminus T_k(G)$

check whether an edge e has been *explored* based on its ID and layer index $l(e)$. For each popped edge e, Line 6 computes its support upper bound $s^+(u)$. If e survives the support check, e will be set to *survived* (Line 6) and the edges *triangle held* by e in lower layers (i.e., unexplored candidate edges) will be pushed

into H if they are not already in H (Lines 9-10). Otherwise, e is set to *discarded* and the early termination process is invoked (Lines 12-13). The deletion may be cascaded and some *survived* edges may be set to *discarded* during the process. When the algorithm terminates, the vertices which directly connect at least one *survived* edge are the followers of x.

The time complexity of the algorithm is $\mathcal{O}(|\triangle|)$ in the worst case because each triangle is at most accessed six times: to push candidate edges into H, compute upper bounds and compute the cascades of the deletion.

Algorithm Correctness. We show the deletion of the edges in Algorithm 17 has a *valid order* O for the computation of $T_k(G_x)$. An edge e may be *implicitly* deleted when e never become a candidate edge, i.e., when there is no early termination in the finding followers, (1) no triangle hold paths exist for e; or (2) some edges on e's triangle hold paths (without early termination) are *discarded*, thus no triangle hold paths exist when applying early termination.

According to Theorem 21, we can safely discard such edge e. An edge e may also be *explicitly* deleted in O if (3) u is set *discarded* at Line 12 because it fails the support check when it is popped; or (4) $s^+(u)$ becomes insufficient due to the deletions of the other edges (Line 6 of Algorithm 16). Because $s^+(u)$ is correctly computed (Line 6) and maintained (Algorithm 16), e does not satisfy the support constraint when e is deleted in cases (3) and (4).

Let M denote the remaining edges vertices when Algorithm 17 terminates. As all of the edge in $\mathcal{L}$ have been *explored* explicitly or implicitly, we have $s^+(u) = sup(e, M \cup T_k(G)) \geq k-2$, for every vertex $e \in M$. Consequently, none of the edges in $M \cup T_k(G)$ can be discarded. As such, O is a *valid order* and $T_k(G_x) = \mathcal{V}(M) \cup M \cup T_k(G)$.

4.4.7 The AKT Algorithm

The follower based pruning technique (Theorem 14) can also be applied in the anchored k-truss problem, which indicates that, in the procedure of finding the best anchor, we do not need to compute the followers of a vertex u if it is a follower of another vertex x; that is, the followers of u is always less than the followers of x if $u \in \mathcal{F}(x)$.

Algorithm 18 illustrates the details of `AKT` which finds the best anchor vertex for a given graph G (i.e., $b = 1$). Particularly, we firstly apply Algorithm 15 to compute the *edge onion layers* of G (Line 1). Initially, the candidate anchor set T is set to $\mathcal{L}$ according to Theorem 20. Then we sequentially access vertices in T based on their degrees in $T_{k-1}(G)$ in decreasing order, and compute their followers by Algorithm 17. According to the follower-based pruning, Line 6 excludes the followers of current accessed vertex from T. We continuously maintain the current best anchor with the largest number of followers (denoted by λ) seen so far. We have the best anchor when the algorithm terminates.

To handle general cases where $b > 1$, our `AKT` algorithm can easily fit within the greedy algorithm (Replacing Lines 3-4 of Algorithm 14) to find the best vertex in each iteration. The only difference is that we need to enforce that the support constraint for each anchor edge in previous iterations to be only 1 and $\mathcal{L}$ is computed from anchored (k-1)-truss. Note that in order to avoid computing $T_{k-1}(G_A)$ (Line 1 of Algorithm 15) from scratch in each iteration, we firstly maintain $C_{k-2}(G_A)$ since it should be a supergraph of $T_{k-1}(G_A)$. Moreover, if the (k-1)-truss consists of a set of disconnected subgraphs, we can avoid the re-computation of the followers of a subgraph in the next iteration unless there is a new anchor in this subgraph.

Algorithm 18: **AKT**(G, k)

Input : G : a social network, k : support constraint
Output: the best anchor vertex

```
1 Compute edge onion layers L (Algorithm 15);
2 T ← L (Theorem 20); λ := 0;
3 for each vertex x ∈ T with decreasing order of deg(x, T_{k−1}(G)) do
4     F(x) ← FindFollowersTruss(x, L) (Algorithm 17);
5     if F(x) ≠ ∅ then
6         T := T \ F(x) (Theorem 14);
7     if |F(x)| > λ then
8         λ := |F(x)|;
9 return the best anchor
```

The time complexity of the algorithm remains $\mathcal{O}(bn|\triangle|)$ in the worst case. Nevertheless, our empirical study shows we can significantly improve performance of the straightforward implementation (Algorithm 14) by orders of magnitude, due to a much smaller number of candidate anchors and a more efficient follower computation algorithm.

Algorithm Correctness. (1) For the anchored k-truss problem with $b = 1$ on graph G, we get the correct result immediately based on the correctness of proposed techniques. (2) Assume the algorithm is correct when $b = i$, $i \in N^+$ and returns the anchor set A. (3) Consider the problem with $b = i + 1$, now the k-truss of G is $T_k(G_A)$ since we have $sup(e, G_A)$ satisfies support constraint for any $e \in T_k(G_A)$ and $T_k(G_A)$ is maximal. Then the (k-1)-truss is updated correctly by maintaining the $C_{k-2}(G_A)$. Thus, we get the updated *edge onion layers* $\mathcal{L}$ correctly by Algorithm 15. Since all the techniques are based on $\mathcal{L}$, after running `AKT` on G with $b = 1$ again, we get the correct result $A \cup \{x\}$ for the case of $b = i + 1$ on G. Note that in the (k-1)-truss N of G, for every disconnected subgraph S with $S \in N$ and $x \notin S$, S keeps same after anchoring x, thus, the previous result of anchoring any vertex in S can be reused.

Table 4.3: Statistics of Datasets

Dataset	**Nodes**	**Edges**	d_{avg}	d_{max}
Facebook	4,039	88,234	43.7	1045
Brightkite	58,228	194,090	6.7	1098
Gowalla	196,591	456,830	4.7	9967
Yelp	552,339	1,781,908	6.5	3812
Flickr	105,938	2,316,948	43.7	5465
YouTube	1,134,890	2,987,624	5.3	28754
DBLP	1,566,919	6,461,300	8.3	2023
Pokec	1,632,803	8,320,605	10.2	7266
LiveJournal	3,997,962	34,681,189	17.4	14815
Orkut	3,072,441	117,185,083	76.3	33313

4.5 Performance Evaluation

This section evaluates the effectiveness and efficiency of all techniques through comprehensive experiments. We firstly present the experimental setting and results on algorithms for the anchored k-core and then the anchored k-truss.

4.5.1 Experimental Setting for Anchored k-Core

Datasets. Ten real-life networks were deployed in our experiments and we assume all vertices in each network are initially engaged. The original data of Yelp was downloaded from https://www.yelp.com.au/dataset_challenge, DBLP came from http://dblp.uni-trier.de/ and the others were from http://snap.stanford.edu/. Table 4.3 shows the statistics of the 10 datasets, listed in increasing order of their edge numbers.

Algorithms. To the best of our knowledge, no existing work investigates efficient algorithms for the anchored k-core problem on general graphs. Towards the effectiveness, we tested five algorithms (Rand, Rand1, Rand2, Degree and Exact) to choose different anchors to see the number of followers, compared with our

Table 4.4: Summary of Algorithms for Anchored k-Core

Algorithm	**Description**
`Rand`	randomly chooses b anchors from $G \setminus C_k(G)$
`Rand1`	randomly chooses b anchors from N_1 where $N_1 = \mathcal{L} \cap NB(C_k(G))$
`Rand2`	randomly chooses b anchors from N_2 where $N_2 = \mathcal{L} \cap NB(N_1)$
`Degree`	chooses the b anchors from N_2 with highest degrees in $\mathcal{L} \setminus L_0$
`Exact`	identifies the optimal solution by exhaustively searching all possible combinations of b anchors by Algorithm 11
`Naive`	computes a k-core on G for each candidate anchor $u \in G \setminus C_k(G)$ to find the best anchor in each iteration of Algorithm 8
`Baseline1`	computes a k-core on G for each candidate anchor $u \in \mathcal{L}$ (Theorem 10) to find the best anchor in each iteration of Algorithm 8
`Baseline2`	applies the state-of-art core maintenance algorithm [99] to compute followers for each candidate anchor in `Baseline1`
`BL1+C`	computes a k-core on $\{x\} \cup C_{k-1}(G)$ (Theorem 9) for each candidate anchor x in `Baseline1`
`BL1+CF`	computes a k-core on $CF(x) \cup \{x\} \cup C_k(G)$ (Theorem 12) for each candidate anchor x in `Baseline1`
`BL1+CFE`	finds followers from $CF(x)$ by early termination technique (Theorem 13) for each candidate anchor x in `Baseline1`, i.e., applies Algorithm 11
`BL1+CFEP`	equips the follower based pruning (Theorem 14) in `BL1+CFE`
`OLAK`	equips the upper bound based pruning (Theorem 15) in `BL1+CFEP` and arrives at Algorithm 12

greedy result (`OLAK`). Case studies were made on the greedy result. We also implemented and evaluated the algorithms to assess our techniques incrementally, from a naive algorithm (`Naive`) through to the final advanced algorithm (`OLAK`). One baseline algorithm (`Baseline2`) based on core maintenance is also evaluated. Table 4.4 shows the summary for algorithms.

Parameters. We conducted experiments under different settings by varying the degree constraint k and the budget for the anchors b. The default values of k

and b were both 20. In the experiments, the range of k varied from 5 to 50 and the range of b varied from 1 to 100.

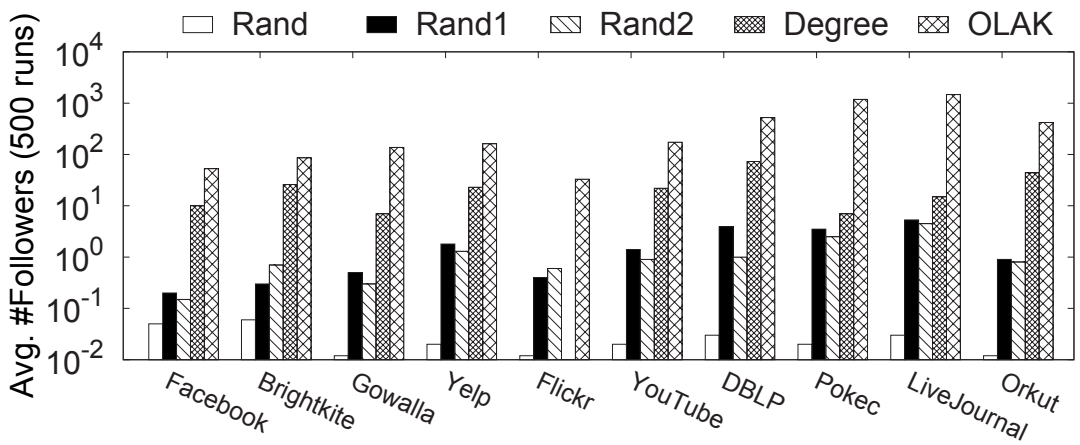
For both anchored k-core and anchored k-truss, all programs were implemented in standard C++ and compiled with G++ in Linux. All experiments were performed on a machine with Intel Xeon 2.3GHz CPU and Redhat Linux System. We evaluate the effectiveness of the algorithms by reporting the number of the followers for the resulting anchors. The efficiency of the algorithms is measured by its running time and the number of the candidate anchors and followers accessed.

4.5.2 Effectiveness of OLAK

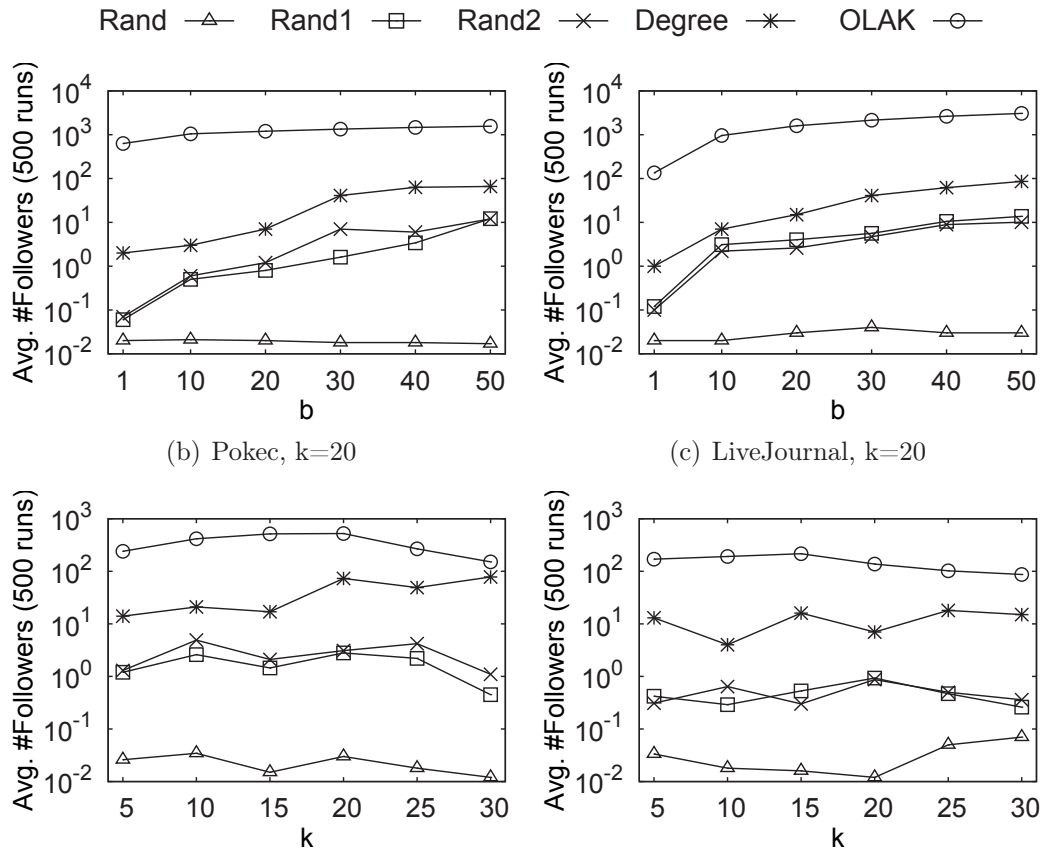
We used the result of `OLAK` to evaluate the effectiveness for all greedy algorithms, since they follow the same heuristic and produce the same results. We also conducted case studies to show real-world examples for the anchored k-core.

Effectiveness of the Greedy Algorithm

Figure 4.10 compares the number of followers w.r.t b anchors identified by `OLAK` with three random approaches (`Rand`, `Rand1` and `Rand2`) and one degree based approach (`Degree`).We report the average number of followers for 500 independent tests in three random methods. The resulting numbers for other two methods are always unique because we choose the highest degree anchors in `Degree` and find the best anchors in `OLAK`. Note that $\mathcal{L} \setminus L_0$ is the set which contains all followers for any anchor, and $\mathcal{L}$ contains all promising anchors. `Degree` basically improves performance by choosing high degree anchors, but is still significantly outperformed by `OLAK`. In Figure 4.10(a), `Degree` fails to get any follower in `Flickr` because a high degree vertex u in N_2 does not necessarily have a follower. It is common to find no followers after 20 random anchors are chosen.



(a) Number of Followers on Different Datasets



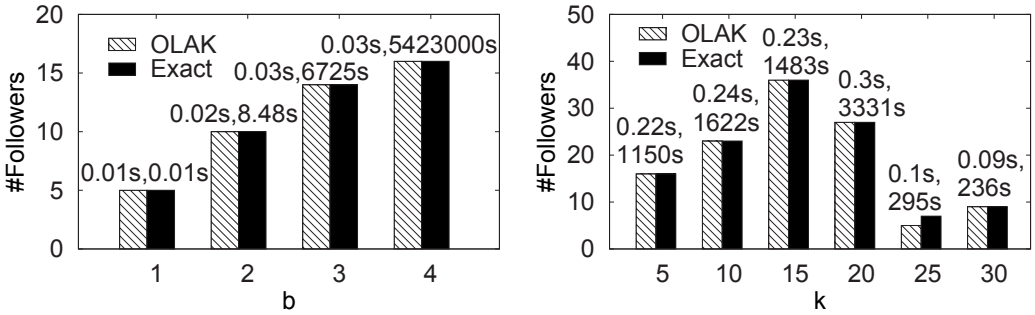
(b) Pokec, k=20

(c) LiveJournal, k=20

(d) DBLP, b=20

(e) Gowalla, b=20

Figure 4.10: Number of Followers



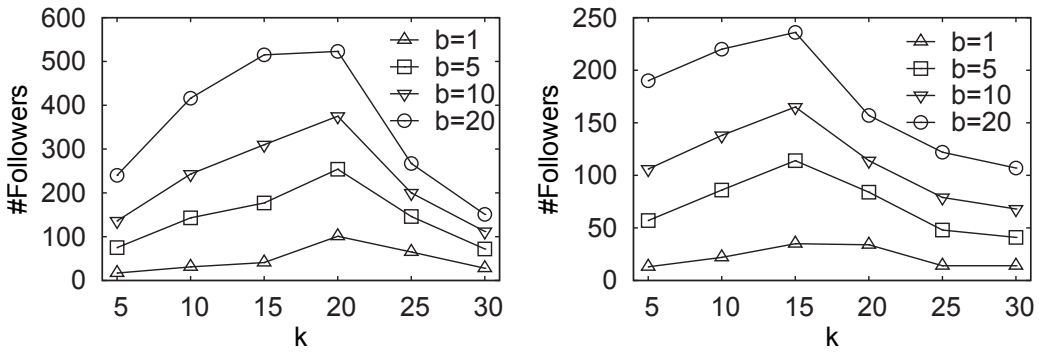
(a) Facebook, k=30 (b) Brightkite, b=2

Figure 4.11: Greedy vs Exact

This is because we observe that, the majority of the vertices do not have any followers in all 10 real-life network, which justifies the necessity for an efficient anchored k-core algorithm to find the critical vertices. We also show that our greedy algorithm achieves much better performance. For `OLAK`, we notice that there are 630 followers in `Pokec` dataset with a single anchored vertex. Figure 4.10(a) shows that `OLAK` underpins more than 1000 followers with 20 anchors on the `Livejournal` and `Pokec`. Figures 4.10(b)-(e) show the margin between `OLAK` and the other algorithms does not change much when k and b vary.

To further justify the effectiveness of `OLAK`, we also compare its performance with `Exact`, which identifies the optimal solution by exhaustively searching two relatively small networks, where b varies from 1 to 4 on `Facebook` and k varies from 5 to 30 on `Brightkite`. Figure 5.5 shows that `OLAK` finds the optimal solution in all but one setting. Note that we only test `Exact` on small datasets with small b values because we cannot afford its running time for other settings.

Figure 4.12 reports the impact of b and k on the size of the followers for `OLAK`. The number of the followers clearly grows with the increase of the budget b. The size becomes relatively small when k is small or large. This is because the majority of the vertices are already in k-core when k is small, while it is difficult to provide enough degree support when k is large.



(a) DBLP (b) Gowalla

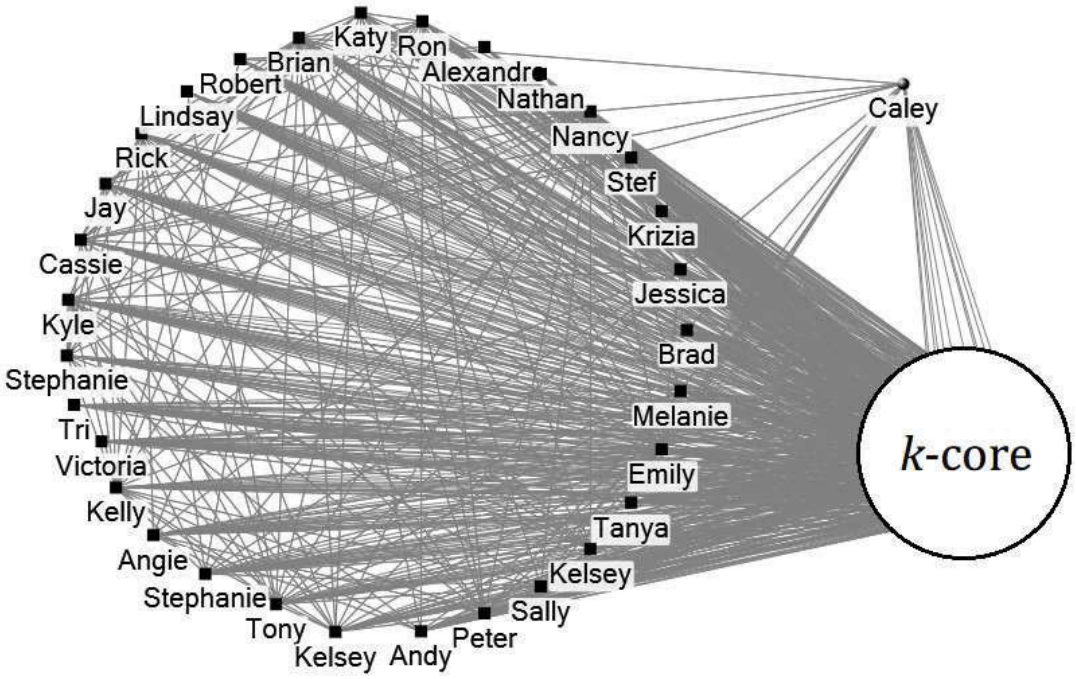
Figure 4.12: Effect of k and b

Case Studies

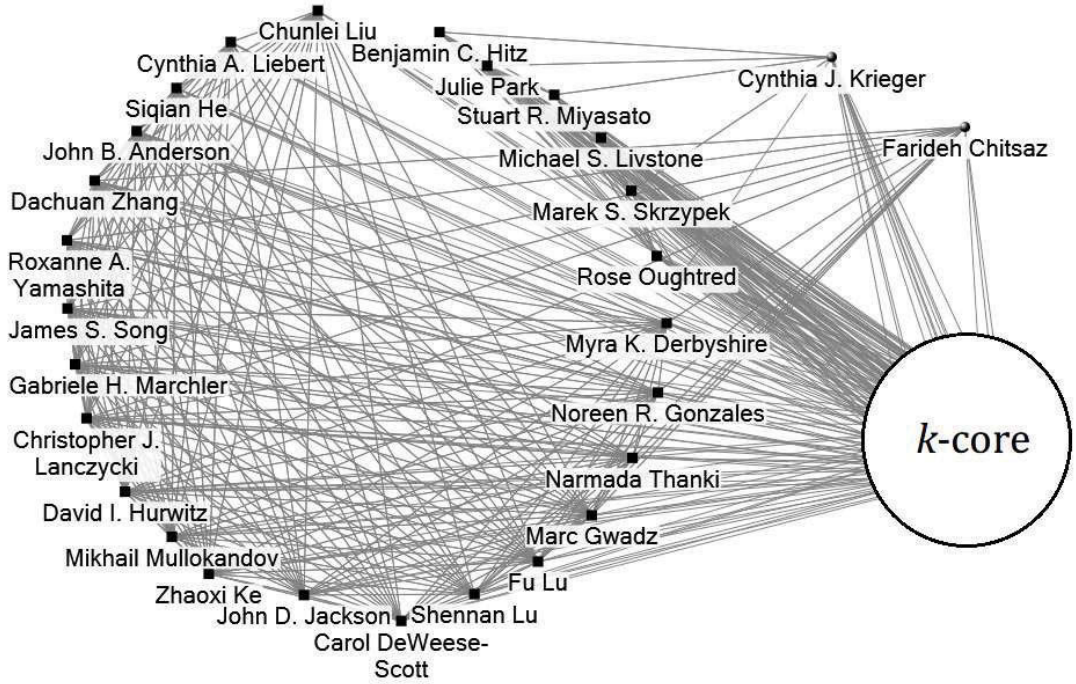
We show the anchors identified by `OLAK` and their corresponding followers in Figures 4.13(a)-(b). Figure 4.13(a) shows that when the user "Caley" alone is anchored, there are 31 followers in `Yelp` with $k = 30$. It is interesting that only 10 of them are neighbors of "Caley", and the others are supported indirectly.

In Figure 4.13(b), `DBLP` is deployed and k is set to 20. In the case study, there is an edge between two authors if they co-author at least three papers. When $b = 2$, two authors are identified by `OLAK` and there are 26 followers. We find that although the two anchored authors have not co-authored any papers, they belong to the same community and 8 out of their 9 papers in `DBLP` have been published in Nucleic Acids Research. Not surprisingly, all their followers are also from the same community, and there are already considerably large number of co-authored papers among them.

We also investigated the characteristics of the anchors identified by `OLAK` in different settings and datasets. It is non-trivial to understand the potential of a vertex based on its local structure information (e.g., degree or neighbor's degrees), due to the complicated cascade behavior of unraveling in the networks, but it does justify the importance of `OLAK`.

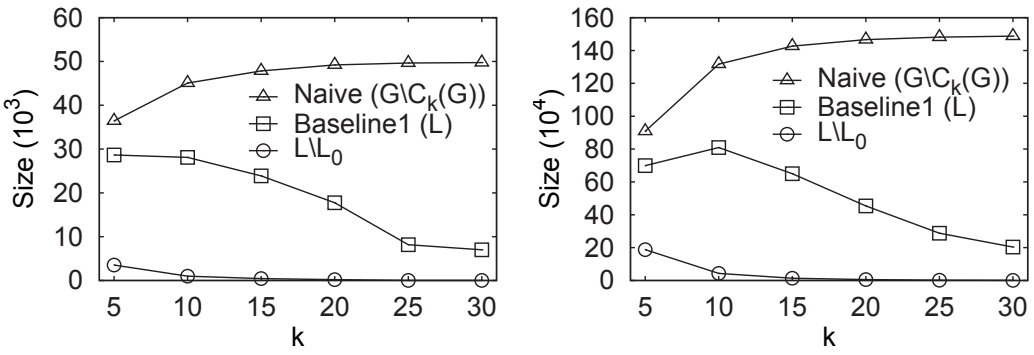


(a) Yelp, k=30, b=1



(b) DBLP, k=20, b=2

Figure 4.13: Case Studies



(a) Brightkite, b=1 (b) DBLP, b=1

Figure 4.14: Reducing Candidate Anchors

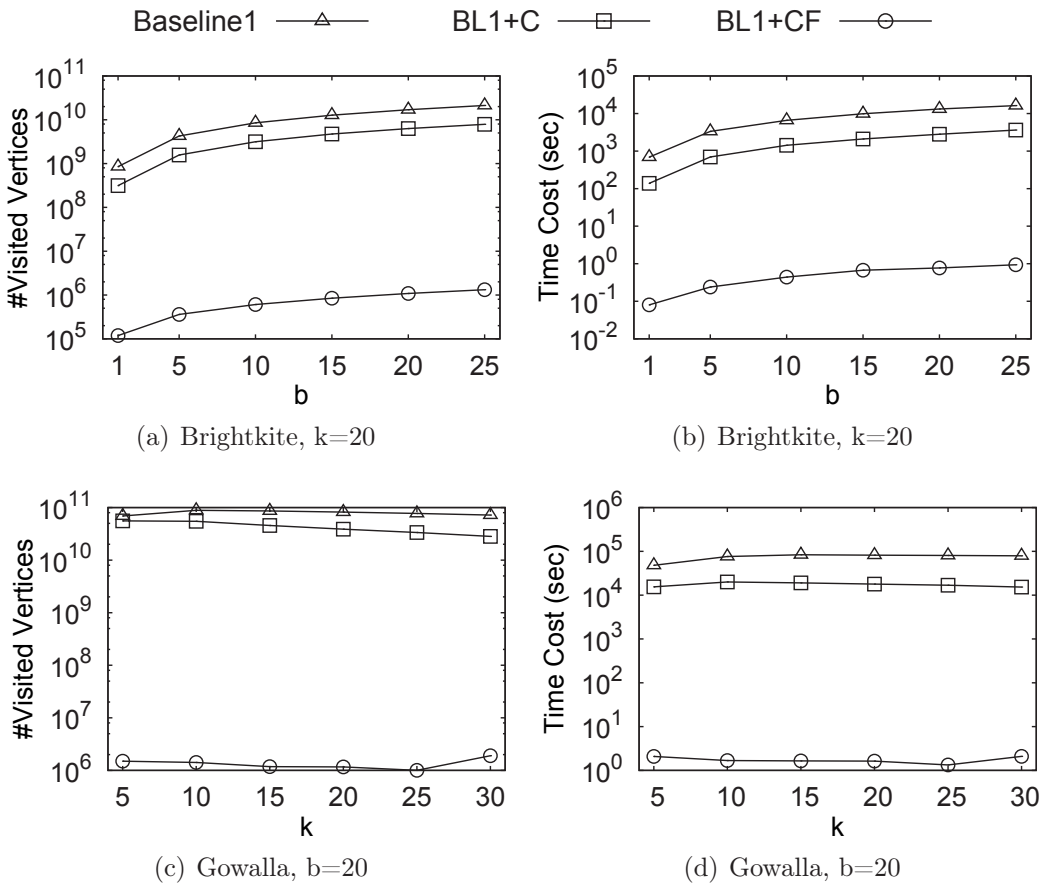
4.5.3 Efficiency of OLAK

We first investigate the efficiency of the techniques, then compare our `OLAK` algorithm with `Baseline1` and `Baseline2` under different settings.

Evaluation of Individual Techniques

The essence of our proposed techniques is to use the *onion layer* structure to speed up the anchored k-core computation by significantly reducing the number of candidate anchors and candidate followers. Below, we evaluate the proposed techniques against these two criteria.

Reducing Candidate Anchors. Figure 4.14 reports the sizes of $G \setminus C_k(G)$, *onion layers* $\mathcal{L}$ and (k-1)-shell (i.e.,$\mathcal{L} \setminus L_0$) on two networks `Brightkite` and `DBLP` with $b = 1$ and k varied from 5 to 30. Recall that `Naive` checks all vertices in $G \setminus C_k$, and the other three algorithms (`Baseline1`, `Baseline2` and `OLAK`) only consider the vertices from $\mathcal{L}$ as candidate anchors (Theorem 10). We also report the size of (k-1)-shell ($\mathcal{L} \setminus L_0$), which bounds the size of the candidate followers by Theorem 9. As expected, the size of $G \setminus C_k(G)$ grows with k because the size of k-core decreases with k. Conversely, the size of (k-1)-shell is much smaller,



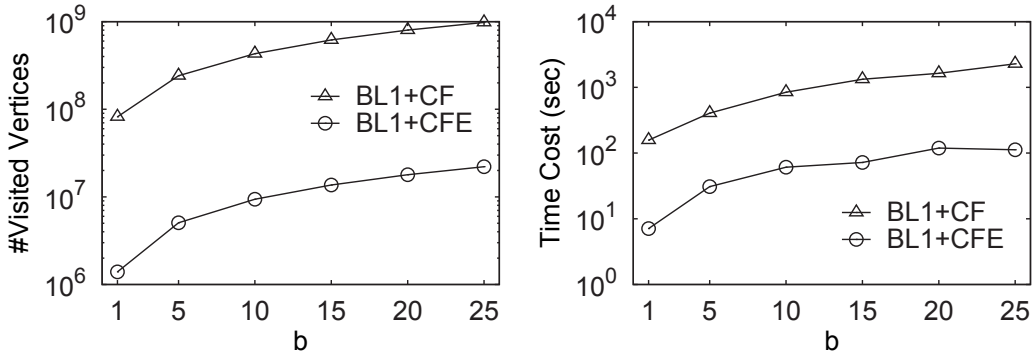
(a) Brightkite, k=20 (b) Brightkite, k=20

(c) Gowalla, b=20 (d) Gowalla, b=20

Figure 4.15: Pruning Candidate Followers

and drops with the growth of k. The size of *onion layers* $\mathcal{L}$ also decreases quickly with k. It also shows that the majority of the vertices in $\mathcal{L}$ are neighbors of (k-1)-shell vertices, especially for small k, which is not considered in the computation of the followers.

Pruning Candidate Followers. Figure 4.15 demonstrates the effectiveness of the pruning techniques which help us to eliminate non-promising candidate followers. Three algorithms were evaluated using the number of visited followers and the running time on two networks `Brightkite` and `Gowalla` by varying b and k, respectively. In `Baseline1`, all vertices in G are regarded as candidate



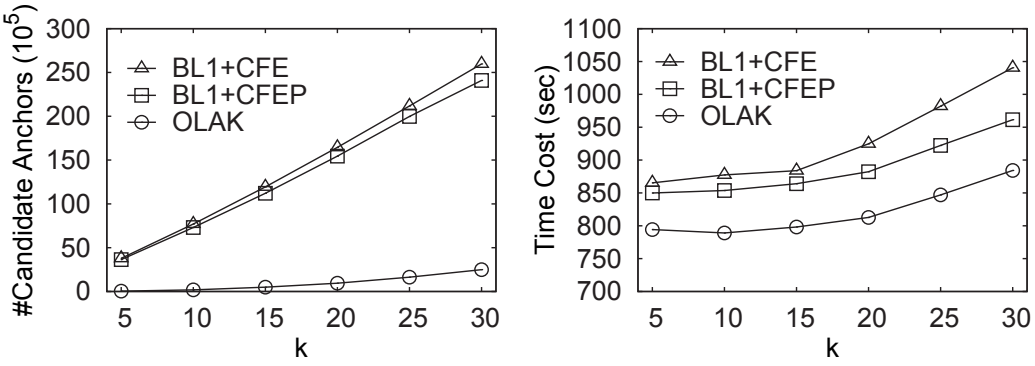
(a) Pokec, k=20 (b) Pokec, k=20

Figure 4.16: Effectiveness of Early Termination

followers during the k-core computation. `BL1+C` represents `Baseline1` equipped with *onion layers*, where candidate followers are obtained from the (k-1)-shell. (Theorem 9), and `BL1+CF` is `Baseline1` equipped with the candidate exploration technique, which only explores the vertices in $CF(x)$ for each candidate anchor x (Theorem 12). We report that both pruning techniques significantly reduce the number of the candidate followers explored, especially the support path based candidate follower exploration (Theorem 12).

Early Termination. Figure 4.16 shows that our early termination technique (Theorem 13) , applied in `BL1+CFE`, can further significantly reduce the number of explored vertices during computation, and hence improves performance by at least one order of magnitude.

Pruning Candidate Anchors. Figure 4.17 evaluates the effectiveness of the follower based and upper-bound based candidate anchor pruning techniques on the `Orkut` dataset with k ranging from 5 to 30. In particular, `BL1+CFE` is the `OLAK` algorithm without these two pruning techniques. `BL1+CFEP` includes the follower-based pruning (Theorem 14), and `OLAK` further equips the algorithm with upper-bound based pruning (Theorem 15). We report that both pruning techniques contribute to the performance of `OLAK`. One interesting observation



(a) Orkut, b=20 (b) Orkut, b=20

Figure 4.17: Further Pruning Candidate Anchors

is that although upper-bound based pruning eliminates many more candidate anchors than follower-based pruning, but their contributions in terms of running time do not have a big difference. This is because the majority of the candidate anchors pruned by their upper-bound are immediately excluded by our follower computation algorithm (Algorithm 11).

We observe that the most powerful optimization is pruning candidate followers by onion layers, followed by early termination, reducing candidate anchors and two pruning rules for candidate anchors.

Performance Evaluation

We evaluate the performance of `Baseline1`, `Baseline2` and `OLAK` in different settings.

Different Datasets. Figure 4.18 reports the performance of the three algorithms on 10 networks with $k = 20$ and $b = 20$. The datasets are ordered by their network sizes (i.e., the number of edges). Not surprisingly, the performance of `Baseline1` is very poor and cannot finish computation on 8 networks within one week. Its performance is significantly enhanced by applying the state-of-the-art core maintenance algorithm for the follower computation on *onion layers*.

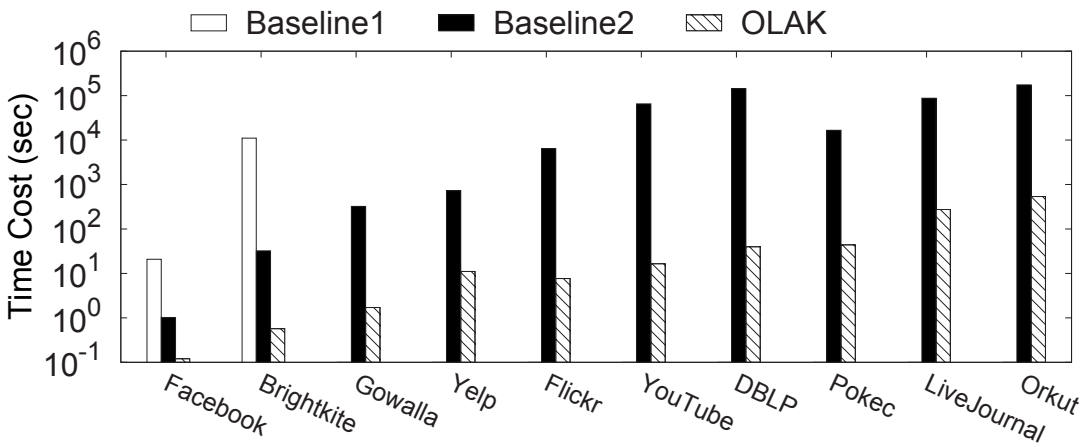


Figure 4.18: Running Time on Different Datasets

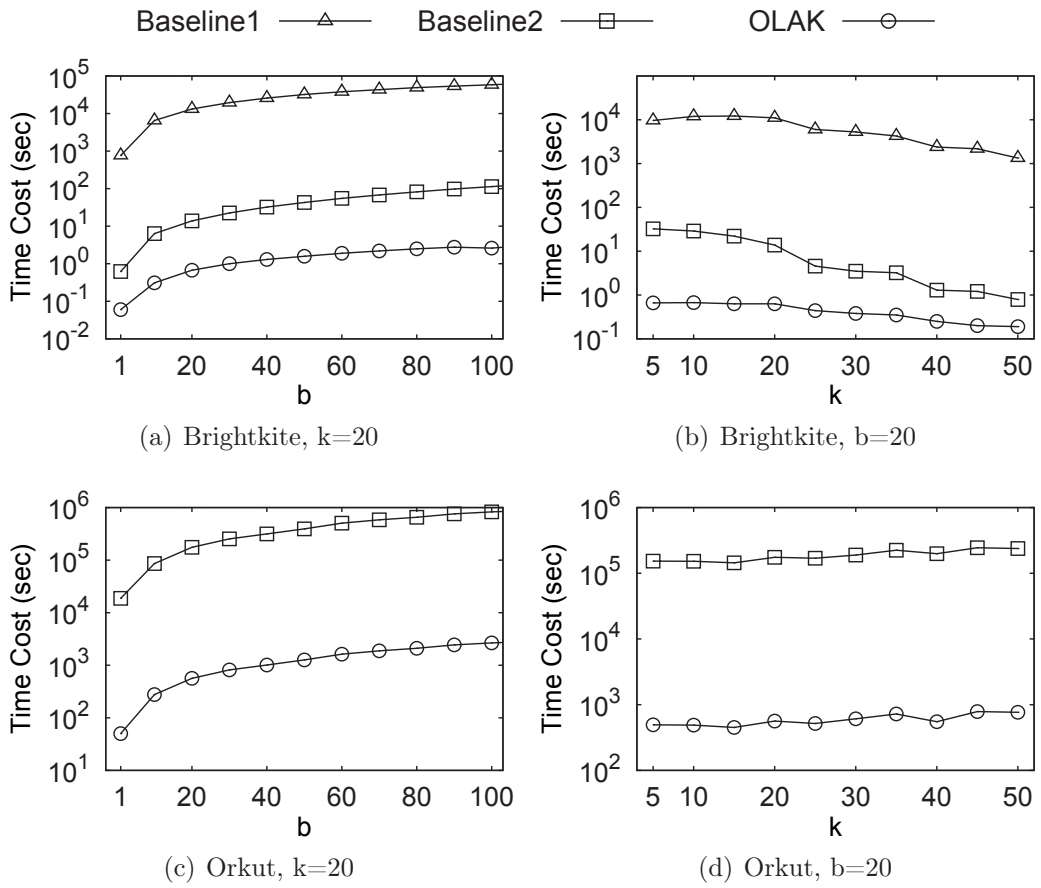
`OLAK` outperforms `Baseline2` by a large margin with up to 3 orders of magnitude.

We observed that the size of the *onion layers* has great impact on the running time of `OLAK`, which is closely related to the network size. This is because the main computation is conducted on the vertices in the *onion layers*. Other factors such as avg. degree, max. degree and the number of *onion layers* in datasets do not make noticeable differences.

Effect of k and b. Figure 4.19 studies the impact of k and b on the three algorithms against two datasets `Brightkite` and `Orkut`, with b varied from 1 to 100 and k ranged from 5 to 50. `OLAK` significantly outperforms the two baseline algorithms under all settings. We omit `Baseline1` for `Orkut` as it cannot finish the computation within one month.

4.5.4 Experimental Setting for Anchored k-Truss

Algorithms. To the best of our knowledge, no existing work investigates the anchored k-truss problem. Towards the effectiveness, we tested four algorithms (`RandT`, `RandT+`, `Triangle` and `ExactT`) to choose different anchors to see the number of followers, compared with our greedy result (`AKT`). Case studies were



(a) Brightkite, k=20 (b) Brightkite, b=20

(c) Orkut, k=20 (d) Orkut, b=20

Figure 4.19: Effect of k and b

made on the greedy result. We also implemented and evaluated the algorithms to assess our techniques incrementally, from a naive algorithm (`NaiveT`) through to the final advanced algorithm (`AKT`). One baseline algorithm (`BaselineM`) based on truss maintenance is also evaluated. Table 4.5 shows the summary for algorithms.

Datasets. In the experiments for the anchored k-truss, we deployed the same ten real-life networks in Table 4.3.

Parameters. We conducted experiments under different settings by varying the support constraint k and the budget for the anchors b. According to the

Table 4.5: Summary of Algorithms for Anchored k-Truss

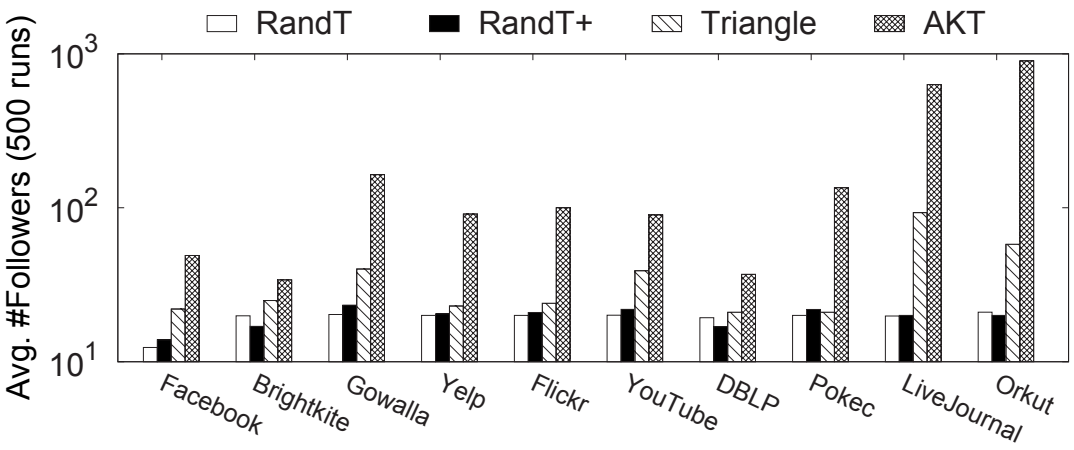
Algorithm	**Description**
`RandT`	randomly chooses b anchors from G
`RandT+`	randomly chooses b anchors from $\mathcal{L}$
`Triangle`	chooses the first b vertices from $\mathcal{L}$ as the anchors with the decreasing order of triangle numbers in $\mathcal{L}$ containing each vertex
`ExactT`	identifies the optimal solution by exhaustively searching all possible combinations of b anchors by Algorithm 17
`NaiveT`	computes a k-truss on G for each candidate anchor $u \in G$ to find the best anchor in each iteration of Algorithm 14
`BaselineT`	computes a k-truss on G for each candidate anchor x in $\mathcal{L}$ (Theorem 20)
`BaselineM`	applies the state-of-the-art truss maintenance algorithm [102] to compute followers for each candidate anchor in `BaselineT`
`BLT+C`	computes a k-truss on $\{x\} \cup T_{k-1}(G)$ (Theorem 19) for each candidate anchor x in `BaselineT`
`AKT`	finds followers from $CF(x)$ through layer-by-layer search on $\mathcal{L}$ (Theorem 21 and 22) for each candidate anchor x in `BaselineT`, i.e., arrives at Algorithm 18

characteristics of different datasets, the default values of k were 40 for Orkut and 15 for other datasets, respectively. The default value of b was 20.

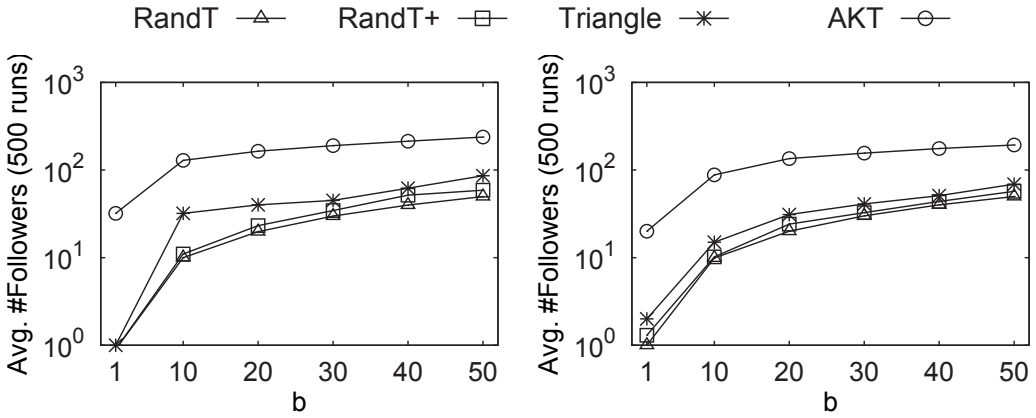
The other experimental setting is same with the anchored k-core experiments. The running time is set as INF if it exceeds 10^5 seconds.

4.5.5 Effectiveness of AKT

Figure 4.20 compares the number of followers w.r.t b anchors identified by `AKT` with two random approaches (`RandT`, `RandT+`) and one triangle number based approach (`Triangle`). We report the average number of followers for 500 independent tests in two random methods. The resulting numbers for other two methods are always unique because we choose the anchors with the highest triangle numbers in $\mathcal{L}$ and find the best anchors in `AKT`. In Figure 4.20(a), `Triangle` basically outperforms `RandT` and `RandT+`, but is still significantly outperformed

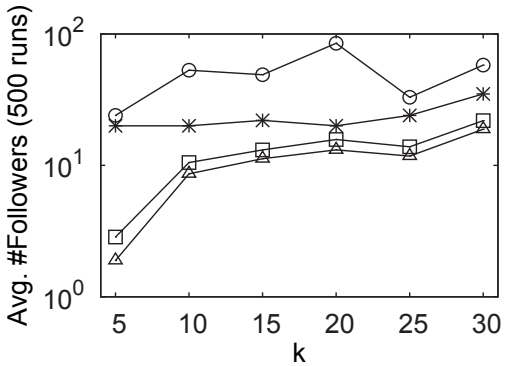


(a) Number of Followers on Different Datasets

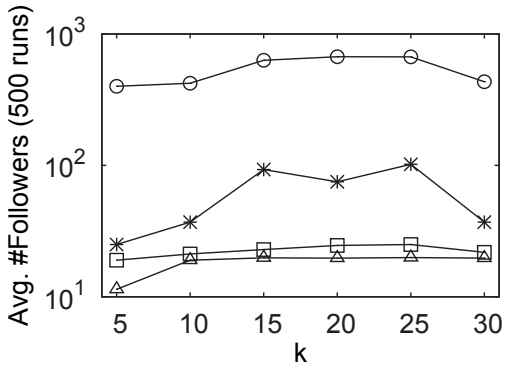


(b) Gowalla, k=15

(c) Pokec, k=15

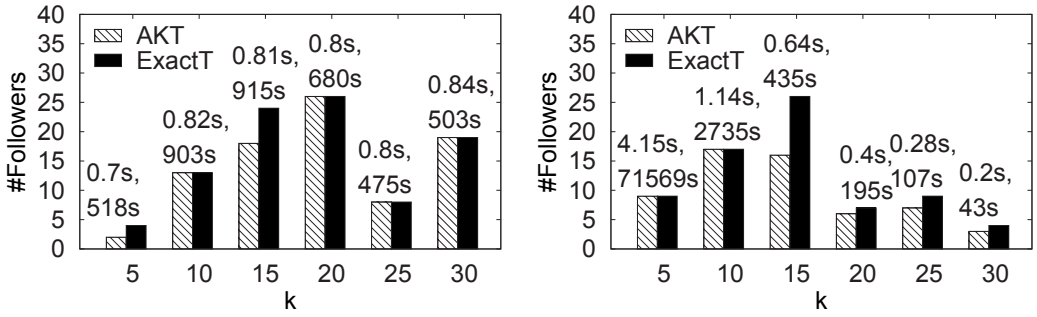


(d) Facebook, b=20



(e) LiveJournal, b=20

Figure 4.20: Number of Followers



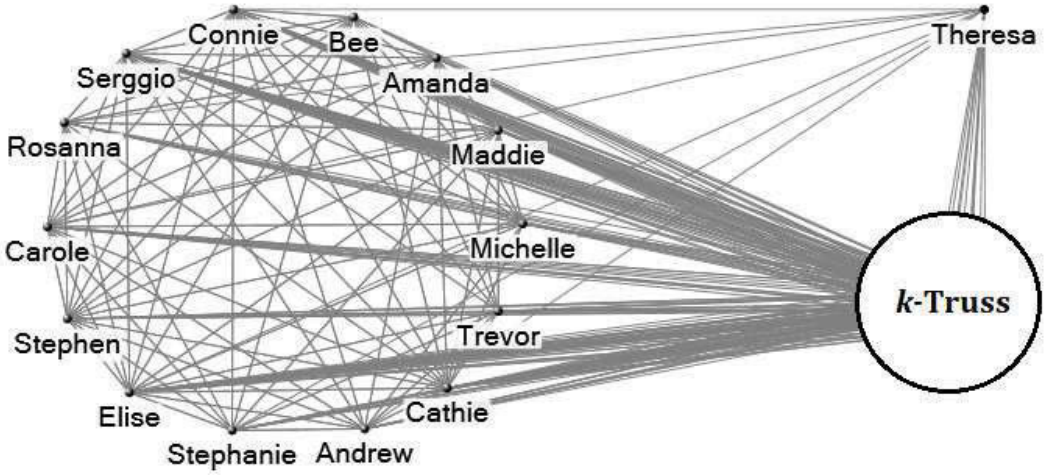
(a) Facebook, b=2 (b) Brightkite, b=2

Figure 4.21: Greedy vs Exact

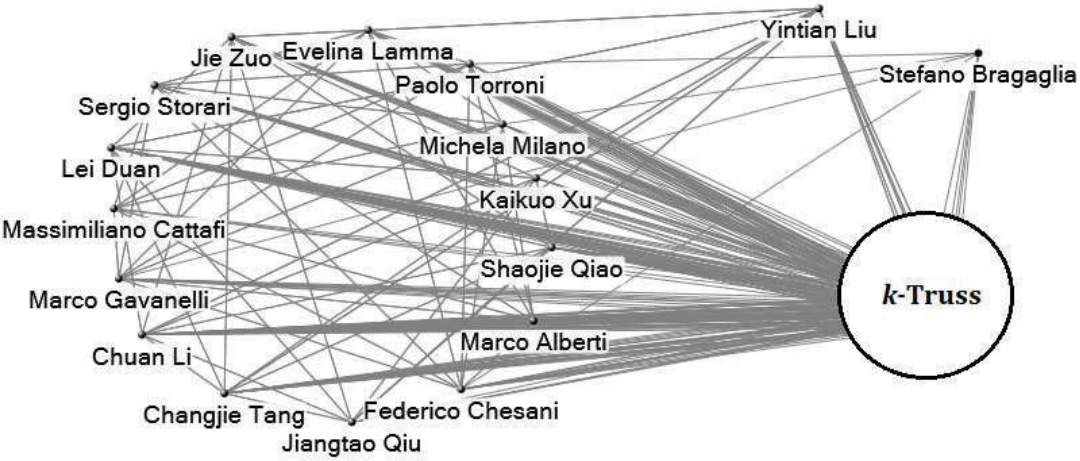
by `AKT`. `Triangle` is beaten by the random methods in `Pokec` because a vertex involved in large number of triangles does not necessarily have a large number of followers. Figures 4.20(b)-(e) show the margin between `AKT` and the other algorithms does not change much when k and b vary.

We also compare the performance of `AKT` with `ExactT`, which identifies the optimal solution by exhaustively searching two relatively small datasets, where k varies from 5 to 30. Figure 4.21 shows that the numbers of followers from `AKT` are comparable with `ExactT`. Note that we only test `ExactT` on small datasets with small b values because we cannot afford its running time for other settings.

We show the anchors identified by `AKT` and their corresponding followers in Figures 4.22(a)-(b). Figure 4.22(a) shows that when the user "Theresa" alone is anchored, there are 15 followers (including "Theresa") in `Yelp` with $k = 10$. It is interesting that only 7 of them are neighbors of "Theresa", and the others are supported indirectly. In Figure 4.22(b), `DBLP` is deployed with $k = 15$. In the case study, there is an edge between two authors if they co-author at least one paper. When $b = 2$, two authors are identified by `AKT` and there are 10 followers. We find that the authors saved in the anchored k-truss have strong connections to the original k-truss, while excluded by the network collapse.



(a) Yelp, k=10, b=1

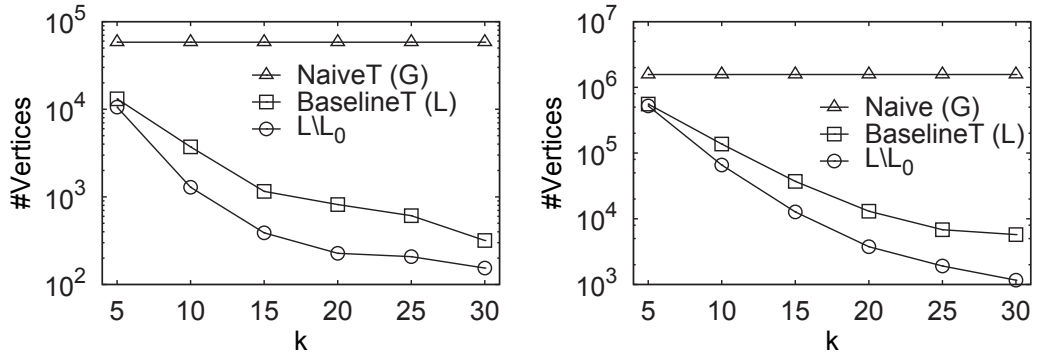


(b) DBLP, k=12, b=2

Figure 4.22: Case Studies

4.5.6 Efficiency of AKT

Reducing Candidate Anchors. Figure 4.23 reports the sizes of G, *edge onion layers* $\mathcal{L}$ and (k-1)-hull (i.e.,$\mathcal{L}\backslash\mathcal{L}_0$) on two networks `Brightkite` and `DBLP`. Recall that `NaiveT` need to check all vertices in G because anchoring vertices in $T_k(G)$ can also have followers, and the other algorithms (`BaselineT`, `BLT+C` and `AKT`) only consider the vertices from $\mathcal{L}$ as candidate anchors (Theorem 20). We also



(a) Brightkite, b=1 (b) DBLP, b=1

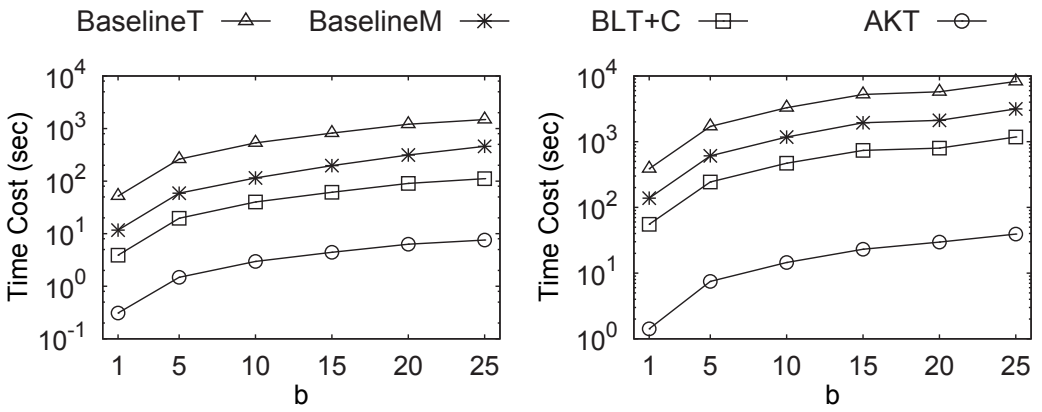
Figure 4.23: Reducing Candidate Anchors and Followers

report the size of (k-1)-hull ($H_{k-1}(G)$), which bounds the size of the candidate followers by Theorem 19. As expected, the sizes of $\mathcal{L}$ and $H_{k-1}(G)$ drop with the growth of k.

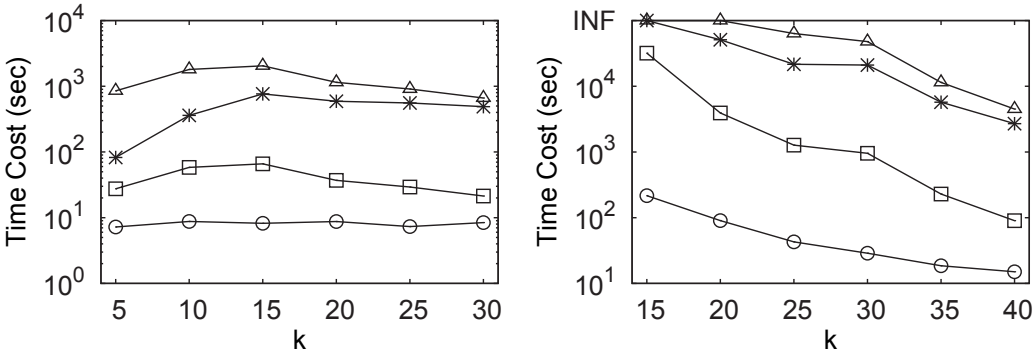
Pruning Candidate Followers. Figure 4.24 demonstrates the effectiveness of the pruning techniques which help us to eliminate non-promising candidate followers. Three algorithms were evaluated using the running time on four networks by varying b or k. We report that both pruning techniques significantly reduce the running time, especially the triangle hold path based layer-by-layer candidate follower search in `AKT` (Theorem 21 and 22).

Effect of k and b. Figure 4.24 also studies the impact of k and b on the four algorithms. `AKT` significantly outperforms the two baseline algorithms under all settings. We terminate the algorithms when the running time exceeds 10^5 seconds. The margin between `AKT` and other algorithms is not as large as that in `OLAK` because the computation of k-truss occupies larger runtime percentage in `AKT` than k-core in `OLAK`.

Different Datasets. Figure 4.25 reports the performance of the three algorithms on 10 networks. The datasets are ordered by their the number of edges. Not surprisingly, the performance of `BaselineT` is poor and cannot finish com-



(a) Brightkite, k=15 (b) Gowalla, k=15



(c) Facebook, b=20 (d) DBLP, b=20

Figure 4.24: Effect of Algorithms with Different k and b

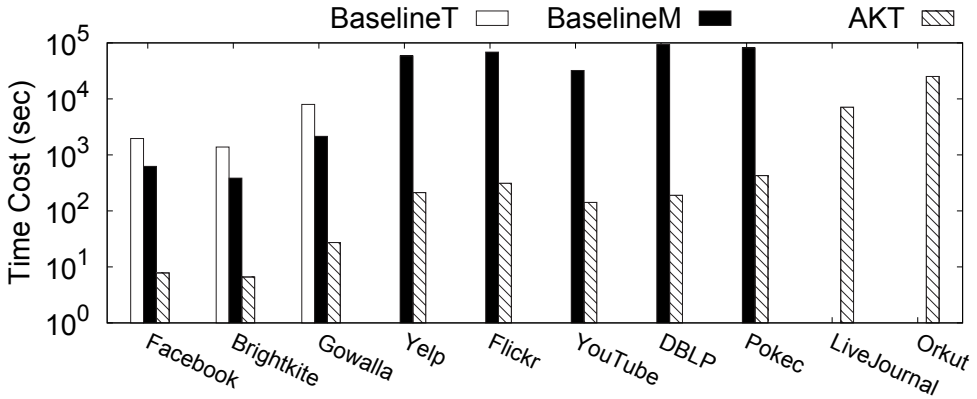


Figure 4.25: Running Time on Different Datasets

putation on 5 networks within 10^5 seconds. `BaselineM` outperforms `BaselineT` while is beaten by anchored k-truss by a large margin. We observed that the computation of triangle listing and k-truss occupies much more runtime percentage than k-core in `OLAK`, thus the outperforming margin in `AKT` is less than that in `OLAK`.

4.6 Conclusion

In this chapter, we study the problem of anchored k-core and the problem of anchored k-truss, which aims to *anchor* a set of vertices in a network such that the size of the resulting k-core and k-truss is maximized, respectively. The anchored k-core problem is proposed by Bhawalkar and Kleinberg *et* al. and shown to be NP-hard. Further considering the strength of ties, we propose the anchored k-truss problem and prove it is NP-hard. The hardness of the two problems motivates us to develop greedy algorithms. We design the *onion layer* and *edge onion layer* structures to maintain a small set of vertices and edges in the graph, such that (1) we only need to consider the vertices in the layer structure to find the best anchor; and (2) the layer structure enables us to develop an efficient follower computation algorithm using a layer-by-layer paradigm. The layer structure also helps us to develop early termination and pruning techniques to further prune follower and anchor candidates. Then we present our `OLAK` and `AKT` algorithms by combining all the proposed techniques. Empirical study shows that we can find critical vertices in the network whose participation may lead to a large number of followers. Extensive experiments on 10 real-life networks show that the proposed techniques improve performance of the naive solutions by orders of magnitude.

Chapter 5

Finding Critical Users

5.1 Introduction

The user engagement on social network has attracted significant interests over recent years [85, 89, 14]. Nevertheless, the connection between user importance and overall network engagement is still not clear. As mentioned in previous chapters, k-core is a simple and popular model based on degree constraint, which has been widely used to measure the network engagement [65, 26, 25, 2, 38]. Assuming all users in a community/group are initially engaged, each individual has two strategies, to remain engaged or drop out. Particularly, a user will remain engaged if and only if at least k of his/her friends are engaged (i.e., degree constraint). A user with less than k friends engaged will drop out, and his/her leave may be contagious and forms a cascade of the departure (i.e., collapse) in the network. When the collapse stops, the remaining engaged users corresponds to the well-known concept k-core, the maximal induced subgraph in which every vertex has at least k neighbors. The size of k-core can be used to measure the overall engagement of the social network. It motivates us to find critical users in social networks based the strength of user engagement.

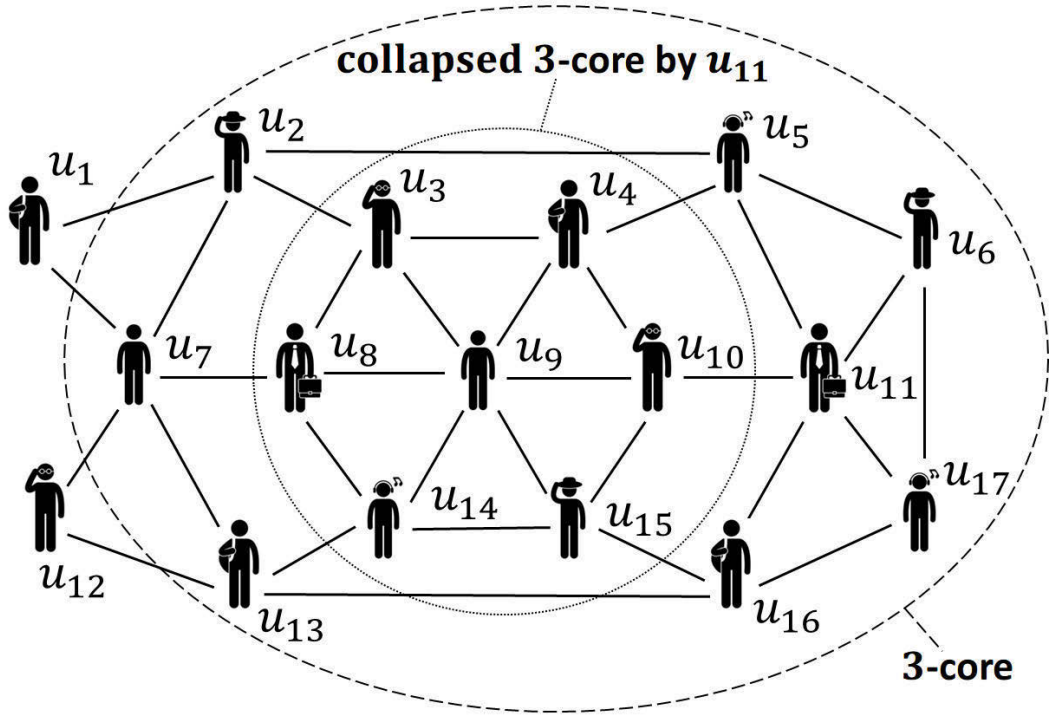


Figure 5.1: Motivating Example

A natural question is that, given a limited budget b, how to find b vertices (i.e., users) in a network so that we can get the smallest k-core by removing these b vertices. This problem is named the collapsed k-core problem in this chapter, which aims to collapse the engagement of the network with the greatest extent for a given budget b. By developing an efficient and scalable solution for this problem, we can quickly identify critical users whose leave will collapse the network most severely. These users are critical for the overall engagement of social networks. For instance, we can find most valuable users, to sustain or destroy the engagement of the networks. We can also evaluate the robustness of network engagement against the vertex attack.

Example 20. *Suppose there is a study group, and the number of friends in the group reflects the willingness of engagement for each member (i.e., user). If*

one drops out, he/she will weaken the willingness of his/her friends to remain engaged, which may incur the collapse of the group. As illustrated in Figure 5.1, we model 17 *members in a study group and their relationship as a network. According to the above engagement model with k=3, i.e., a person will drop out if there are less than* 3 *friends,* 15 *members will remain engaged; that is,* 3*-core of the network is the whole network excluding* u_1 *and* u_{12}*. Clearly, if users in* 3*-core drop out regardless the number of friends, e.g., attracted by another group, the network will further collapse. The extent of the collapse varies among different users. For instance, although* u_9 *has* 6 *friends in* 3*-core, the departure of* u_9 *will not further lead to the leave of other users because each of his/her neighbors still has* 3 *friends engaged. On the contrary, the leave of* u_{11} *will lead to the leave of 7 members in the group including* u_2*,* u_5*,* u_6*,* u_7*,* u_{13}*,* u_{16}*, and* u_{17}*. In this sense, it is more cost-effective to give* u_{11} *the incentive (e.g., bonus) to ensure his/her engagement or persuade him/her to leave the group.*

Note that the collapsed k-core problem finds the critical users inside of the k-core, while the anchored k-core problem in Chapter 4 finds the users outside of k-core. The collapsed k-core problem is to reinforce the stability of k-core communities, while the anchored k-core problem is to enlarge the k-core for the health of the entire social network. To the best of our knowledge, we are the first to propose and investigate the collapsed k-core problem. We prove the problem is NP-hard for any k value. To avoid enumerating all possible answer sets with size b, we resort to greedy heuristics where the best vertex is obtained in each iteration. Through theoretical analyses, we significantly reduce the number of candidate vertices to speed up the computation. We develop an efficient algorithm, namely CKC, to solve the collapsed k-core problem. Our comprehensive experiments on 9 real-life networks demonstrate the efficiency of our proposed techniques and the effectiveness of the collapsed k-core model.

Road Map. Section 5.2 introduces the collapsed k-core problem and present our complexity analysis on the problem. Section 5.3 presents our solution for the collapsed k-core problem. Section 5.4 evaluates the proposed algorithms. Section 5.5 concludes the chapter.

5.2 Preliminaries

In this section, we first give some necessary notations and recall the concept of k-core and its corresponding algorithm. Then, we formally define the collapsed k-core problem. Table 5.1 summarizes the notations used in this chapter.

5.2.1 Problem Definition

We consider an unweighted and undirected graph $G = (V, E)$, where V (resp. E) represents the set of vertices (resp. edges) in G. When the context is clear, we use a set S of vertices to represent the induced subgraph of G with $S \subseteq G$. We use n (resp. m) to denote the number of vertices (resp. edges) in the graph G and we assume $m > n$. We denote the adjacent vertices set of u in G by $NB(u, G)$, which is also called the neighbors set of u in G. We use $deg(u, S)$, the degree of u in S, to represent the number of adjacent vertices of u in S. $NB(u, S)$ (resp. $deg(u, S)$) is also written as $NB(u)$ (resp. $deg(u)$) when the context is clear. Given a subgraph S, $NB(S)$ denotes the union of the neighbors of the vertices in S. The concept of k-core has been widely used to describe cohesive subgraphs, which is formally defined as follows.

Definition 20. ***k-core****. Given a graph G and a positive integer k, an induced 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 every $u \in S$; and (ii) S is maximal, i.e., any subgraph $S' \supset S$ cannot be a k-core.*

Table 5.1: Summary of Notations

Notation	Definition
G	an unweighted and undirected graph
u, v, x	vertex in the graph
n, m	the number of vertices and edges in G
A	a set of collapsers vertices
G_x (G_A)	graph G collapsed by x (A)
k	the degree constraint
b	the budget for the number of collapsers
$C_k(G)$	k-core of G
$\|C_k\|$	the number of vertices in $C_k(G)$
$C_k(G_A)$	collapsed k-core with vertices in A deleted
$\mathcal{F}(x)$ ($\mathcal{F}(A)$)	followers of a collapser x (A)
$deg(u, S)$	the number of adjacent vertices of u in S
$NB(u, S)$	the adjacent vertices of u in S

As shown in Algorithm 7, the k-core of a graph G can be obtained by recursively removing the vertices whose degrees are less than k, with time complexity $\mathcal{O}(m)$ [12]. In real applications, the value of k is determined by users based on their requirement for cohesiveness. The resulting k-core will be more cohesive if the k value becomes larger.

In this chapter, once a vertex u in G is **collapsed**, it is always removed from k-core regardless of the degree constraint.

Definition 21. ***collapsed*** k***-core****. Given a graph G and a set $A \subseteq G$ of vertices, the collapsed k-core, denoted by $C_k(G_A)$, is the corresponding k-core of G with vertices in A removed.*

In addition to the deletion of the collapsed vertices in A, more vertices in $C_k(G)$ might be deleted as well due to the contagious nature of the k-core computation. These vertices are called **followers** of the collapsed vertices A, denoted by $\mathcal{F}(A, G)$, because they will remain in k-core if the vertices in A are not deleted. The size of the followers reflects the effectiveness of the collapsed vertices,

where $\mathcal{F}(A, G) = C_k(G) \setminus \{C_k(G_A) \cup A\}$. In the following, we may use **collapsers** to represent the collapsed vertices, and use $\mathcal{F}(A)$ to denote $\mathcal{F}(A, G)$ when the context is clear.

Problem Statement. Given a graph G, a degree constraint k and a budget b, the **collapsed k-core problem** aims to find a set A of b collapsed vertices in G so that the size of the resulting collapsed k-core, $C_k(G_A)$, is minimized; that is, $\mathcal{F}(A, G)$ is maximized.

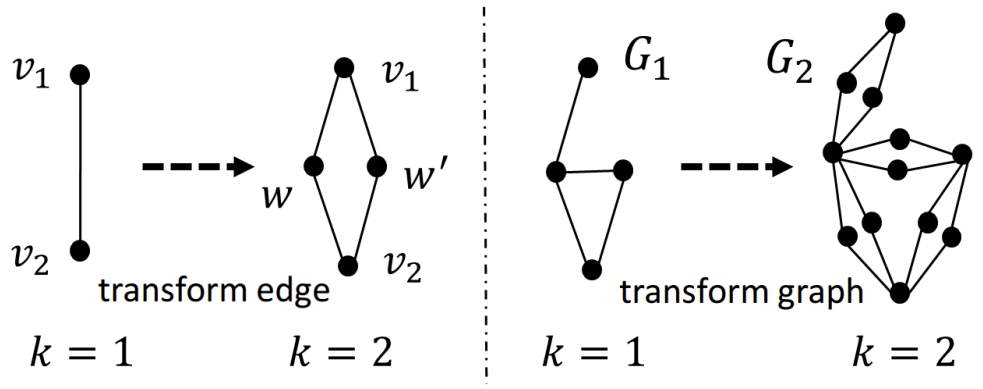
Example 21. *In Figure 5.1, if we set $k = 3$ and $b = 1$, the result of the collapsed k-core problem can be $A = \{u_{11}\}$ with $C_k(G_A) = \{u_3, u_4, u_8, u_9, u_{10}, u_{14}, u_{15}\}$ and $\mathcal{F}(A, G) = \{u_2, u_5, u_6, u_7, u_{13}, u_{16}, u_{17}\}$.*

5.2.2 Problem Complexity

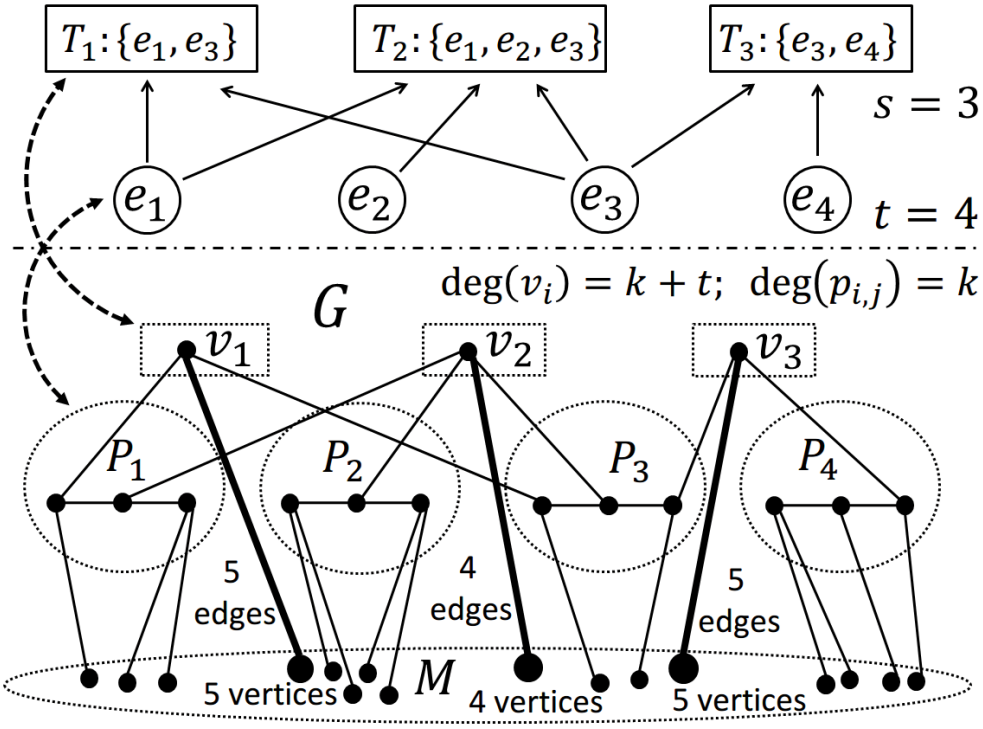
Theorem 23. *The collapsed k-core problem is NP-hard for any k.*

Proof. (1) When $k = 1$, we reduce the collapsed k-core problem to the maximum independent set problem [87]. To delete a vertex from 1-core during the collapsed 1-core computation, we have to remove all its adjacent vertices, i.e., make the vertex independent. Consequently, the problem of finding the maximum independent set S in a graph G is equivalent to finding the set of vertices $G \setminus S$ such that $G \setminus S$ is minimum and collapsing them can lead to an empty 1-core. Note that we need to try at most $n - 1$ times ($1 \leq b < n$) to find the minimum $G \setminus S$. Thus, we have the collapsed k-core problem is NP-hard when $k = 1$.

(2) When $k = 2$, we reduce the collapsed 2-core problem to the case of $k = 1$, which has been proved to be NP-hard. Given any graph G_1 with n vertices and m edges, we construct another graph G_2 with $n + 2m$ vertices and $4m$ edges as follows. For each edge (v_1, v_2) in G_1, we add two virtual vertices w and w' and



(a) Construction Example, $k = 2$



(b) Construction Example, $k = 3$ (can extend to $k > 3$)

Figure 5.2: Examples for NP-hardness Proof

construct the following four edges in G_2: (v_1, w), (w, v_2), (v_1, w') and (w', v_2), as shown in Figure 5.2 (a). An example of graph construction is also illustrated in Figure 5.2 (a). We do not need to include any virtual vertices in the optimal solution of collapsed 2-core because the influence of deleting a virtual vertex

can always be covered by deleting one of its two neighbor vertices (non-virtual vertices). Therefore, the deletion of each edge in G_1 during the computation is always mapped to the deletion of four corresponding edges in G_2. Then the optimal solution of collapsed 2-core on G_2 is also that of collapsed 1-core on G_1. As a result, the collapsed k-core problem is NP-hard when $k = 2$.

(3) When $k \geq 3$, we reduce the collapsed k-core problem to the maximum coverage problem [52]; that is finding at most b sets to cover the largest number of elements, where b is a given budget. Firstly, we consider an arbitrary instance of maximum coverage problem with s sets $T_1, .., T_s$ and t elements $\{e_1, .., e_t\} = \cup_{1 \leq i \leq s} T_i$. Then we construct a corresponding instance of the collapsed k-core problem in a graph G as follows.

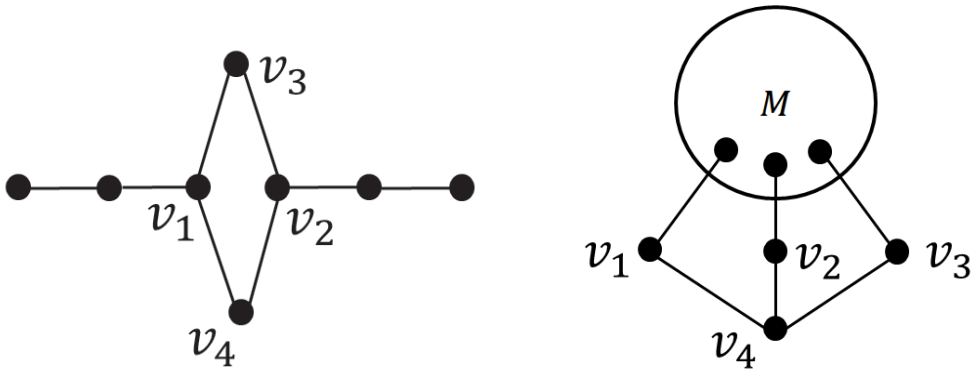
The set of vertices in G consists of three parts: M, V, and P. M consists of $(t + s)^4$ vertices in which every pair of vertices in M are adjacent. V consists of s vertices, v_1, v_2, ..., v_s, where vertex v_i corresponds to the set T_i for any $1 \leq i \leq s$. For each vertex v_i $(1 \leq i \leq s)$, we add $k + t - |T_i|$ edges from v_i to $k + t - |T_i|$ unique vertices in M. Here, by unique, we mean that each vertex in M can be used at most once when adding edges to vertices outside M. P consists of t parts P_1, P_2, ..., P_t, where each part P_i $(1 \leq i \leq t)$ corresponds to the element e_i and P_i consists of s vertices $p_{i,1}$,$p_{i,2}$, ..., $p_{i,s}$. For each P_i $(1 \leq i \leq t)$ we first add $s - 1$ edges, that is, for each $1 \leq j < s$, we add an edge from $p_{i,j}$ to $p_{i,j+1}$. For each set T_i $(1 \leq i \leq s)$ and each element e_j $(1 \leq j \leq t)$, if $e_j \in T_i$, we add an edge $(v_i, p_{j,i})$ in G. At this stage, the degree of each vertex in P is at most 3. Next, we add edges from vertices in P to unique vertices in M to guarantee that the degree of each vertex in P is exactly k. This can be done since $k \geq 3$. Then the construction of G is completed. Clearly, G is a k-core. Figure 5.2 (b) shows an example of the graph G with $k = 3$ constructed from 3 sets and 4 elements.

The key idea is that we ensure that: (i) only vertices in V need to be considered as collapsed vertices, since any vertex in M or P cannot have more followers than a vertex in V; (ii) none of the vertices in M will be deleted during the computation; (iii) all P_i have the same size for $1 \leq i \leq t$; and (iv) when a vertex v_i $(1 \leq i \leq s)$ is removed, for each part P_j $(1 \leq j \leq t)$ connected with v_i (i.e., $e_j \in T_i$), all vertices in P_j will be deleted due to degree constraint. By doing this, the optimal solution of the collapsed k-core problem corresponds to optimal solution of the maximum coverage problem. Since the maximum coverage problem is NP-hard, we prove that the collapsed k-core problem is NP-hard for any $k \geq 3$. □

We also show the properties of monotone and non-submodular towards the collapsed k-core problem in Theorem 24.

Theorem 24. *Let $f(A) = |\mathcal{F}(A)|$. We have f is monotone but not submodular for any k.*

Proof. Suppose there is a set $A' \supseteq A$. For every vertex u in $\mathcal{F}(A)$, u will still be deleted in the collapsed k-core with the collapsers set A', because removing vertices in $A' \setminus A$ cannot increase the degree of u. Thus $f(A') \geq f(A)$ and f is monotone. For two arbitrary collapsers sets A and B, if f is submodular, it must hold that $f(A \cup B) + f(A \cap B) \leq f(A) + f(B)$. We show that the inequality does not hold using counterexamples. When $k = 1$, we use the example shown in Figure 5.3 (a). Suppose $k = 1$, $A = \{v_1\}$ and $B = \{v_2\}$, we have $\mathcal{F}(A \cup B) = \{v_3, v_4\}$, $\mathcal{F}(A \cap B) = \mathcal{F}(A) = \mathcal{F}(B) = \emptyset$, so the inequation does not hold. When $k = 2$, we use the example shown in Figure 5.3 (b). Here, M is a complete graph with $4 \times k$ vertices. When $k = 2$, if $A = \{v_1\}$ and $B = \{v_2\}$, we have $\mathcal{F}(A \cup B) = \{v_3, v_4\}$, $\mathcal{F}(A \cap B) = \mathcal{F}(A) = \mathcal{F}(B) = \emptyset$, so the inequation does not hold. When $k > 2$, we add $k - 2$ edges between v_i and M, for each



(a) $k = 1$ (b) $k = 2$ (can extend to $k > 2$)

Figure 5.3: Examples for Non-submodular

$1 \leq i \leq 4$. We can prove that for $A = \{v_1\}$ and $B = \{v_2\}$, the inequation is still violated. □

5.3 Our Approach

5.3.1 Motivation

A straightforward solution of the collapsed k-core problem is to exhaustively enumerate all possible set A with size b, and compute the resulting collapsed k-core for each possible A. The time complexity of $\mathcal{O}(\binom{n}{b}m)$ is cost-prohibitive. Considering the NP-hardness of the problem, we resort to the greedy heuristic which iteratively finds the best collapser, i.e., the vertex with the largest number of followers. Clearly, we only need to consider the vertices in $C_k(G_A)$ since all other vertices will be deleted by degree constraint during k-core computation. Thus, a greedy algorithm is shown in Algorithm 19 with time complexity $\mathcal{O}(bnm)$, where n and m correspond to the number of candidate collapsers in each iteration (Line 3) and the cost of follower computation (Line 4).

Algorithm 19: **GreedyCKC**(G, k, b)

Input : G : a social network, k : degree constraint,
b : number of collapsers
Output: A : the set of collapsers

```
1 A := ∅; i := 0;
2 while i < b do
3     for each u ∈ C_k(G_A) do
4         Compute F(A ∪ u, G);
5     u* ← the best collapser in this iteration;
6     A := A ∪ u*; i := i + 1; update C_k(G_A);
7 return A
```

The number of vertices in $C_k(G_A)$ at Line 3 is still considerably large, which motivates us to develop two effective pruning rules to further reduce the candidate vertices in each iteration of the greedy algorithm.

5.3.2 Reducing Candidate Collapsers

For presentation simplicity, in this subsection, we introduce two pruning rules to find the vertex with the largest number of followers in the first iteration of the greedy algorithm (i.e., $A = \emptyset$). They can be immediately extended to the following iterations of the greedy algorithm by using the updated $C_k(G_A)$ to replace $C_k(G)$.

Theorem 25 indicates that only vertices with degree k in k-core and their neighbors in k-core can have followers. Particularly, P denotes the vertices in k-core of G with degree k, while T represents vertices in P as well as their neighbors within k-core.

Theorem 25. *Given a graph G and the set $P = \{u : deg(u, C_k(G)) = k\}$, if a collapsed vertex x has at least one follower, x is from T where $T = P \cup \{u : u \in C_k(G)\ \&\ NB(u, G) \cap P \neq \emptyset\}$; that is $|\mathcal{F}(x, G)| > 0$ implies $x \in T$.*

Proof. We prove that a vertex $x \in G \setminus T$ cannot have any follower. (1) If $x \in G \setminus C_k(G)$, x will be deleted in k-core computation and hence $|\mathcal{F}(x)| = 0$. (2) If $x \in C_k(G) \setminus T$, x survived in k-core computation and for each x's neighbor u within $C_k(G)$, we have $deg(u, C_k(G)) > k$ since $x \notin T$. Consequently, if x is deleted, we have $deg(u, C_k(G)) \geq k$; that is, the removal of x cannot be propagated to any of its neighbors regarding degree constraint and hence other vertices. It means x does not have any follower. Since $(G \setminus C_k(G)) \cup (C_k(G) \setminus T) \cup T = G$, we have $|\mathcal{F}(x, G)| > 0$ implies $x \in T$. □

In the following theorem, we further reduce the candidate vertices by excluding vertices which have been identified as followers of other vertices.

Theorem 26. *Given two vertices x and u in graph G, we have $\mathcal{F}(u) \subset \mathcal{F}(x)$ if $u \in \mathcal{F}(x)$.*

Proof. $u \in \mathcal{F}(x)$ implies that u will be deleted if x is collapsed. For every vertex in $\mathcal{F}(u)$, if x is collapsed, it will also be deleted since u will be deleted and collapsing x cannot increase degrees for vertices. Thus $\mathcal{F}(u) \subseteq \mathcal{F}(x)$. Since $u \in \mathcal{F}(x)$ and $u \notin \mathcal{F}(u)$, we have $\mathcal{F}(u) \subset \mathcal{F}(x)$. □

According to Theorem 26, in the procedure of finding a best collapser, every vertex which is a follower of a vertex can be excluded from candidate collapsers. Consequently, checking promising collapsers first, which may have large number of followers, can skip more vertices in the computation. Naturally, a vertex with more neighbors in the set P is more promising because all its neighbors in P will follow the vertex to be deleted. Thus, to further reduce the number of candidate collapsers, we try collapsing vertices in decreasing order of their degrees in P.

Algorithm 20: **CKC**(G, k)

Input : G : a social network, k : degree constraint,
Output: x : the best collapser

```
1 C_k(G) := compute C_k(G);
2 P := {u : deg(u, C_k(G)) = k};
3 T := P ∪ {u : u ∈ C_k(G) & NB(u, G) ∩ P ≠ ∅};
4 for each u ∈ T (Theorem 25) do
5 |   Compute F(u, G);
6 |   T := T \ F(u, G) (Theorem 26);
7 return the best collapser
```

5.3.3 CKC Algorithm

By taking advantage of two pruning rules in Theorems 25 and 26, Algorithm 20 illustrates the details of CKC algorithm which finds the best collapser for a given graph G (i.e., $b = 1$). Particularly, we first compute the k-core of graph G (Line 1) and find the set P of vertices with degree k in C_k (Line 2). According to Theorem 25, we find the set T of vertices in P, and vertices which are inside C_k and are neighbors of at least one vertex in P (Line 3). To compute $\mathcal{F}(u, G)$, we can continue the k-core computation in Line 1 with vertex u deleted (Line 5). We have the best collapser when the algorithm terminates.

To handle the general case with $b > 1$, our CKC algorithm can be easily fit to the greedy algorithm (replacing Line 3 and 4) to find the best collapser in each iteration. In order to avoid the re-computation of P (Line 2) and T (Line 3) in the following iterations, we incrementally update two sets at the end of each iteration. Specifically, let P_1 denote the vertices whose degrees are decreased to k during the computation and P_2 denote the vertices which are discarded during the computation, we have $P = P \cup (P_1 \setminus P_2)$; Towards the set T, we include new vertices in $NB(P_1)$ and delete vertices in $NB(P_2)$ which do not have any neighbor in the updated P. Additionally, if we find a vertex $u \in F(x)$

in one iteration of Algorithm 1, x is always a better candidate collapser than u in following iterations, because deleting other vertices cannot change the fact that x has more followers than u (Theorem 26). Actually, we do not need to consider u as a candidate in following iterations because u will be excluded from k-core whenever x is removed. In our implementation, we order the candidates by their number of neighbors in P in each iteration to prune more candidate collapsers.

5.4 Performance Evaluation

This section evaluates the effectiveness and efficiency of the proposed techniques through comprehensive experiments.

5.4.1 Experimental Setting

Algorithms. To the best of our knowledge, there is no existing work investigating the collapsed k-core problem and corresponding algorithms. In this chapter, we implement and evaluate the following algorithms.

- **Baseline**. The baseline greedy algorithm (Algorithm 19). In each iteration, it conducts collapsed k-core computation on every vertex in the updated k-core to find the best collapser.
- **CKC**. The greedy algorithm in which collapsed k-core algorithm (Algorithm 20) is used in each iteration.

Datasets. Nine real-life networks are deployed in our experiments and we assume all vertices in each network are initially engaged. The original data of `Yelp` is from `https://www.yelp.com.au/dataset_challenge`, `DBLP` is from `http://dblp.uni-trier.de/` and the others are from `http://snap.stanford.edu/`.

Table 5.2: Statistics of Datasets

Dataset	**Vertices**	**Edges**	d_{avg}	$\|C_{20}\|$
Facebook	4,039	88,234	43.7	1,854
Brightkite	58,228	194,090	6.7	900
Gowalla	196,591	456,830	4.7	3,841
Yelp	552,339	1,781,908	6.5	20,839
YouTube	1,134,890	2,987,624	5.3	18,890
DBLP	1,566,919	6,461,300	8.3	29,564
Pokec	1,632,803	8,320,605	10.2	10,817
LiveJournal	3,997,962	34,681,189	17.4	469,951
Orkut	3,072,441	117,185,083	76.3	2,242,775

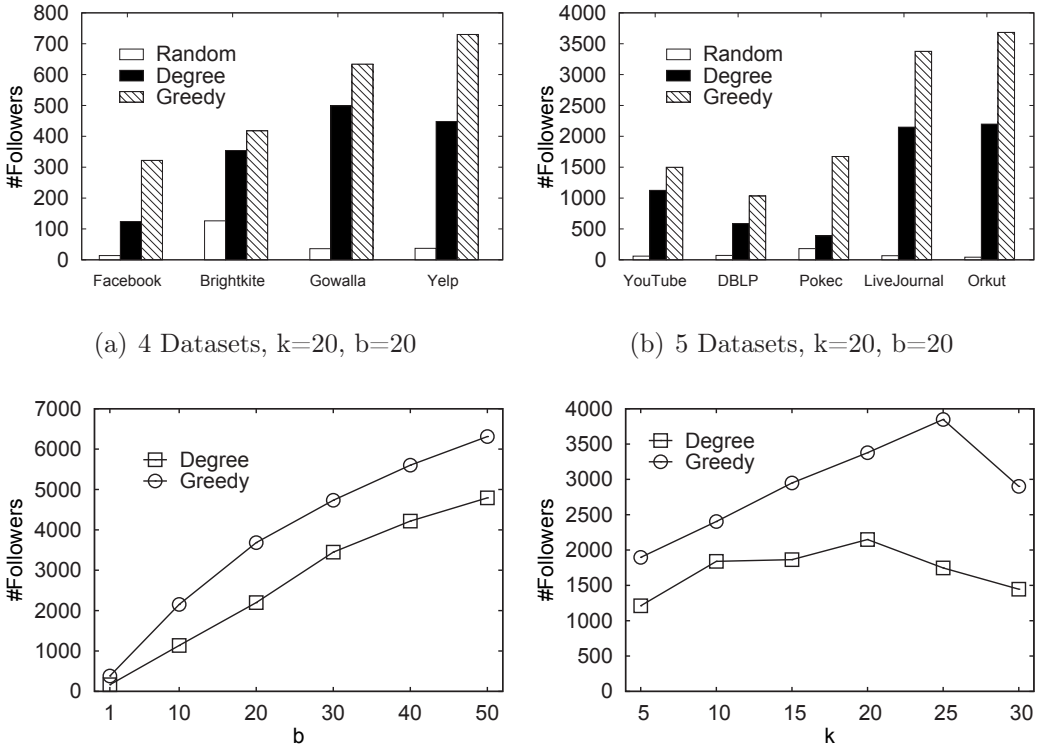
In DBLP, we consider each author as a vertex and there is an edge for a pair of authors if they have at least one co-authored paper. We use the original vertex and edges information in other datasets. Table 5.2 shows statistics of 9 datasets which are listed in increasing order of their edge numbers.

Parameters. We conduct experiments under different settings by varying degree constraint k and the budget of collapsers b. The default values of k and b are both 20. In the experiments, the range of k varies from 5 to 50 and the range of b varies from 1 to 100.

All programs are implemented in standard C++ and compiled with G++ in Linux. All experiments are performed on a machine with Intel Xeon 2.8GHz CPU and Redhat Linux System. We evaluate the effectiveness of the algorithms by reporting the number of the followers for resulting collapsers. The efficiency of the algorithms is measured by running time and the number of vertices accessed.

5.4.2 Effectiveness

We compare the number of followers produced by CKC with the results of other approaches, and also conduct a case study to demonstrate a detailed example of the collapsed k-core.



(a) 4 Datasets, k=20, b=20

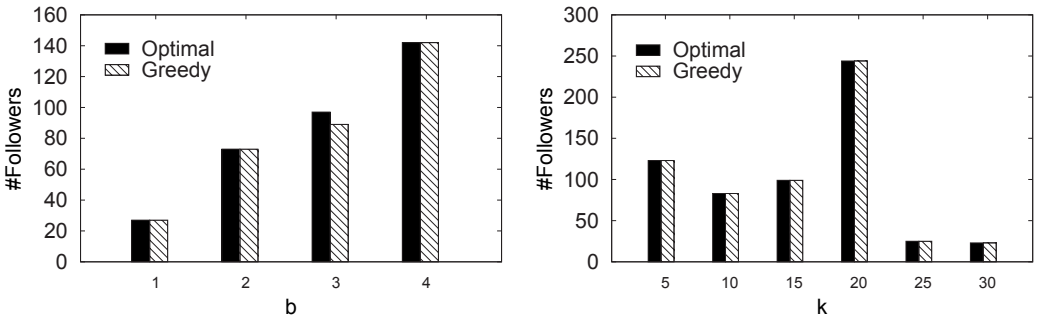
(b) 5 Datasets, k=20, b=20

(c) Orkut, k=20

(d) LiveJournal, b=20

Figure 5.4: Number of the Followers

Effectiveness of the Greedy Algorithm. Figure. 5.4 compares the number of followers w.r.t b collapsers identified by CKC algorithm with that of two other approaches, in which one randomly chooses b collapsers from vertices in k-core (**Random**) and the other chooses b collapsers in the candidate set T (Theorem 25) with the largest degrees (**Degree**). For *Random*, we report the average number of the followers for 100 independent testings . Figures 5.4 (a) and (b) show that although *Degree* based approach significantly improves the performance, but it is outperformed by our approach with a big margin. This implies that it is not effective to find collapsers simply based on degree information. Figures 5.4 (c) and (d) report the impact of b and k on the number of followers for CKC. The number becomes relatively small when k is small or large.



(a) Facebook, k=20 (b) Brightkite, b=2

Figure 5.5: Greedy vs Optimal

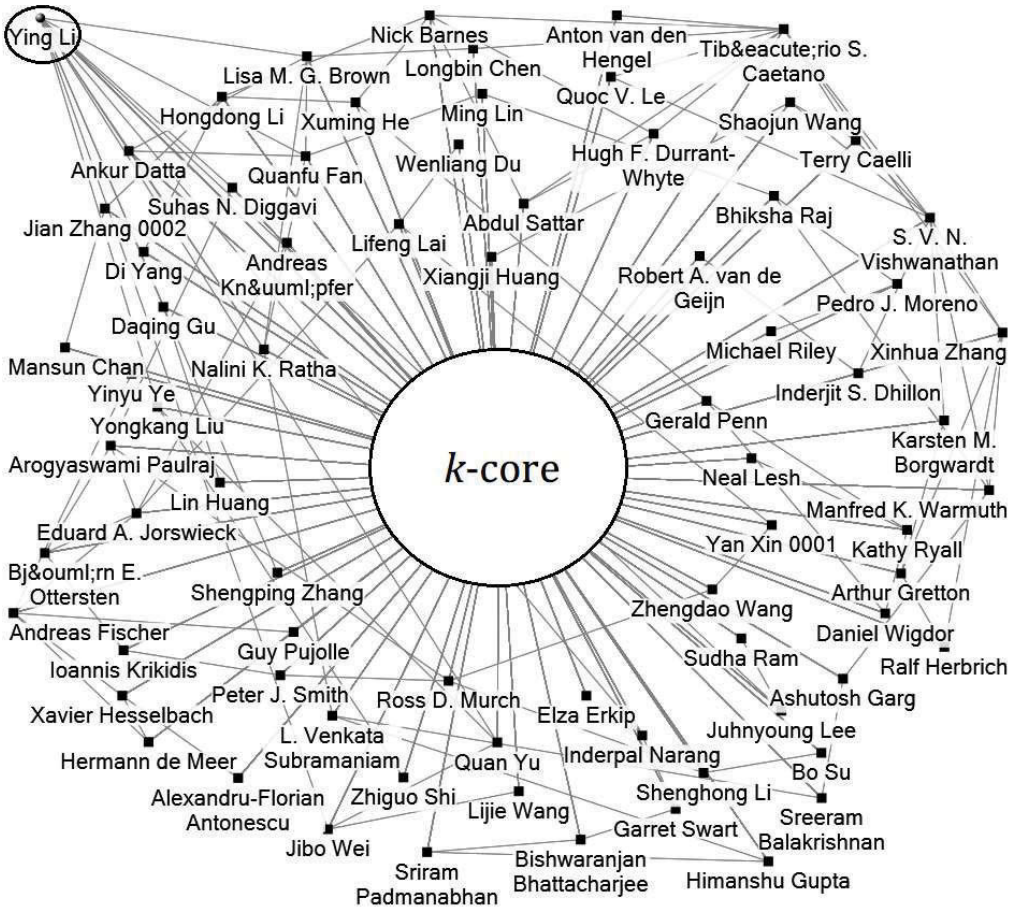


Figure 5.6: Case Study on DBLP, k=20, b=1

Greedy vs Exact. To further justify the effectiveness of the greedy approach, we also compare its performance with that of optimal algorithm (**Optimal**), which conducts exhaustively search on two relatively small networks with b varying from 1 to 4 on `Facebook` and k varying from 5 to 30 on `Brightkite`. Figure 5.5 shows that the greedy algorithm achieves the optimal solution except under one setting.

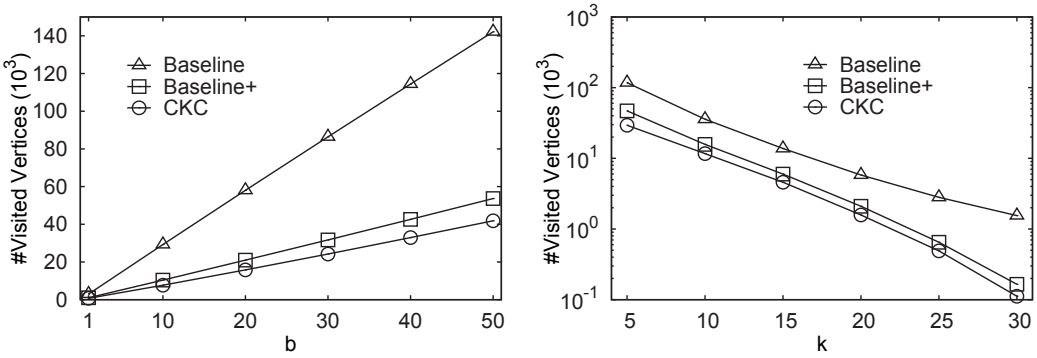
Case Study on DBLP. Figure 5.6 depicts the collapser identified by the greedy algorithm on `DBLP` with $b = 1$ and $k = 20$ as well as the corresponding followers. For a clear presentation, edges between each author and authors in k-core are integrated as one edge. It is interesting that the author "Ying Li" alone has 74 followers, and only 12 of them are neighbors of "Ying Li". Moreover, we observe that the followers include many professors and at least one IEEE fellow (Nalini K. Ratha). This shows the overall engagement of the network can be severely damaged by the leave of a few individuals.

5.4.3 Efficiency

We first investigate the efficiency of the individual techniques, then compare our CKC algorithm with Baseline.

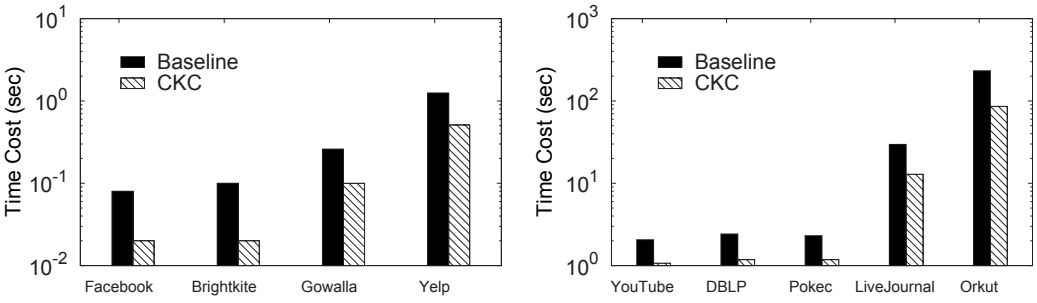
Evaluation of Individual Techniques. Figure 5.7 reports the number of visited vertices, i.e., the size of candidate collapsers, in three algorithms. Algorithm **Baseline**+ represents *Baseline* algorithm equipped with candidate collapsers reducing technique (Theorem 25). We can see the number of visited vertices significantly drops by Theorem 25 on `DBLP` for different k and b. It is reported that Theorem 26 further reduces the number of candidate collapsers, which is used in algorithm CKC.

Performance of Baseline and CKC. Figures 5.8 (a) and (b) report the per-

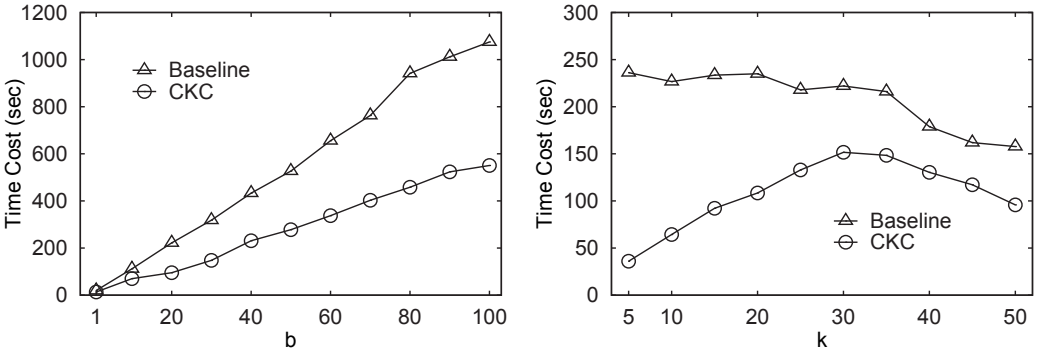


(a) DBLP, k=20 (b) DBLP, b=20

Figure 5.7: Effectiveness of Reducing Candidate Collapsers



(a) 4 Datasets, k=20, b=20 (b) 5 Datasets, k=20, b=20



(c) Orkut, k=20 (d) Orkut, b=20

Figure 5.8: Performance of Baseline and CKC

formance of two algorithms on 9 networks with $k = 20$ and $b = 20$. Datasets are ordered by their network sizes (i.e., the number of edges) where the largest

network `Orkut` has 117 million edges. We can see CKC runs several times faster than *Baseline* on all datasets. It is shown that CKC is also scalable to the growth of the network size, which identifies a set of 20 collapsers in 110 seconds on `Orkut`. Figures 5.8 (c) and (d) study the impact of k and b on two algorithms against `Orkut`, with b varying from 1 to 100 and k ranging from 5 to 50. We can see CKC is scalable towards the growth of b and outstanding on running time for different k, especially for small or large k. It is reported that CKC significantly outperforms *Baseline* under all settings.

5.5 Conclusion

In this chapter, we propose and study the problem of collapsed k-core, which intends to find a set of vertices whose deletion can lead to the smallest k-core of the network. We prove the problem is NP-hard for any given k. An efficient algorithm is proposed, which significantly reduces the number of candidate vertices to speed up the computation. Empirical study shows our method can find critical users in the network whose leave leads a large number of users to drop out. Extensive experiments on 9 real-life networks demonstrate our method is scalable on large size networks.

Chapter 6

Epilogue

The study on social networks becomes increasingly important with the growing capacity and activity in social networking sites, such as Facebook and Twitter. In this thesis, we consider three basic components in mining cohesive subgraph to better accommodate specific real-life applications. Accordingly, three fundamental objectives on social networks are explored: (i) discover subgraphs with cohesiveness on both user engagement and similarity; (ii) prevent unraveling of social networks based on subgraph decompositions; and (iii) find critical users to reinforce communities.

6.1 Conclusions

Towards objective (i), we propose a novel cohesive subgraph model, called (k,r)-core, which considers the cohesiveness of a subgraph on both graph structure and vertex attribute. We show that the problem of enumerating the maximal (k,r)-cores and finding the maximum (k,r)-core are both NP-hard. Several novel pruning techniques are proposed to substantially improve algorithm efficiency, including candidate size reduction, early termination, checking maximals and

upper bound estimation techniques. Effective search orders are devised for enumeration, finding the maximum, and maximal check algorithms. Comprehensive experiments on real-life data demonstrate that the (k,r)-core model can find interesting communities, and performance of two mining algorithms is effectively improved by proposed techniques.

Towards objective (ii), we study the problem of anchored anchored k-core, which was introduced by Bhawalkar and Kleinberg *e*t al. in the context of user engagement. The problem has been shown to be NP-hard and inapproximable. Further considering the strength of ties, we propose the anchored k-truss problem and prove it is NP-hard. Solving this problem can help us to identify the individuals whose participation is most critical to the overall engagement of strong tie communities. We design the *onion layer* and *edge onion layer* structures to maintain a small set of vertices and edges in the graph to significantly speed up computation. We also develop early termination and pruning techniques to further prune follower and anchor candidates. Then we present our `OLAK` and `AKT` algorithms by combining all the proposed techniques. Empirical study shows that we can find critical vertices in the network whose participation may lead to a large number of followers. Extensive experiments show that the two algorithms outperform the baselines by orders of magnitude.

Towards objective (iii), we propose the collapsed k-core problem to find the vertices whose leave can lead to the smallest k-core, i.e., to identify critical users for network engagement. Solving the problem helps us to find the most critical users whose leave may significantly break network engagement, i.e., lead a large number of other users to drop out. We prove the problem is NP-hard. Then, an efficient algorithm is proposed, which significantly reduces the number of candidate vertices. Comprehensive experiments on real-life social networks demonstrate the effectiveness and efficiency of proposed techniques.

6.2 Future Work

In many real-life networks, it is rather natural to consider both structure and attribute values in many graph problems. As cohesive subgraph mining is one of the most fundamental problems, we can expect a variety of extensions of (k,r)-core model can be used for different scenarios by (i) applying possible combinations of existing cohesive subgraph models (e.g., k-core, k-truss, clique, quasi-clique, and dense subgraph); and (ii) considering the cohesive subgraph computation on multi-dimensional networks [55].

The techniques developed in this thesis can shed light on the computation of these models. For instance, our study suggests that it is less efficient to sequentially apply the state-of-the-art techniques for each constraint (i.e., cohesive subgraph model). Instead, we need to carefully integrate the computation of the multiple constraints at each search step. Our study also indicates that, to deal with the multiple constraints, it is crucial to develop advanced early termination and maximal check techniques as well as design good visiting orders. When finding the (k,r)-trusses whose vertices form k-truss on the graph and clique on the similarity graph, the (k,k')-core upper bound technique can be directly applied to this problem because a k-truss is also a (k-1)-core. With similar rationale, other proposed techniques can also be extended or provide insights to the counterparts of new cohesive subgraph models.

Analyzing the network structure of these models is another important task to prevent network unraveling and improve network stability. The ideas in constructing the layer structures on k-core and k-truss give the inspiration that utilizing the vertex deletion order in computation of a model, which may efficiently find the vertices for preventing network unraveling. Considering the specific application scenarios, it is also interesting to efficiently find critical users on corresponding cohesive subgraph models, such as k-truss and k-plex.

REFERENCES

[1] How does facebook suggest groups for me to join? `https://www.facebook.com/help/382485908586472?helpref=uf_permalink`. Accessed: 15 Aug. 2017.

[2] J. Abello and F. Queyroi. Fixed points of graph peeling. In *ASONAM*, pages 256–263, 2013.

[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.-C. Chang, I. Korpeoglu, and Ö. Ulusoy. Distributed-core view materialization and maintenance for large dynamic graphs. *TKDE*, 26(10):2439–2452, 2014.

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

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

[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 *NIPS*, 2005.

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

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

[10] A. Anderson, D. P. Huttenlocher, J. M. Kleinberg, and J. Leskovec. Effects of user similarity in social media. In *WSDM*, pages 703–712, 2012.

[11] E. Bakshy, I. Rosenn, C. Marlow, and L. A. Adamic. The role of social networks in information diffusion. In *WWW*, pages 519–528, 2012.

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

[13] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In *SIGMOD*, pages 431–444, 2015.

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

[15] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *MSR*, pages 137–143, 2006.

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

[17] J. Cannarella and J. A. Spechler. Epidemiological modeling of online social network dynamics. *CoRR*, abs/1401.4208, 2014.

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

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

[20] 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.

[21] K. Chen and C. Lei. Network game design: hints and implications of player interaction. In *NETGAMES*, page 17, 2006.

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

[23] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *KDD*, pages 1240–1248. ACM, 2012.

[24] K. Cheung and L. F. Tian. Learning user similarity and rating style for collaborative recommendation. *Inf. Retr.*, 7(3-4):395–410, 2004.

[25] 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.

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

[27] M. S.-Y. Chwe. Structure and strategy in collective action 1. *American journal of sociology*, 105(1):128–156, 1999.

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

[29] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

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

[31] T. Dang and E. Viennet. Community detection based on structural and attribute similarities. In *International Conference on Digital Society*, pages 7–12, 2012.

[32] Y. Dong, J. Zhang, J. Tang, N. V. Chawla, and B. Wang. Coupledlp: Link prediction in coupled networks. In *SIGKDD*, pages 199–208, 2015.

[33] N. Eagle, A. S. Pentland, and D. Lazer. Inferring friendship network structure by using mobile phone data. *PNAS*, 106(36):15274–15278, 2009.

[34] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *SEA*, pages 364–375, 2011.

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

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

[37] Y. Fang, H. Zhang, Y. Ye, and X. Li. Detecting hot topics from twitter: A multiview approach. *J. Information Science*, 40(5):578–593, 2014.

[38] D. Garcia, P. Mavrodiev, and F. Schweitzer. Social resilience in online communities: the autopsy of friendster. In *COSN*, pages 39–50, 2013.

[39] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *JACM*, 23(1):43–49, 1976.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[41] C. Giatsidis, F. Malliaros, D. M. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *IAAI*, 2014.

[42] E. Gilbert and K. Karahalios. Predicting tie strength with social media. In *SIGCHI*, pages 211–220, 2009.

[43] M. K. Goldberg, S. Kelley, M. Magdon-Ismail, K. Mertsalov, and A. Wallace. Finding overlapping communities in social networks. In *SocialCom/PASSAT*, pages 104–113, 2010.

[44] M. S. Granovetter. The strength of weak ties. *American journal of sociology*, 78(6):1360–1380, 1973.

[45] J. Healy, J. Janssen, E. E. Milios, and W. Aiello. Characterization of graphs using degree cores. In *WAW*, pages 137–148, 2006.

[46] D. Hristova, M. Musolesi, and C. Mascolo. Keep your friends close and your facebook friends closer: A multiplex network approach to the analysis of offline and online social ties. In *ICWSM*, 2014.

[47] 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.

[48] X. Huang and L. V. S. Lakshmanan. Attribute-driven community search. *PVLDB*, 10(9):949–960, 2017.

[49] X. Huang, L. V. S. Lakshmanan, and J. Xu. Community search over big graphs: Models, algorithms, and opportunities. In *ICDE*, pages 1451–1454, 2017.

[50] X. Huang, W. Lu, and L. V. S. Lakshmanan. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *SIGMOD*, pages 77–90, 2016.

[51] J. J. P. III, S. Moreno, T. L. Fond, J. Neville, and B. Gallagher. Attributed graph models: modeling network structure with correlated attributes. In *WWW*, pages 831–842, 2014.

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

[53] 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.

[54] 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):888–893, 2010.

[55] M. Kivelä and e. Arenas. Multilayer networks. *Journal of Complex Networks*, 2(3):203–271, 2014.

[56] N. D. Lane, Y. Xu, H. Lu, S. Hu, T. Choudhury, A. T. Campbell, and F. Zhao. Enabling large-scale human activity inference on smartphones using community similarity networks (csn). In *UbiComp*, pages 355–364, 2011.

[57] P. Lee, L. V. S. Lakshmanan, and E. E. Milios. CAST: A context-aware story-teller for streaming social content. In *CIKM*, pages 789–798, 2014.

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

[59] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W. Ma. Mining user similarity based on location history. In *SIGSPATIAL*, page 34, 2008.

[60] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 26(10):2453–2465, 2014.

[61] H. Liu, Z. Hu, A. U. Mian, H. Tian, and X. Zhu. A new user similarity model to improve the accuracy of collaborative filtering. *Knowl.-Based Syst.*, 56:156–166, 2014.

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

[63] R. D. Luce and A. D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[64] H. Luo, C. Niu, R. Shen, and C. Ullrich. A collaborative filtering framework based on both local user similarity and global user similarity. *Machine Learning*, 72(3):231–245, 2008.

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

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

[67] A. Nanopoulos, H. Gabriel, and M. Spiliopoulou. Spectral clustering in social-tagging systems. In *WISE*, pages 87–100, 2009.

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

[69] J.-P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, D. Lazer, K. Kaski, J. Kertész, and A.-L. Barabási. Structure and tie strengths in mobile communication networks. *PNAS*, 104(18):7332–7336, 2007.

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

[71] R. Rotabi, K. Kamath, J. M. Kleinberg, and A. Sharma. Detecting strong ties using network motifs. In *WWW*, pages 983–992, 2017.

[72] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

[73] A. E. Sariyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *PVLDB*, 10(3):97–108, 2016.

[74] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *WWW*, pages 927–937, 2015.

[75] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, pages 285–295, 2001.

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

[77] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978.

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

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

[80] P. Singla and M. Richardson. Yes, there is a correlation - from social networks to personal behavior on the web. In *WWW*, pages 655–664, 2008.

[81] S. Sintos and P. Tsaparas. Using strong triadic closure to characterize ties in social networks. In *SIGKDD*, pages 1466–1475, 2014.

[82] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *PNAS*, 109(16):5962–5966, 2012.

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

[84] J. Wang, J. Cheng, and A. W. Fu. Redundancy-aware maximal cliques. In *KDD*, pages 122–130, 2013.

[85] X. Wang, R. Donaldson, C. Nell, P. Gorniak, M. Ester, and J. Bu. Recommending groups to users using user-group engagement and time-dependent matrix factorization. In *AAAI*, 2016.

[86] 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.

[87] G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization*, pages 185–208, 2001.

[88] B. Wu and X. Pei. A parallel algorithm for enumerating all the maximal k -plexes. In *PAKDD*, pages 476–483, 2007.

[89] 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.

[90] Y. Wu, R. Jin, X. Zhu, and X. Zhang. Finding dense and connected subgraphs in dual networks. In *ICDE*, pages 915–926, 2015.

[91] R. Xiang, J. Neville, and M. Rogati. Modeling relationship strength in online social networks. In *WWW*, pages 981–990, 2010.

[92] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In *SIGMOD*, pages 505–516, 2012.

[93] J. Yang, J. J. McAuley, and J. Leskovec. Community detection in networks with node attributes. In *ICDM*, pages 1151–1156, 2013.

[94] 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.

[95] 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.

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

[97] 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.

[98] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *ICDE*, pages 1049–1060, 2012.

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

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

[101] 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.

[102] R. Zhou, C. Liu, J. X. Yu, W. Liang, and Y. Zhang. Efficient truss maintenance in evolving networks. *CoRR*, abs/1402.2807, 2014.

[103] Q. Zhu, H. Hu, J. Xu, and W. Lee. Geo-social group queries with minimum acquaintance constraint. *CoRR*, abs/1406.7367, 2014.

[104] Z. Zou and R. Zhu. Truss decomposition of uncertain graphs. *Knowl. Inf. Syst.*, 50(1):197–230, 2017.