Efficient Core Computation in Bipartite and Multilayer Graphs

by

Boge Liu

B.E. Tsinghua University, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

IN THE SCHOOL

OF

Computer Science and Engineering

Thursday 16th July, 2020

Thesis/Dissertation Sheet

Surname/Family Name	:	**Liu**
Given Name/s	:	**Boge**
Abbreviation for degree as give in the University calendar	:	**PhD**
Faculty	:	**Engineering**
School	:	**Computer Science and Engineering**
Thesis Title	:	**Efficient Core Computation in Bipartite and Multilayer Graphs**

Abstract 350 words maximum: (PLEASE TYPE)

Graphs are widely used to model the relationships of entities in a large spectrum of applications including social networks, world wide web, collaboration networks, and biology. Cohesive subgraph mining, as a fundamental graph problem, extracts highly connected structures from large graphs. Among the cohesive subgraph models, the core model, in which each node from the subgraph subject to a minimum degree constraint, has attracted great attention due to its elegant property and effectiveness in graph analysis. However, the massive graph volume and rapid evolution present huge challenges for core computation, which need highly efficient solutions. In this thesis, we study the problems of core computation in bipartite graphs and multilayer graphs.

Firstly, we study the problem of (α,β)-core computation in bipartite graphs. We present an efficient algorithm for (α,β)-core computation based on a novel index such that the algorithm runs in linear time regarding the result size. We prove that the index only requires $O(m)$ space where m is the number of edges in the bipartite graph. We also devise an efficient algorithm with time complexity $O\ (\delta \cdot m\ )$ for index construction where δ is bounded by $\sqrt{m}$ and is much smaller than $\sqrt{m}$ in practice.

Secondly, we study the problem of (α,β)-core maintenance when the bipartite graph is dynamically updated. We show that we can decide whether a node should be updated or not by visiting its neighbours. Based on this locality property, we propose an efficient maintenance algorithm which only needs to visit a local subgraph near the inserted or removed edge. Furthermore, we discuss how to implement our maintenance algorithm in parallel.

Finally, we study the problem of core computation in multilayer graphs, which is challenging due to the various combinations of layers. We propose a novel concept named CoreCube, which records the results of core computation on every combination of layers. We develop efficient algorithms to compute the CoreCube and devise a hybrid storage method that achieves a superior trade-off between the size of CoreCube and the query time. Extensive experiments on real-life datasets demonstrate our algorithms are effective and efficient.

ORIGINALITY STATEMENT

INCLUSION OF PUBLICATIONS STATEMENT

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

Publications can be used in their thesis in lieu of a Chapter if:

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

Please indicate whether this thesis contains published material or not:

☐ This thesis contains no publications, either published or submitted for publication

☒ Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement

☐ This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the UNSW Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Candidate's Name	**Signature**	**Date (dd/mm/yy)**

Abstract

Graphs are widely used to model the relationships of entities in a large spectrum of applications including social networks, world wide web, collaboration networks, and biology. Cohesive subgraph mining, as a fundamental graph problem, extracts highly connected structures from large graphs. Among the cohesive subgraph models, the core model, in which each node from the subgraph subject to a minimum degree constraint, has attracted great attention due to its elegant property and effectiveness in graph analysis. However, the massive graph volume and rapid evolution present huge challenges for core computation, which need highly efficient solutions. In this thesis, we study the problems of core computation in bipartite graphs and multilayer graphs.

Firstly, we study the problem of (α, β)-core computation in bipartite graphs. The (α, β)-core is an induced subgraph of a bipartite graph with node degrees not smaller than α in the upper layer and not smaller than β in the lower layer, respectively. We present an efficient algorithm for (α, β)-core computation based on a novel index such that the algorithm runs in linear time regarding the result size (thus, the algorithm is optimal since it needs at least linear time to output the result). We prove that the index only requires $O(m)$ space where m is the number of edges in the bipartite graph. We also devise an efficient algorithm with time complexity $O(\delta \cdot m)$ for index construction where δ is bounded by $\sqrt{m}$ and is much smaller than $\sqrt{m}$ in practice.

Secondly, we study the problem of (α, β)-core maintenance when the bipartite graph is dynamically updated. We show that we can decide whether a node should be updated or not by visiting its neighbors. Based on this locality property, we propose an efficient maintenance algorithm which only needs to visit a local subgraph near the inserted or removed edge. Furthermore, we discuss how to handle the case when a batch of edges are inserted/removed and how to implement our maintenance algorithm in parallel.

Finally, we study the problem of core computation in multilayer graphs, which is challenging due to the various combinations of layers. We propose a novel concept named CoreCube, which records the results of core computation on every combination of layers. We develop efficient algorithms to compute the CoreCube and devise a hybrid storage method that achieves a superior trade-off between the size of CoreCube and the query time. Extensive experiments on real-life datasets demonstrate our algorithms are effective and efficient.

Publications

- **Boge Liu**, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, Jingren Zhou. Efficient (α, β)-core Computation: an Index-based Approach. WWW 2019. (Chapter 3)

- **Boge Liu**, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, Jingren Zhou. Efficient (α, β)-core Computation in Bipartite Graphs. VLDB Journal 2020. (Chapter 4)

- **Boge Liu**, Fan Zhang, Chen Zhang, Wenjie Zhang, Xuemin Lin. CoreCube: Core Decomposition in Multilayer Graphs. WISE 2019. (Chapter 5)

- Chen Zhang, Fan Zhang, Wenjie Zhang, **Boge Liu**, Ying Zhang, Lu Qin, Xuemin Lin. Exploring Finer Granularity within the Cores: Efficient (k, p)-Core Computation. ICDE 2020.

Dedication

To my parents

my relatives

my friends

For their love and support

Acknowledgements

First and foremost, I would like to deliver my sincere gratitude to my supervisor Prof. Wenjie Zhang and co-supervisor Prof. Xuemin Lin for their support and guidance on my Ph.D study. I have benefited greatly from their sincere conversations, harsh criticism and continuous encouragement. They not only enlighten me with knowledge and encourage me to overcome the difficult time, but also affect me with their passion, diligence, and earnestness in the research.

Secondly, I am grateful to have the opportunity to work closely with Dr. Long Yuan and Dr. Fan Zhang who gave me constant encouragement and much invaluable advice. They are active and inspiring in discussions and shared with me a lot of insightful ideas. They have walked me through all the stages of the writing of this thesis. Without their consistent and illuminating instructions, this thesis could not have reached its present form.

Thirdly, I am glad to work in a wonderful Database Research Group. I would like to thank the following group members: Prof. Ying Zhang, Dr. Xin Cao, Dr. Zengfeng Huang, Dr. Longbin Lai, Dr. Weiwei Liu, Dr. Yuan Long, Dr. Xiang Wang, Dr. Xiaoyang Wang, Dr. Fan Zhang, Dr. Yixiang Fang, Dr. Jianye Yang, Dr. Shiyu Yang, Dr. Xing Feng, Dr. Shenlu Wang, Dr. Xubo Wang, Dr. Fei Bi, Dr. Wei Li, Dr. Haida Zhang, Dr. Chen Zhang, Dr Yang Yang, Ms. Xiaoshuang Chen, Ms. Yuting Zhang, Mr. Xuefeng Chen, Mr. You Peng, Mr. Kai Wang, Mr.

Qingyuan Linghu, Mr. Yizhang He, Mr. Jiahui Yang. The time we spent together will be memorized forever.

Finally, I would like to thank my family members, for their continued affection and support throughout my Ph.D.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Graphs are widely used to model the relationships of entities in a large spectrum of applications including social networks, world wide web, collaboration networks, and biology. Cohesive subgraph mining, as a fundamental graph problem, extracts highly connected structures from large graphs. The cohesive subgraph model of k-core has attracted great attention due to its elegant property and linear-time computation [Sei83]. Given a graph G, k-core is a maximal subgraph of G such that every node in the subgraph is connected to at least k other nodes within the same subgraph. It has a wide range of applications such as social contagion [UBMK12], [ZZQ$^+$17], influential spreader identification [KGH$^+$10], collapse prediction [MDFM19], anomalies detection [SEF16], core resilience [LSE$^+$18], and user engagement study [MV13].

Despite the power of k-core decomposition in mining cohesive subgraphs and analyzing the graph structure, k-core is basically defined based on general (single-layer and unipartite) graphs. It is worth noting that general graphs sometimes are inadequate in expressing inherent properties in real-graphs. For example, many real-world relationships across various entities are inherently modelled as bipartite graphs, such as customer-product networks [WDVR06], user-page networks [BXG$^+$13], gene co-expression networks [KKND11], collaboration networks [Ley02], etc. At the same time, there are usually multiple types of interactions (edges) among entities (nodes) in real-life graphs, e.g., the relationship between two users in a social network can be friends, colleagues, relatives and so on. The entities and interactions are usually modelled as a multilayer graph, where each layer records a certain type of interaction among entities [DMR16]. Using the traditional k-core model defined on general graphs cannot fully exploit such inherent properties in these graphs. Furthermore, considering that the graph usually undergoes highly dynamic updates nowadays and the requests for core computation can be issued frequently in real applications, it is too expensive to compute core from scratch each time. Therefore, in this thesis, we concentrate on extending the traditional k-core model to bipartite graphs and multilayer graphs, and developing efficient core computation and maintenance algorithms for them.

Section 1.1 briefly describes the background and motivations of investigating the above problems, explain the challenges faced by these problems, and introduce the main ideas of our solutions. Section 1.2 summarizes the contributions of this thesis for each problem investigated. Thesis organization is presented in Section 1.3.

1.1 Motivations

1.1.1 (α, β)-core Computation in Bipartite Graphs

With the proliferation of bipartite graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing bipartite graph data. Among them, the problem of computing (α, β)-core of a bipartite graph for given α and β has been recently studied in [CB15, DLHM17]. Formally, a bipartite graph $G = (U, V, E)$ is a graph with the nodes divided into two separate sets, U and V, such that every edge connects one node in U to another node in V. Given $G = (U, V, E)$ and two integers α and β, the (α, β)-core of G consists of two node sets $U' \subseteq U(G)$ and $V' \subseteq V(G)$ such that the subgraph induced by $U' \cup V'$ is the maximal subgraph of G in which all the nodes in U' have degree at least α and all the nodes in V' have degree at least β.

Applications. Computing (α, β)-cores has many real applications.

(1) Online group recommendation. Group recommendation aims at recommending products to a group of users who may or may not share similar tastes, e.g., recommending movies for friends to watch together [AYRC$^+$09, YCL14, CM13, GLRW13]. Fault-tolerant group recommendation is proposed to deal with missing values in incomplete data and has shown its effectiveness in group recommendation [GMRS11, PG09, NSRK14]. A key step in fault-tolerant group recommendation is to compute fault-tolerant subspace clusters for each user in the group. For a given user u_q, a subspace cluster is a set of u_q's similar users, which is exploited by collaborative filtering [NSNK12] to compute the relevance of products to u_q. Recently, to accelerate the computation of fault-tolerant subspace clustering, [DLHM17] has shown that (α, β)-core is an efficient way of computing fault-tolerant subspace clustering. For a user u_q and user-specific parameters α and β, all the users in the (α, β)-core are treated as u_q's fault-tolerant subspace cluster. Since the α and β values can vary greatly based on users' preference and the degree of tolerance for missing values [DLHM17], efficiently computing (α, β)-core is a critical procedure for online fault-tolerant group recommendation.

(2) Fraudsters detection. In social networks, such as Facebook and Twitter, users and pages form a user-page bipartite graph in which the edge indicates a user likes

a page. To promote certain pages, fraudsters use a larger number of fake accounts to inflate the *Like* counts for these pages; this results in a large number (β) of users liking a few (α) pages. (α, β)-core with small α and large β can facilitate the detection of such fraudsters [BXG+13, AIB+13]. Similar fraud scenario also occurs in E-Commence/Online-Shopping, for example, fraudsters may improve the ranking of certain items by adding items to fake accounts' shopping lists.

(3) A key step to other problems in bipartite graphs. Computing (α, β)-core can also serve as a key step to solve other important graph problems, such as biclique computation [LSH08, ZPR+14] and quasi-biclique computation [LSLW, LLW10].

Motivations. In the literature, an online algorithm [DLHM17] is proposed for the computation of (α, β)-core. However, it has to traverse the entire graph to compute the (α, β)-core for given α and β. This makes it impractical to real scenarios, especially while taking into consideration that real bipartite graphs nowadays can be very large and the requests for computing (α, β)-core can be issued frequently. For example, the consumer-product networks of Amazon or Alibaba often reach billion-scale [LSY03, WHZ+18]; in the application of online group recommendation, there can be millions of groups issuing recommendation requests at the peak time [GXL+10, YCL14, LSY03]. To recommend products to these groups, we need to compute (α, β)-core with different α and β for each user in every group. Therefore, numerous underlying computations of (α, β)-cores with different combinations of α and β have to be processed in realtime (typically within half a second [LSY03]). However, it is shown in our experiments that, even in Orkut dataset with 327 million edges, existing method spends 236 seconds to compute (α, β)-core for a group of ten users. For fraud detection case, we also need to do lots of (α, β)-core computations to union results together because fraudsters may hide behind different combinations of α and β values [BXG+13, AIB+13] and we don't want to miss out the suspicious people. Motivated by this, we aim to devise an index-based optimal algorithm (linear time with respect to the result size) to compute the (α, β)-core for given α and β.

Challenges. To achieve our goal, we adopt an index-based approach. Straightforwardly, if we store the (α, β)-cores for all possible α and β combination, we can obtain the (α, β)-core in optimal time for given α and β. Nevertheless, this approach will take $O(n^3)$ space to store all results where n is the number of nodes in a bipartite graph. Obviously, this is prohibitive for a very large graph. Below, we present the challenges to be overcome in this paper.

- *Challenge 1: Optimal (α, β)-core computation vs Space Efficiency.* Considering that even one particular (α, β)-core for a given $(\alpha,\ \beta)$ may have $O(n)$ size and there could be $O(n^2)$ different combinations of α and β values, it is a challenge to develop a compact index such that we can compute (α, β)-core for given α and β in optimal time.

- *Challenge 2: Efficient index construction.* The proposed index is built upon the results of core decomposition on bipartite graphs. Note that core decomposition on general (single-layer and unipartite) graphs [BZ03] requires $O(m)$ time, simply extending this strategy to bipartite graphs with two disjoint node sets will lead to $O(\mathsf{dmax} \cdot m)$ time, where dmax is the maximum degree of nodes and m is the number of edges in G. dmax could be very large in real graphs (e.g., $\mathsf{dmax} > 10^7$ in Web Trackers dataset), such method is impractical for large graphs. Hence, it is a challenge to devise an efficient algorithm to construct the index.

Our Solutions. In this paper, we address the above challenges. Regarding challenge 1, we propose an index-based algorithm to process the (α, β)-core queries. We observe that although the relationships among (α, β)-cores with arbitrarily given α and β values are complicated, for a fixed $\alpha(\beta)$, the (α, β)-cores are monotonously included with respect to the increase of $\beta(\alpha)$. Following this observation, we define $\beta_{\max,\alpha}$ and $\alpha_{\max,\beta}$ for the nodes in U and V respectively, and we prove that for given α and β, the corresponding (α, β)-core can be determined through $\beta_{\max,\alpha}$ and $\alpha_{\max,\beta}$ uniquely. However, using $\beta_{\max,\alpha}$ and $\alpha_{\max,\beta}$ alone cannot achieve the goal of optimal (α, β)-core query processing. Therefore, we further organize nodes and devise an index structure named BiCore-Index. Based on BiCore-Index, to answer an (α, β)-core query, our query processing algorithm only needs to visit the nodes contained in the (α, β)-core once, which means the running time of the algorithm is only dependent on the result size rather than the size of the given graph (thus, optimal).

Regarding challenge 2, although BiCore-Index is functional and can support optimal (α, β)-core query processing, it is compact and we non-trivially prove that the size of BiCore-Index can be bounded by $O(m)$, where m is the number of edges in G. In addition, to efficiently construct the BiCore-Index, we first present a basic algorithm by iterating the entire graph $\mathsf{dmax}_U + \mathsf{dmax}_V$ times, where dmax_U and dmax_V are the maximum degree of nodes in U and V, respectively, and its time complexity is $O(\mathsf{dmax} \cdot m)$, where $\mathsf{dmax} = \max\{\mathsf{dmax}_U, \mathsf{dmax}_V\}$. However, dmax could be very large in real graphs [BA99]. Therefore, we improve the basic algorithm and further propose an efficient algorithm by sharing the computation during the index construction. We show that the time complexity of our proposed algorithm is $O(\delta \cdot m)$, where δ is the maximum value such that the (δ, δ)-core in G is nonempty and is bounded by $\sqrt{m}$. In our experiments, it is shown that δ is much smaller than $\sqrt{m}$ in practice.

1.1.2 (α, β)-core Maintenance in Bipartite Graphs

Although, BiCore-Index is useful in bipartite graphs for online group recommendation and frustrater detection [LYL+19], in real applications, such as online social networks [KNT10], web graph [OZZ07], and collaboration network [AHL12], graphs are generally dynamic, i.e., the graphs are frequently updated by node/edge insertion/deletion. For example, Facebook has more than 1.3 billion users and approximately 5 new users join Facebook every second [OMK15]; Twitter has more than 300 million users and 3 new users join Twitter every second [OMK15]. Therefore, supporting graph updates efficiently is important for the practical applicability of a graph algorithm in real applications. In the literature, numerous studies on the fundamental graph problems on dynamic graphs have been conducted, such as core maintenance problem in unipartite graphs [SGJS+13, ZYZQ17], reachability [FLL+11], densest subgraphs [ELS15], and pattern matching [ZLWX14].

Motivated by this, we aim to develop efficient BiCore-Index maintenance algorithms in dynamic graphs. Furthermore, as today's graphs grow in scale [LGHB07, DBS18] and current commodity servers are generally equipped with multi-cores [SB13], it is natural to solve graph problems in parallel [DBS18, SRM14]. Therefore, we also investigate the problem of implementing our algorithms in parallel.

Challenges and our solutions. As graphs are frequently updated in many applications, BiCore-Index should support efficient maintenance when the graph is dynamic. The state-of-art core maintenance algorithms on unipartite graphs (general graphs) require extra neighbor information for each node and auxiliary data structures [ZYZQ17] to maintain an order of nodes. Although the state-of-art core maintenance algorithms only need to use $O(n)$ extra space on general graphs, extending the techniques for general graphs to maintain index in bipartite graphs makes the space cost reach $O(\mathsf{dmax} \cdot n)$ because the containment relationship of (α, β)-core is more complicate than general k-core. Hence, it is a challenge to devise efficient algorithms that can maintain BiCore-Index without extra space cost. Moreover, existing core maintenance algorithms [LYM13, ZYZQ17, WQZ+16] focus on single-core computation because the insertion/removal of edges spreads influence among nodes in a complicate way and it is hard to predict the core change without processing nodes in a certain order. Therefore, it is a challenge to maintain BiCore-Index in a parallel manner. In summary, we need to answer the following two questions:

- How to update BiCore-Index in dynamic graphs efficiently?
- Can we develop effective parallel algorithms for BiCore-Index maintenance?

Regarding the first question, we first propose an algorithm to maintain BiCore-Index in dynamic graphs by reducing unnecessary computation during the procedure of updating BiCore-Index. Then, we show that we can decide whether a node in BiCore-Index should be updated or not by visiting its neighbors locally. Based on this locality property, we further devise a locality-based algorithm that updates BiCore-Index locally. Regarding the second question, we find that the updating process can be split into independent subprocesses which can be executed based on the BiCore-Index before update. To update BiCore-Index, we merge the results by selecting the largest (insertion) or smallest (removal) value computed among all subprocesses. Moreover, we discuss about how to maintain BiCore-Index when a batch of edges are inserted and removed.

1.1.3 Core Decomposition in Multilayer Graphs

In real-life networks, there are usually multiple types of interactions (edges) among entities (nodes), e.g., the relationship between two users in a social network can be friends, colleagues, relatives and so on. The entities and interactions are usually modelled as a multilayer graph, where each layer records a certain type of interaction among entities [DMR16]. Because of the strong modeling paradigm to handle various interactions among a set of entities, there are significant existing studies of multilayer graphs, e.g., [BGHS12, LSQ+18]. Previous works usually focus on mining dense structures from multilayer graphs according to given parameters, e.g., [ZZL18]. Nevertheless, graph decomposition, as a fundamental graph problem [WQZ+16], remains largely unexplored on multilayer graphs.

Core decomposition (or k-core decomposition), as one of the most well-studied graph decomposition, is to compute the core number for every node in the graph [Sei83]. It is a powerful tool in modeling the dynamic of user engagement in social networks. In practice, a user u tends to adopt a new behavior if there are a considerable number of friends (e.g., the core number of u) in the group who also adopted the same behavior [MV13]. Core decomposition is also theoretically supported by Nash equilibrium in game theory [BKL+15]. It has a variety of applications, e.g., graph visualization [AHDBV05a], internet topology [CHK+07] and user engagement [ZLZ+18, ZZZ+17]. Extending the single-layer core decomposition to multilayer graphs is a critical task which can benefit a lot of applications considering the various real-world interactions between entities.

Given a multilayer graph, the multilayer k-core on a set of layers is defined as a set of nodes whose minimum degree in the induced subgraph of each layer is at least k. The core number of a node on a set of layers is the largest k such that the multilayer k-core on these layers contains the node. Multilayer core decomposition on a set of layers is to compute the core number for each node on these layers. In this paper, we propose CoreCube which records the core numbers of each node for every combination of layers in a multilayer graph. In the following, we show the details for some application examples.

User Engagement Evaluation. In social networks, users may participate in multiple groups with different themes, where each group forms a layer in the multilayer graph. For instance, the authors in a coauthor network have different coauthor relationship on different venues (conferences or journals). For any given user-interested combination of venues (correspond to layers), CoreCube of the coauthor network can immediately answer the engagement level for each author, i,e, the core numbers [MV13]. Given a degree constraint k, we can also immediately retrieve a cohesive user group from CoreCube, i.e., the multilayer k-core.

Biological Module Analysis. In biological networks, different interactions between the modules are detected with different methods due to data noise and technical limitations [HYH$^+$05]. Analyzing module structure according to single method, i.e., on a single layer, may not be accurate. CoreCube allows us to study the connections between modules for any combination of potential methods. Thus, we can find co-expression clusters and verify the effectiveness of detection methods.

Challenges and Our Solutions. To conduct core decomposition in multilayer graphs, we face two main challenges: computation and storage.

- *Challenge 1: Efficient CoreCube Computation.* Although core decomposition on a single-layer graph can be computed in linear time, it becomes very challenging on a multilayer graph because the combination number of layers is exponential to the number of layers. In the general case, no polynomial-time algorithm may exist for computing the CoreCube. To the best of our knowledge, there is only one similar work [GBG17] where the algorithms can be adapted to compute the CoreCube while it is hard to share the computation among different combination of layers.

- *Challenge 2: Effective CoreCube Storage.* Furthermore, in order to efficiently retrieve multilayer k-core from CoreCube, we need to store the computation results into disks. Considering the result size is exponential to the number of layers, it is a challenge to develop a storage method which uses as few space as possible while supports quick retrieval of query results.

Regarding Challenge 1, we develop an efficient algorithms that assigns an upper bound of core number to each node and gradually converges the upper bound to its real value. The initial upper bound is carefully selected based on previous computation results such that our algorithm can save as much computation as possible. Regarding Challenge 2, we devise a method which avoids duplicated storage by only recording the difference between core number in each layer. With our storage methods, multilayer k-core can be quickly retrieved by summing up all the values stored in related files.

1.2 Contributions

In this section, we summarize the contributions of our thesis. We propose efficient and effective approaches to handle the three problems discussed above. For each of these problems, our contributions are briefly summarized below.

1.2.1 (α, β)-core Computation in Bipartite Graphs

For (α, β)-core computation in bipartite graphs, the main contributions of this thesis are summarized below.

1. *The first space-efficient index-based work to compute (α, β)-core .* We propose a non-trivial space-efficient index structure, BiCore-Index, with the size bounded by $O(m)$. To the best of our knowledge, this is the first linear space

index structure to support the optimal computation of (α, β)-core in bipartite graphs.

2. *Efficient algorithms to construct the index.* We carefully consider the computation sharing between two node sets of the bipartite graph when conducting the core decomposition and devise an efficient algorithm to construct the BiCore-Index. We show that the time complexity of our proposed algorithm is $O(\delta \cdot m)$, where δ is the maximum value such that the (δ, δ)-core in G is nonempty and is bounded by $\sqrt{m}$. In our experiments, it is shown that δ is much smaller than $\sqrt{m}$ in practice.

3. *Parallel implementation of index construction algorithms.* To further accelerate the computation of BiCore-Index, we discuss the parallel implementation of our index construction and maintenance algorithms.

4. *Extensive performance studies on real datasets from various domains.* We conduct extensive performance studies on ten real graphs and two synthetic graphs. The experimental results demonstrate the efficiency of our proposed algorithms. The experimental results on real and synthetic graphs (more than 1 billion edges) demonstrate that our algorithms achieve up to 5 orders of magnitude speedup for computing (α, β)-core, and up to 3 orders of magnitude speedup for index construction compared with existing techniques.

1.2.2 (α, β)-core Maintenance in Bipartite Graphs

For core maintenance in bipartite graphs, the main contributions of this thesis are summarized below.

1. *Efficient index maintenance algorithm for dynamic graphs.* We develop a locality-based algorithm to update BiCore-Index, which decide whether a node

in BiCore-Index should be updated or not by visiting its neighbors locally. Moreover, we discuss about how to maintain BiCore-Index when a batch of edges are updated.

2. *Efficient parallel maintenance algorithm.* We devise an efficient parallel index maintenance algorithms by splitting the updating process into independent subprocesses and merging the results by selecting the largest or smallest value computed among all subprocess.

3. *Extensive experiments on real datasets.* We demonstrate the efficiency of our proposed algorithm with ten real graphs and two synthetic graphs. The experimental results show that our algorithm achieves up to 4 orders of magnitude speedup for index maintenance compared with existing techniques.

1.2.3 Core Decomposition in Multilayer Graphs

For core computation in multilayer graphs, the main contributions of this thesis are summarized below.

1. We propose efficient algorithms to compute the CoreCube. Several theorems reveal the inner characteristics of multilayer core decomposition.

2. We devise a hybrid storage method which has a superior trade-off between query processing time and storage size.

3. Extensive experiments demonstrate that our CoreCube computation and query processing are faster than baselines by more than one order of magnitude.

1.3 Organization

This rest of this thesis is organized as follows.

- Chapter 2 provides a survey of the related work.
- Chapter 3 presents the structure of our space-efficient BiCore-Index which answers (α, β)-core query in optimal time, and propose our BiCore-Index constructions algorithms.
- Chapter 4 describes our BiCore-Index maintenance algorithm and explains how to implement it in parallel.
- Chapter 5 presents our CoreCube computation algorithms and space-efficient CoreCube storage method.
- Chapter 6 concludes the thesis and discusses several possible directions for future work.

Chapter 2

Related Work

In this chapter, we will provide an overview of some works related to the three problems discussed in this thesis.

2.1 Cohesive Subgraph Detection in Unipartite Graphs

Seidman first introduces k-core in [Sei83]. [BZ03] gives an efficient linear-time algorithm for core decomposition. Given a graph, the k-core of every input k naturally forms a hierarchical graph decomposition. Core decomposition is applied to many areas of importance, e.g., graph visualization [AHDBV05a], internet topology [CHK+07] and so on. Core decomposition is also studied in weighted graphs [GTV11b], attributed graphs [FCLH16, FCL+17, FCC+17, FWC+18a], multilayer graphs [LZZ+19], and directed graphs [GTV11a, FWC+18b]. Algorithms for core number maintenance in dynamic graphs are proposed in [SGJS+13, SGJS+16, ZYZQ17]. Application of k-core can be found in social networks [GTV11b, YQL+17a, PZZ+18, WCL+18, FCL+18, WYL+19], graph visualization[AHDBV05b, ZP12], protein interaction network analysis [WA05, BH03] and so on. Other cohesive subgraph models are also studied recently, such as clique [YQL$^+$15, YQL$^+$16a, FYC$^+$19, YQZ$^+$18], k-edge connected component [YQL$^+$16b, YQL$^+$17b], and k-mutual-friend subgraph model [ZYZ$^+$18]. A variety of cohesive subgraph models are proposed to handle different scenarios. One of the earliest model is clique [LP49] where every vertex is adjacent to every other vertex in the subgraph. The over-restrictive definition of clique leads to many relaxed models, e.g., n-clique [Luc50], k-plex [SF78], and quasi-clique [ARS02]. Cohesive subgraph models have a lot of applications on different disciplines, such as social networks [MV13, ZZQ$^+$18, ZYZ$^+$18], protein networks [AUANK$^+$03] and brain science [DJN$^+$13].

2.2 Dense Subgraphs in Bipartite Graphs

(α, β)-core is first introduced in [ABF$^+$07]. [CB15], [DLHM17], and [LYL$^+$19] extend the linear k-core mining algorithm to compute (α, β)-core. [Hoc98] generalizes the k-clique on unipartite graph to biclique on bipartite graphs. [Pee03] proves that finding the maximum edge biclique is NP-complete. [ZPR$^+$14] proposes an efficient algorithm to enumerate all bicliques. [SLGL06, LSLW08] relax the definition of biclique to introduce quasi-biclique on bipartite graphs and propose heuristic algorithms to enumerate all quasi-bicliques. [SP18] defines a framework of bipartite subgraphs based on the butterfly motif (2,2-biclique) to model the dense regions in a hierarchical structure. [SMST18] proposes efficient algorithms for counting the butterfly motif.

2.3 Bipartite Graph Models

[GL04] shows that all complex networks can be decomposed into underlying bipartite structures sharing some important statistics. [GL06] model bipartite graphs by assigning degree distribution to each node set separately. [KTV97] uses a Markov chain rewiring algorithm to generate bipartite graphs. Preferential attachment process, which is popularly use in generating scale free networks, is studied on bipartite graphs by [GL06]. [AKP17] extends block two-level Erdős-Rényi model [KPPS14] to bipartite graphs and reproduces both degree distributions and degreewise metamorphosis coefficients like real graphs. Application-specific generative bipartite graph models are also widely studied, including pollination networks in ecology [DFBG09, SRTU09], and protein-domain networks in biology [NOHA09].

2.4 Multilayer Graphs

As a powerful paradigm to model complex networks, multilayer graphs received a lot of interests in the literature [DMR16]. Most existing works focus on mining dense structures on multilayer networks. Zhang *e*t al. [ZZQ+17] detect cohesive subgraphs on a 2-layer graph where one layer corresponds to user engagement and the other corresponds to user similarity. Wu *e*t al. [WJZZ15] find subgraphs where each subgraph is dense on one layer and connected on the other layer. Jethava and Beerenwinkel [JB15] study the densest common subgraph problem to find a subgraph maximizing the minimum average degree on all the layers of a graph. They propose a greedy algorithm without approximation guarantees. Zhu *e*t al. [ZZL18] introduce the notion of coherent cores on multilayer graphs and search diversified coherent k-cores with top sizes on multilayer graphs. Li *e*t al. [LSQ+18] find persistent k-cores on a temporal graph where each layer corresponds to a time span. Galimberti et al. [GBG17] study core decomposition and densest subgraph extraction on multilayer graphs.

Chapter 3

Efficient (α,β)-core Computation in Bipartite Graphs

3.1 Introduction

Many real-world relationships across various entities can be modelled as bipartite graphs, such as customer-product networks [WDVR06], user-page networks [BXG$^+$13], gene co-expression networks [KKND11], collaboration networks [Ley02], etc. With the proliferation of bipartite graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing bipartite graph data. Among them, the problem of computing (α, β)-core of a bipartite graph for given α and β has been recently studied in [CB15, DLHM17]. Formally, a bipartite graph $G = (U, V, E)$ is a graph with the nodes divided into two separate sets, U and V, such that every edge connects one node in U to another node in V. Given $G = (U, V, E)$ and two integers α and β, the (α, β)-core of G consists of two node sets $U' \subseteq U(G)$ and $V' \subseteq V(G)$ such that the subgraph induced by $U' \cup V'$ is the maximal subgraph of G in which all the nodes in U' have degree at least α

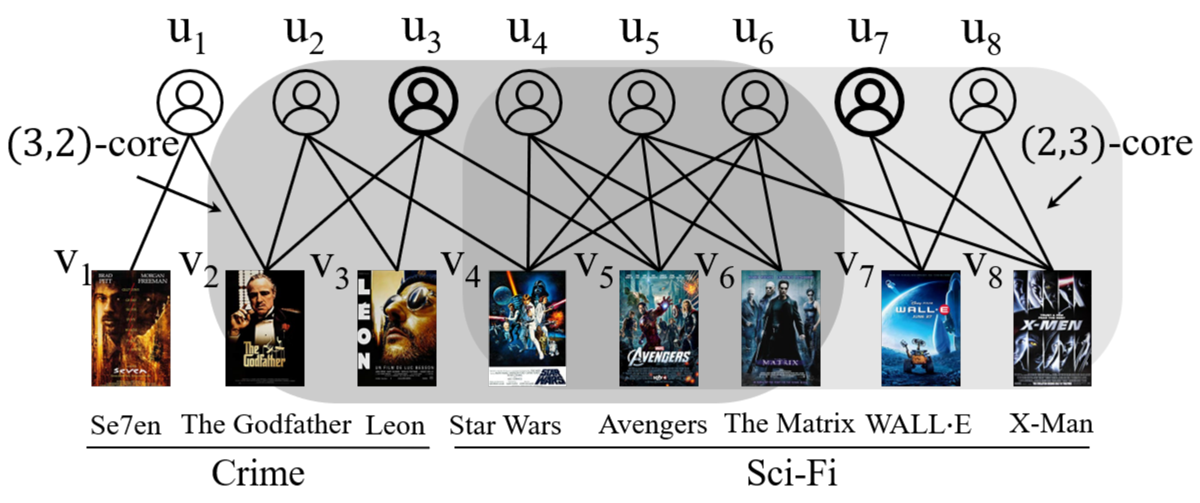


Figure 3.1: Part of a customer-movie network

and all the nodes in V' have degree at least β

Applications. Computing (α,β)-cores has many real applications.

(1) Online group recommendation. Group recommendation aims at recommending products to a group of users who may or may not share similar tastes, e.g., recommending movies for friends to watch together [AYRC$^+$09, YCL14, CM13, GLRW13]. Fault-tolerant group recommendation is proposed to deal with missing values in incomplete data and has shown its effectiveness in group recommendation [GMRS11, PG09, NSRK14]. A key step in fault-tolerant group recommendation is to compute fault-tolerant subspace clusters for each user in the group. For a given user u_q, a subspace cluster is a set of u_q's similar users, which is exploited by collaborative filtering [NSNK12] to compute the relevance of products to u_q. Recently, to accelerate the computation of fault-tolerant subspace clustering, [DLHM17] has shown that (α,β)-core is an efficient way of computing fault-tolerant subspace clustering. For a user u_q and user-specific parameters α and β, all the users in the (α,β)-core are treated as u_q's fault-tolerant subspace cluster. Since the α and β values can vary greatly based on users' preference and the degree of tolerance for

missing values [DLHM17], efficiently computing (α, β)-core is a critical procedure for online fault-tolerant group recommendation.

Example 1.1: Figure 3.1 shows part of the customer-movie network in the IMDB (https://www.imdb.com/) where each node in U represents a user, each node in V represents a movie and each edge indicates the customer has a preference for the movie. Assume that u_3 and u_7 are given as a group and the user-specific α and β for u_3 and u_7 are $(3, 2)$ and $(2, 3)$, respectively. Fault-tolerant group recommendation method first computes $(3, 2)$-core and $(2, 3)$-core, and uses $\{u_2, u_3, u_4, u_5, u_6\}$ and $\{u_4, u_5, u_6, u_7, u_8\}$ as fault-tolerant subspace clusters for u_3 and u_7, respectively. Then it conducts collaborative filtering based on the subspace clusters to further calculate the movie preference. In this case, Sci-Fi movies would be recommended to the group as both u_3 and u_7 have a preference for Sci-Fi movies. $\square$

(2) Fraudsters detection. In social networks, such as Facebook and Twitter, users and pages form a user-page bipartite graph in which the edge indicates a user likes a page. To promote certain pages, fraudsters use a larger number of fake accounts to inflate the *Like* counts for these pages; this results in a large number (β) of users liking a few (α) pages. (α, β)-core with small α and large β can facilitate the detection of such fraudsters [BXG$^+$13, AIB$^+$13]. Similar fraud scenario also occurs in E-Commence/Online-Shopping, for example, fraudsters may improve the ranking of certain items by adding items to fake accounts' shopping lists.

(3) A key step to other problems in bipartite graphs. Computing (α, β)-core can also serve as a key step to solve other important graph problems, such as biclique computation [ZPR$^+$14] and quasi-biclique computation [LSLW08, LLW10].

Motivations. In the literature, an online algorithm [DLHM17] is proposed for the computation of (α, β)-core. However, it has to traverse the entire graph to compute the (α, β)-core for given α and β. This makes it impractical to real scenarios, especially while taking into consideration that real bipartite graphs nowadays can be very large and the requests for computing (α, β)-core can be issued frequently. For example, the consumer-product networks of Amazon or Alibaba often reach billion-scale [LSY03, WHZ$^+$18]; in the application of online group recommendation, there can be millions of groups issuing recommendation requests at the peak time [GXL$^+$10, YCL14, LSY03]. To recommend products to these groups, we need to compute (α, β)-core with different α and β for each user in every group. Therefore, numerous underlying computations of (α, β)-cores with different combinations of α and β have to be processed in realtime (typically within half a second [LSY03]). However, it is shown in our experiments that, even in Orkut dataset with 327 million edges, the existing method spends 236 seconds to compute (α, β)-core for a group of ten users. For fraud detection case, we also need to do lots of (α, β)-core computations to union results together because fraudsters may hide behind different combinations of α and β values [BXG$^+$13, AIB$^+$13] and we don't want to miss out the suspicious people.

Challenges. To achieve our goal, in this paper, we adopt an index-based approach. Straightforwardly, if we store the (α, β)-cores for all possible α and β combination, we can obtain the (α, β)-core in optimal time for given α and β. Nevertheless, this approach will take $O(n^3)$ space to store all results where n is the number of nodes in a bipartite graph. Obviously, this is prohibitive for a very large graph. Below, we present the challenges to be overcome in this paper.

- *Challenge 1: Optimal (α, β)-core computation vs Space Efficiency.* Considering that even one particular (α, β)-core for a given $(\alpha,\ \beta)$ may have $O(n)$ size and there could be $O(n^2)$ different combinations of α and β values, it is a challenge to develop a compact index such that we can compute (α, β)-core for given α and β in optimal time.

- *Challenge 2: Efficient index construction.* The proposed index is built upon the results of core decomposition on bipartite graphs. Note that core decomposition on unipartite graphs (general graphs) [BZ03] requires $O(m)$ time, simply extending this strategy to bipartite graphs with two disjoint node sets will lead to $O(\mathsf{dmax} \cdot m)$ time (details in Section 3.4.1), where dmax is the maximum degree of nodes and m is the number of edges in G. However, dmax could be very large in real graphs (e.g., $\mathsf{dmax} \geq 10^7$ in Web Trackers dataset), such method is impractical for large graphs. Therefore, it is a challenge to devise an efficient algorithm to construct the index.

Contributions. In this paper, we overcome all the above challenges. The preliminary version is published in [LYL+19]. The main contributions of this work are summarized as follows:

(1) The first space-efficient index-based work to compute (α, β)-core . In this paper, we propose a non-trivial space-efficient index structure, BiCore-Index, with the size bounded by $O(m)$. To the best of our knowledge, this is the first linear space index structure to support the optimal computation of (α, β)-core in bipartite graphs.

(2) Efficient algorithms to construct the index. We carefully consider the computation sharing between two node sets of the bipartite graph when conducting the core decomposition and devise an efficient algorithm to construct the BiCore-Index. We show that the time complexity of our proposed algorithm is $O(\delta \cdot m)$, where δ is the maximum value such that the (δ, δ)-core in G is nonempty and is bounded by $\sqrt{m}$. In our experiments, it is shown that δ is much smaller than $\sqrt{m}$ in practice.

(4) Parallel implementation of index construction. To further accelerate the com-

putation of BiCore-Index and its maintenance on dynamic graphs, we discuss the parallel implementation of our index construction and maintenance algorithms.

(5) Extensive performance studies on real datasets from various domains. We conduct extensive performance studies on ten real graphs and two synthetic graphs. The experimental results demonstrate the efficiency of our proposed algorithms.

Outline. Section 3.2 gives the problem definition and the existing solution. Section 3.3 introduces our proposed index, BiCore-Index, and the optimal algorithm to compute (α, β)-core for arbitrary α and β. Section 3.4 presents algorithms to construct BiCore-Index. Section 3.5 discusses how to implement our index construction algorithms in parallel. Section 3.6 evaluates our algorithms using extensive experiments and Section 3.7 concludes the paper.

3.2 Preliminaries

A bipartite graph $G = (U, V, E)$ is a graph consisting of two disjoint sets of nodes U and V such that every edge from $E \subseteq U \times V$ connects one node of U and one node of V. We use $U(G)$ and $V(G)$ to denote the two disjoint node sets of G and $E(G)$ to represent the edge set of G. We denote the number of nodes in $U(G)$ and $V(G)$ as n_U and n_V, the total number of nodes as n and the number of edges in $E(G)$ as m. The degree of a node $u \in U(G) \cup V(G)$, denoted by $\mathsf{deg}(u, G)$, is the number of neighbors of u in G. We also use $\mathsf{dmax}_U(G)$ ($\mathsf{dmax}_V(G)$) to denote the maximum degree among all the nodes in $U(G)$ ($V(G)$), i.e., $\mathsf{dmax}_U(G) = \max\{\mathsf{deg}(u, G) | u \in U(G)\}$ ($\mathsf{dmax}_V(G) = \max\{\mathsf{deg}(v, G) | v \in V(G)\}$). For simplicity, we omit G in the notations if the context is self-evident. For a bipartite graph G and two node sets $U' \subseteq U(G)$ and $V' \subseteq V(G)$, the bipartite subgraph induced by U' and V' is the subgraph G' of G such that $U(G') = U'$, $V(G') = V'$ and $E(G') = E(G) \cap (U' \times V')$.

Definition 2.1: ((α, β)-core) Given a bipartite graph G and two integers α and β, the (α, β)-core of G, denoted by $\mathcal{C}_{\alpha,\beta}$, consists of two node sets $\mathcal{U} \subseteq U(G)$ and $\mathcal{V} \subseteq V(G)$ such that the bipartite subgraph g induced by $\mathcal{U} \cup \mathcal{V}$ is the maximal subgraph of G in which all the nodes in $\mathcal{U}$ have degree at least α and all the nodes in $\mathcal{V}$ have degree at least β, i.e., $\forall u \in \mathcal{U}, \mathsf{deg}(u, \mathrm{g}) \geq \alpha \land \forall v \in \mathcal{V}, \mathsf{deg}(v, \mathrm{g}) \geq \beta$.

Similar to the traditional k-core in unipartite graph, (α, β)-core is not necessarily connected. Note that when $\alpha = \beta$, (α, β)-core degenerates to the k-core in unipartite graph. An important property of k-core in unipartite graph is that if $k_1 \geq k_2$, k_1-core must be contained in k_2. Similar property can also be found in (α, β)-core. Specifically, Given a bipartite graph G, $\mathcal{C}_{\alpha,\beta}$ is contained in $\mathcal{C}_{\alpha',\beta'}$ if $\beta' \leq \beta$ and $\alpha' \leq \alpha$.

Problem Statement. In this paper, we study the problem of efficient computation of (α, β)-core for given α and β. For ease of presentation, we refer a request of computing the (α, β)-core for given α and β as an *(α, β)-core query* and denote it as $Q_{\alpha,\beta}$. Our object is to design a time-optimal algorithm for processing (α, β)-core queries on large bipartite graphs.

Existing Solution. Given an (α, β)-core query $Q_{\alpha,\beta}$, the state-of-the-art algorithm to compute $\mathcal{C}_{\alpha,\beta}$ is proposed in [DLHM17]. Intuitively, it computes $\mathcal{C}_{\alpha,\beta}$ by iteratively removing nodes in $U(G)$ with degree less than α and nodes in $V(G)$ with degree less than β until no more nodes can be removed. The above algorithm adopts an online paradigm to process (α, β)-core queries. For a query $Q_{\alpha,\beta}$, its time complexity to compute $\mathcal{C}_{\alpha,\beta}$ is $O(m)$ in the worst case. Nevertheless, the graphs are typically very large in real applications (e.g., there are 327 million edges in Orkut dataset). Therefore, this algorithm cannot satisfy the real-time requirements for (α, β)-core queries since it needs to traverse the whole graph for a $Q_{\alpha,\beta}$. In our experiment, we take it as the baseline solution for (α, β)-core computation.

Table 3.1: Summary of Notations

$G = (U, V, E)$	a bipartite graph with two node sets U and V, an edge set E
$U(G), V(G), E(G)$	node sets U, V and edge set E of G
n, m	number of nodes and edges of G
$\mathsf{deg}(u, G)$	the degree of node u in $U(G) \cup V(G)$
$\mathsf{dmax}_U(G), \mathsf{dmax}_V(G)$	maximum degree of nodes in U, V
$\mathcal{C}_{\alpha,\beta}$	(α, β)-core
$\mathcal{C}_{\alpha,\beta}.\mathcal{U}, \mathcal{C}_{\alpha,\beta}.\mathcal{V}$	two node sets of (α, β)-core
$Q_{\alpha,\beta}$	(α, β)-core query
$\beta_{\max,\alpha}(u)$	the maximum value of β regarding α such that u is in the corresponding $\mathcal{C}_{\alpha,\beta}$
$\alpha_{\max,\beta}(u)$	the maximum value of α regarding β such that u is in the corresponding $\mathcal{C}_{\alpha,\beta}$
$\mathbb{I}$, $\mathbb{I}^U$, $\mathbb{I}^V$	BiCore-Index, BiCore-Index for nodes in $U(G)$, BiCore-Index for nodes in $V(G)$
δ	the maximum value s.t. $\mathcal{C}_{\delta,\delta} \neq \emptyset$

3.3 Space-Efficient Index and Time-Optimal Query Processing

In this section, we organize all the (α, β)-cores into a linear space index structure, through which an (α, β)-core query can be answered in optimal time, i.e, linear time with respect to the result size. In Section 3.3.1, we first introduce a naive index structure, which is based on the fact that when $\alpha(\beta)$ is fixed, (α, β)-core with larger $\beta(\alpha)$ is contained in the one with smaller $\beta(\alpha)$. After analyzing the problems in the naive index structure, we present our linear space index structure, BiCore-Index. In Section 3.3.2, we show that any (α, β)-core query can be answered in optimal time based on BiCore-Index and present the query processing algorithm. At last in Section 3.3.3, we prove that for any bipartite graph G, the space complexity of BiCore-Index can be bpounded by $O(m)$ where m is the number of edges in G.

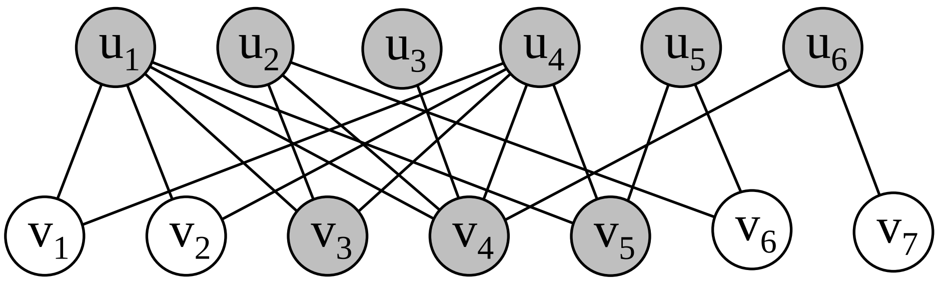


Figure 3.2: A bipartite graph G

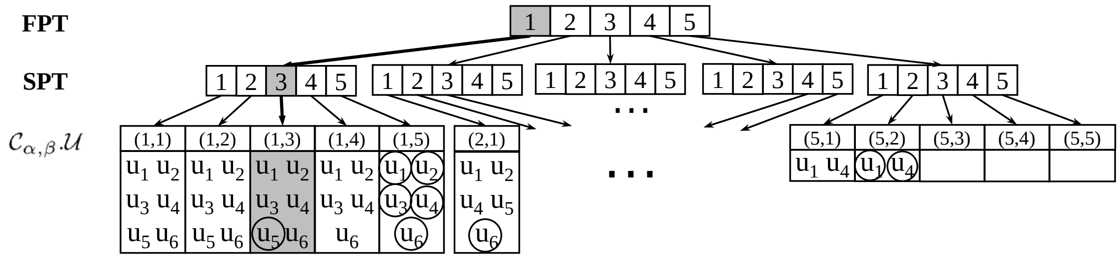


Figure 3.3: Naive Index

3.3.1 BiCore-Index

A Naive Index Structure. To support optimal (α, β)-core query processing, a naive index is as follows: we pre-compute (α, β)-cores for all the possible α and β and store them in the index. Then, for all possible combination of α and β, we record the location of the corresponding (α, β)-core in the index through two level pointer tables. Given a query $Q_{\alpha,\beta}$, we can compute $\mathcal{C}_{\alpha,\beta}$ in optimal time by visiting the nodes stored in the location referred by the (α, β) value. Note that the time cost is optimal since we can find the location referred by the α and β value in $O(1)$ time and output (α, β)-core with time linear to the result size. We show the naive index in the following example.

Example 3.1: Considering the graph G in Figure 3.2, the naive index of G is shown in Figure 3.3. For ease of presentation, we only show the nodes in $U(G)$ in Figure 3.3 and the nodes in $V(G)$ can be indexed similarly. In the index, all the pre-computed (α, β)-cores are stored and shown in the bottom bucket of Figure 3.3. For instance, $(1, 3)$-core is $\{\{u_1, \ldots u_6\}, \{v_3, v_4, v_5\}\}$, thus, $u_1, \ldots, u_6$ are stored in the grey bucket in Figure 3.3. Since both the maximum possible α value (dmax_U) and β value (dmax_V) of G are 5, the first-level pointer table (FPT) contains 5 pointers and each sub-table contains 5 pointers. Suppose the given query is $Q_{1,3}$, we can compute $\mathcal{C}_{1,3}$ by following bold arrows and obtain $\mathcal{C}_{1,3}.\mathcal{U} = \{u_1, \ldots, u_6\}$. $\square$

This naive index can achieve optimal query processing time, however, it requires $O(n^3)$ space. Clearly, it is prohibitive for large graphs. In order to make the index based approach practical, we aim to further reduce the space of the index without sacrificing the optimal query processing property.

Observing the naive index in Figure 3.3, we can find the following two problems exist, which leads to its huge space consumption. The first one is that a node may be stored multiple time in the index. For example, when $\alpha = 1$, u_1 is stored five times in the index, namely, in $\mathcal{C}_{1,1}, \mathcal{C}_{1,2}$, $\mathcal{C}_{1,3}$, $\mathcal{C}_{1,4}$ and $\mathcal{C}_{1,5}$. The same problem also exists on other nodes and other α values. The second one is that empty entries are also kept in the index. For example, there exist no $\mathcal{C}_{5,3}$, $\mathcal{C}_{5,4}$ and $\mathcal{C}_{5,5}$ in G. These entries should be managed to be removed while not affecting the optimal time complexity.

BiCore-Index Structure. We aim to reduce the space consumption of the naive index by addressing the two problems discussed above. Given a bipartite graph G, for a node $u \in U(G)$ and a specific α, if we know the (α, β)-core with maximum β value containing u, we can infer that u is also contained in any (α, β')-Core of G where β' is smaller than the maximum β value. For example, when $\alpha = 1$, since u_1 is contained in $\mathcal{C}_{1,5}$, we know u_5 is also contained in $\mathcal{C}_{1,1}, \mathcal{C}_{1,2}$, $\mathcal{C}_{1,3}$ and $\mathcal{C}_{1,4}$. In other word, storing u_1 at $\mathcal{C}_{1,1}, \mathcal{C}_{1,2}$, $\mathcal{C}_{1,3}$ and $\mathcal{C}_{1,4}$ is redundant regarding (α, β)-core query processing and we only need to store it at $\mathcal{C}_{1,5}$ (marked with circle in Figure 3.3).

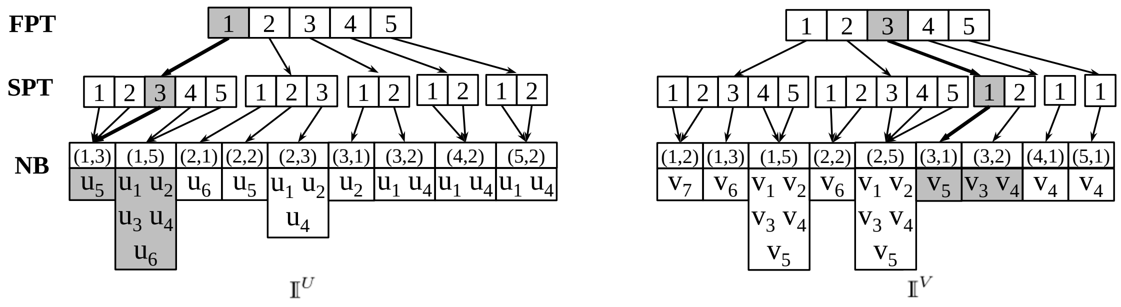


Figure 3.4: BiCore-Index and procedure of QueryOPT for $Q_{1,3}$

Therefore, to address the redundant nodes storage problem in the naive index, for a specific α, we remove the nodes $u \in U(G)$ from the (α, β)-cores that contains u but does not have the maximum β value.

For the empty entry problem, besides the existing empty entries in the index, the node removal procedure introduced above leads to new empty entries. For example, in Figure 3.3, after the node removal, $\mathcal{C}_{1,1}.\mathcal{U}$ is empty. To address this problem, we can remove the empty entry for (α, β)-core from the index and adjust the pointer in SPT pointing to (α, β)-core to point to the first (α, β')-core such that it is not empty and $\beta' > \beta$. To compute $C_{\alpha,\beta}.\mathcal{U}$, we follow the pointer in β-th element of the α-th sub-table in SPT (assume the pointed (α, β)-core is $\mathcal{C}_{\alpha,\beta'}$) and collect remaining nodes from $C_{\alpha,\beta'}$ to $C_{\alpha,\beta''}$, where $C_{\alpha,\beta''}$ is the last non-empty (α, β)-core after the node removal regarding α. The above analysis also holds for any node $v \in V(G)$, a specific β and the naive index structure for nodes in $V(G)$.

Following the above idea, we give the formal definition of our index. Before that, to character the (α, β)-core with the maximum β (α) value that contains a node regarding a specific α (β), we define:

Definition 3.1:

1. $\beta_{\max,\alpha}(u)$. Given a bipartite graph G and an integer α, for each node $u \in U(G) \cup V(G)$, $\beta_{\max,\alpha}(u)$ is the maximum value of β such that u is contained

in the corresponding $\mathcal{C}_{\alpha,\beta}$. If no such β, $\beta_{\max,\alpha}(u) = 0$.

2. $\alpha_{\max,\beta}(u)$**.** Given a bipartite graph G and an integer β, for each node $u \in U(G) \cup V(G)$, $\alpha_{\max,\beta}(u)$ is the maximum value of α such that u is contained in the corresponding $\mathcal{C}_{\alpha,\beta}$. If no such α, $\alpha_{\max,\beta}(u) = 0$.

Our index, BiCore-Index, denoted by $\mathbb{I}$, is a three-level tree structure with two parts for nodes in $U(G)$ and $V(G)$ respectively, denoted by $\mathbb{I}^U$and $\mathbb{I}^V$. As $\mathbb{I}^U$is symmetrical to $\mathbb{I}^V$, we focus on $\mathbb{I}^U$here.

- *Node Blocks* (NB). The third level of $\mathbb{I}^U$, named the node blocks, is a double linked list. Each block in the list is associated with a (α, β) value and contains the nodes $u \in U(G)$ with $\beta_{\max,\alpha}(u) = \beta$.
- *First-level Pointer Table* (FPT). The first level of $\mathbb{I}^U$is an array with dmax_U elements. Each element contains a pointer to an array in the second level. We use $\mathbb{I}^U[\alpha]$ to represent the α-th element.
- *Second-level Pointer Table* (SPT). The second level of $\mathbb{I}^U$consists of dmax_U arrays (sub-table). The α-th array is pointed by $\mathbb{I}^U[\alpha]$. The length of the α-th array is the maximum β value regarding the node block (α, β) pointed by the α-th array. We use $\mathbb{I}^U[\alpha][\beta]$ to denote the β-th element of the α-th array in SPT. The pointer in $\mathbb{I}^U[\alpha][\beta]$ points to the first node block with associated (α, β') value, where $\beta' \geq \beta$.

Example 3.2: Figure 3.4 shows the BiCore-Index of G. In NB, u_1 is in node block $(1, 5)$ since $\beta_{\max,1}(u_1) = 5$. In FPT, since $\mathsf{dmax}_U = 5$, the array of FPT contains 5 pointers pointing to the corresponding array in SPT. Different from the naive index, the length of the arrays is not unique. For example, the length of the second array is 3. This is because for $\alpha = 2$, the $(2, \beta)$ node block with maximum β value kept in NB is node block $(2, 3)$. The pointer in the 1st element of the 1st array in SPT points to node block $(1, 3)$ as node block $(1, 1)$ and $(1, 2)$ do not exist in NB. *A key point needed to note here is that a node block in $\mathbb{I}^U$ and a node block in $\mathbb{I}^V$ may share the same associated (α, β) value, but their meanings are different.* For example, v_1 is contained in node block $(1, 5)$ in $\mathbb{I}^V$ means v_1 is contained in $(5, 1)$-core while u_1 is contained in node block $(1, 5)$ in $\mathbb{I}^U$ means u_1 is contained in $(1, 5)$-core. □

3.3.2 Optimal Query Processing

With BiCore-Index, for a query $Q_{\alpha,\beta}$, we compute $\mathcal{C}_{\alpha,\beta}$ by retrieving $\mathcal{C}_{\alpha,\beta}.\mathcal{U}$ through $\mathbb{I}^U$ and $\mathcal{C}_{\alpha,\beta}.\mathcal{V}$ through $\mathbb{I}^V$. The algorithm, QueryOPT, is shown in Algorithm 1.

Algorithm. For a given $Q_{\alpha,\beta}$, if the (α, β)-core is empty, QueryOPT immediately returns $\emptyset$ as the result since either $\mathbb{I}^U[\alpha]$ or $\mathbb{I}^U[\alpha][\beta]$ is empty (line 2-3). If the (α, β)-core is not empty, it first retrieves $\mathcal{C}_{\alpha,\beta}.\mathcal{U}$ and computes the node block nb referred by the pointer in $\mathbb{I}^U[\alpha][\beta]$ (line 5). After that, it iteratively processes the node block in $\mathbb{I}^U$.NB until the first element of the associated value of nb is not the given α (line 6 and 10). All the nodes in visited nb are added into $\mathcal{C}_{\alpha,\beta}.\mathcal{U}$ (line 7-9). The nodes in $\mathcal{C}_{\alpha,\beta}.\mathcal{V}$ are retrieved similarly and $\mathcal{C}_{\alpha,\beta}$ is returned at the end (line 12-13).

Example 3.3: Figure 3.4 illustrates the procedure of QueryOPT to compute $\mathcal{C}_{1,3}$. The processing steps are shown in bold arrows and the visited elements are marked in grey. To compute $\mathcal{C}_{1,3}.\mathcal{U}$, QueryOPT follows the pointer kept in the 1st element in $\mathbb{I}^U$.FPT and the 3rd element of the 1st array in $\mathbb{I}^U$.SPT and

Algorithm 1: QueryOPT

```
Input: $\mathbb{I}$ of $G$ and $Q_{\alpha,\beta}$
Output: $\mathcal{C}_{\alpha,\beta}$ of $G$
1  $\mathcal{C}_{\alpha,\beta} \leftarrow \emptyset$;
2  if $\mathbb{I}^U$.FPT.size() $< \alpha$ or $\mathbb{I}^U[\alpha]$.size() $< \beta$ then
3      return $\emptyset$
4  end if
5  nb $\leftarrow$ node block pointed by $\mathbb{I}^U[\alpha][\beta]$;
6  while the first element of the associated value of nb $= \alpha$ do
7      for each $u \in$ nb do
8          $\mathcal{C}_{\alpha,\beta}.\mathcal{U} \leftarrow \mathcal{C}_{\alpha,\beta}.\mathcal{U} \cup u$;
9      end for
10     nb $\leftarrow$ next node block in $\mathbb{I}^U$.NB;
11 end while
12 Compute $\mathcal{C}_{\alpha,\beta}.\mathcal{V}$ similarly;
13 return $\mathcal{C}_{\alpha,\beta}$
```

obtains u_5 in the node block $(1,3)$. It continues to visit node block $(1,5)$ and stops at node block $(2,1)$ since the first element of $(2,1)$ is larger than 1. Thus, $\mathcal{C}_{1,3}.\mathcal{U} = \{u_1, u_2, u_3, u_4, u_5, u_6\}$. Similarly, QueryOPT follows the pointer kept in the 3rd element in $\mathbb{I}^V$.FPT and the 1st element of the 3rd array in $\mathbb{I}^V$.SPT and obtains $\mathcal{C}_{1,3}.\mathcal{V} = \{v_3, v_4, v_5\}$. □

Correctness. Based on Definition 3.1, we know that $C_{\alpha,\beta}.\mathcal{U}$ consists of the nodes $u \in U(G)$ with $\beta_{\max,\alpha}(u) \geq \beta$. Meanwhile, a node $u \in U(G)$ is contained in the node block (α, β) if and only if $\beta_{\max,\alpha}(u) = \beta$. According to pointing strategy used in SPT, by visiting all the node blocks from the one pointed by $\mathbb{I}^U[\alpha][\beta]$ to (α, h)

where h is the largest value such that (α, h)-core $\neq \emptyset$, Algorithm 1 computes $\mathcal{C}_{\alpha,\beta}.\mathcal{U}$ correctly. Similarly, we can prove that Algorithm 1 computes $\mathcal{C}_{\alpha,\beta}.\mathcal{V}$ correctly.

Theorem 3.1: *Given a $Q_{\alpha,\beta}$ posed on a bipartite graph G,* QueryOPT *computes $\mathcal{C}_{\alpha,\beta}$ in $O(|\mathcal{C}_{\alpha,\beta}.\mathcal{U}| + |\mathcal{C}_{\alpha,\beta}.\mathcal{V}|)$, which is optimal.*

Proof: For a given α, each $u \in U(G)$ appears at most once in the node blocks pointed by the elements in α-th array in SPT. Therefore, no duplicate node is added in $\mathcal{C}_{\alpha,\beta}.\mathcal{U}$ in line 8. Similarly, no duplicate node is added in $\mathcal{C}_{\alpha,\beta}.\mathcal{V}$ in line 12. Since all the nodes visited in QueryOPT are exactly the nodes we need to retrieve, QueryOPT computes $\mathcal{C}_{\alpha,\beta}$ in $O(|\mathcal{C}_{\alpha,\beta}.\mathcal{U}| + |\mathcal{C}_{\alpha,\beta}.\mathcal{V}|)$ time, which is optimal as it is linear to the result size. □

3.3.3 Space Complexity of BiCore-Index

In this section, we prove the linear space complexity of BiCore-Index. We first show that the size of SPT can be bounded by $O(m)$ in Lemma 3.1. Then, we prove that the space complexity of BiCore-Index is $O(m)$ in Theorem 3.2.

Lemma 3.1: *Given a bipartite graph G, the space of its* SPT *is bounded by $O(m)$.*

Proof: Let $u_1, u_2, u_3, \ldots, u_{n_U}$ be any given sequence of $u \in U(G)$. Starting from an empty graph with only $V(G)$, we add nodes in $U(G)$ with their incident edges to the graph one by one following the sequence until we finally get G. Suppose that u_i is just added to the graph. As u_i cannot be contained in any (α, β)-core whose $\alpha > \mathsf{deg}(u_i, G)$, u_i only influences the length of the k-th arrays in SPT with $1 \leq k \leq \mathsf{deg}(u_i, G)$. Since the length of the α-th array increases at most one after insertion of u_i, the size of SPT increases at most $\mathsf{deg}(u_i, G)$. Thus, the space of SPT in $\mathbb{I}^U$is bounded by $O(\sum_{u\in U(G)} \mathsf{deg}(u, G)) = O(m)$. Similarly, it can be shown that the space of SPT in $\mathbb{I}^V$ is also bounded by $O(m)$. Therefore, the space of SPT is bounded by $O(m)$. □

Theorem 3.2: *Given a bipartite graph G, the space of its* BiCore-Index *is bounded by $O(m)$.*

Proof: Since both $\mathsf{dmax}_U(G)$ and $\mathsf{dmax}_V(G)$ is smaller than m, the size of FPT can be bounded by $O(m)$. Furthermore, for each node $u \in U(G) \cup V(G)$, the number of node blocks containing u is $\mathsf{deg}(u)$. Hence, the space of NB is $O(\sum_{u \in U(G) \cup V(G)} \mathsf{deg}(u, G)) = O(m)$. According to Lemma 3.1, the space of BiCore-Index can be bounded by $O(m)$. □

3.4 Index Construction Algorithm

In this section, we introduce how to construct BiCore-Index efficiently. Based on the structure of BiCore-Index, if we know $\beta_{\max,\alpha}(u)$ for each node $u \in U(G)$ regarding all possible α and $\alpha_{\max,\beta}(v)$ for each node $v \in V(G)$ regarding all possible β (in consistent with the literature on unipartite graphs, we call the procedure as core decomposition as well), the construction of BiCore-Index is straightforward and can be finished in $O(m)$ time as shown in Section 3.4.3. Therefore, we first present techniques to conduct the core decomposition.

In Section 3.4.1, we first propose a basic solution on computing core decomposition. That is for a fixed $\alpha(\beta)$, we can compute all the (α, β)-core by removing the edges in the entire graph in one pass. After conducting such computation for $\alpha(\beta)$ from 1 to the maximum value, the core decomposition result is obtained. However, the maximum value of $\alpha(\beta)$ equals to the maximum degree and is too large in real graphs. In Section 3.4.2, we propose a computation-sharing algorithm which can obtain the core decomposition result by only iterating α and β from 1 to δ, where

δ is the maximum value such that the corresponding (δ, δ)-core is non-empty. Finally, in Section 3.4.3, we show how to construct BiCore-Index based on the core decomposition result.

3.4.1 A Basic Core Decomposition Algorithm

Inspired by the algorithm in [DLHM17], considering a node $u \in U(G)$ and a specific α, if $u \in \mathcal{C}_{\alpha,\beta}.\mathcal{U}$ and $u \notin \mathcal{C}_{\alpha,\beta+1}.\mathcal{U}$, we know $\beta_{\max,\alpha}(u) = \beta$. Moreover, for a specific α, $\mathcal{C}_{\alpha,\beta+1}$ is contained in $\mathcal{C}_{\alpha,\beta}$. Therefore, for a specific α, if we compute all the possible (α, β)-cores in increasing order of β by iteratively removing nodes in $U(G)$ with degree less than α and nodes in $V(G)$ with degree less than β, we can obtain $\beta_{\max,\alpha}(u)$ for all nodes $u \in U(G)$ regarding the specific α. Following this way, we can compute $\beta_{\max,\alpha}(u)$ for all $u \in U(G)$ by iterating all possible α values of G in a bottom-up manner. $\alpha_{\max,\beta}(v)$ can be computed similarly.

Algorithm 2: BasicDecom

```
Input: $G = (U \cup V, E)$
Output: $\beta_{\max,\alpha}(u)$ for $u \in U(G)$, $\alpha_{\max,\beta}(v)$ for $v \in V(G)$
1 for each $\alpha = 1$ to $\mathsf{dmax}_U$ do
2     compute$\beta_{\max}(G, \alpha)$;
3 end for
4 for each $\beta = 1$ to $\mathsf{dmax}_V$ do
5     compute$\alpha_{\max}(G, \beta)$;
6 end for
```

Algorithm. The basic algorithm, BasicDecom, is shown in Algorithm 2. BasicDecom first computes $\beta_{\max,\alpha}(u)$ for nodes in $u \in U(G)$. Since the maximum possible value of α for all nodes in $U(G)$ is dmax_U, it iterates α between 1 and

Procedure compute$\beta_{\max}$$(G, \alpha)$

```
1  $G' \leftarrow G$;
2  while $\exists u \in U(G') : \mathsf{deg}(u, G') < \alpha$ do
3      remove $u$ and its incident edges from $G'$;
4  end while
5  while $G' \neq \emptyset$ do
6      $\beta \leftarrow \min_{v \in V(G')} \mathsf{deg}(v, G')$;
7      while $\exists v \in V(G') : \mathsf{deg}(v, G') \leq \beta$ do
8          remove $v$ and its incident edges from $G'$;
9          while $\exists u \in U(G') : \mathsf{deg}(u, G') < \alpha$ do
10             $\beta_{\max,\alpha}(u) \leftarrow \beta$;
11             remove $u$ and its incident edges from $G'$;
12         end while
13     end while
14 end while
```

dmax_U and computes $\beta_{\max,\alpha}(u)$ for $u \in U(G)$ regarding the specific α by invoking compute$\beta_{\max}$ (line 1-3). Similarly, $\alpha_{\max,\beta}(v)$ for $v \in V(G)$ are computed in line 4-6.

Procedure compute$\beta_{\max}$ computes $\beta_{\max,\alpha}(u)$ for all the nodes in $u \in U(G)$ for a given α. It first removes the nodes and their incident edges in G' whose degree is less than α (line 2-4). Then, it processes the nodes in $U(G)$ in increasing of β (line 7). Whenever a node v with $\mathsf{deg}(v, G') \leq \beta$ is removed (line 8), if there exists a node u with $\mathsf{deg}(u, G') < \alpha$ in G', we know that $u \in \mathcal{C}_{\alpha,\beta}$ but $u \notin \mathcal{C}_{\alpha,\beta+1}$ (line 11), which means $\beta_{\max,\alpha}(u)$ regarding α is β (line 10). Procedure compute$\alpha_{\max}$ follows a similar framework as compute$\beta_{\max}$ to compute $\alpha_{\max,\beta}(v)$ for $v \in V(G)$ regarding a given β.

Procedure compute$\alpha_{\max}$(G, β)

```
1  $G' \leftarrow G$;
2  while $\exists v \in V(G')$ : deg$(v, G') < \beta$ do
3      remove $v$ and its incident edges from $G'$;
4  end while
5  while $G' \neq \emptyset$ do
6      $\alpha \leftarrow \min_{u \in U(G')}$ deg$(u, G')$;
7      while $\exists u \in U(G')$ : deg$(u, G') \leq \alpha$ do
8          remove $u$ and its incident edges from $G'$;
9          while $\exists v \in V(G')$ : deg$(v, G') < \beta$ do
10             $\alpha_{\max,\beta}(v) \leftarrow \alpha$;
11             remove $v$ and its incident edges from $G'$;
12         end while
13     end while
14 end while
```

Example 4.1: In Figure 3.5, we show the procedure of computing $\beta_{\max,2}(*)$ with compute$\beta_{\max}$ for the toy graph in Figure 3.2. Since $\alpha = 2$, compute$\beta_{\max}$ removes u_3 at line 2-4. After removing u_3, we get G' shown in Figure 3.5 (a), which is a (2,1)-core. We highlight the nodes removed in each iteration of line 5-14 with gray color. In the first iteration, $\beta = 1$ and compute$\beta_{\max}$ removes v_7 and u_6 at line 8 and 10, respectively. And we know that $\beta_{\max,2}(u_6) = 1$. In the second iteration (Figure 3.5 (b)), $\beta = 2$ and compute$\beta_{\max}$ successively removes v_1, v_2, and v_6. After v_6 is removed, it finds that $deg(u_5, G') < 2$ at line 9. Thus, it removes u_5 and sets $\beta_{\max,2}(u_5) = 2$. v_5 is also removed since the degree of v_5 equals to 2 after the removal of u_5. In the third iteration (Figure 3.5 (c)), $\beta = 3$ and all the remaining

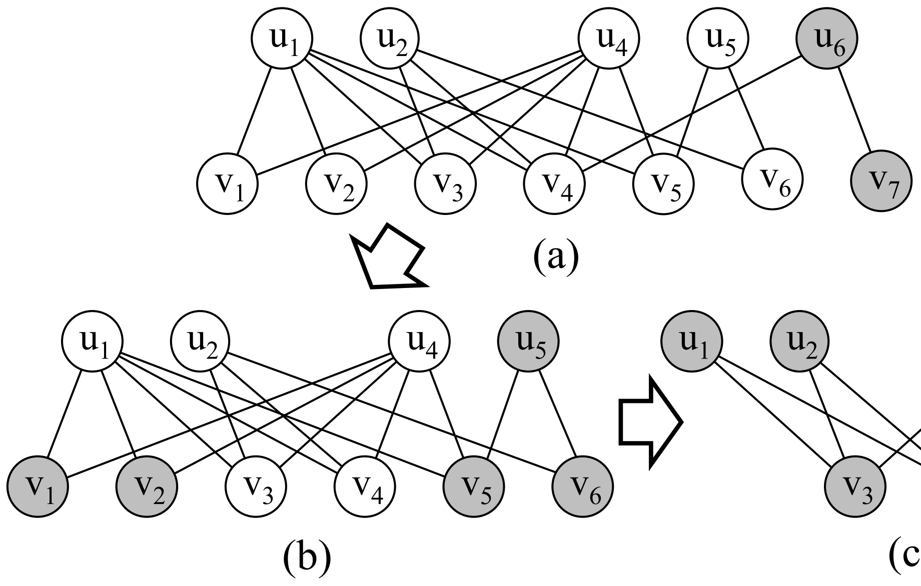


Figure 3.5: The procedure of computing $\beta_{\max,2}(*)$ with compute$\beta_{\max}$. Nodes removed during each iteration are marked in gray.

nodes are removed. $\beta_{\max,2}(u_1)$, $\beta_{\max,2}(u_2)$, and $\beta_{\max,2}(u_4)$ are set as 3. □

Example 4.2: Considering the graph in Figure 3.2, Figure 3.6 shows the procedure of BasicDecom to conduct the core decomposition. Since the decomposition involves all the nodes and large number of values, we only show the procedure for two representative nodes, u_1 and v_4, for brevity. In iteration 1, BasicDecom invokes compute$\beta_{\max}$ with $\alpha = 1$ and finds that u_1 is removed when $\beta = 5$. Thus, it updates $\beta_{\max,1}(u_1)$ as 5. BasicDecom finishes computation in 10 iterations since both dmax_U and dmax_V are 5. □

Correctness. The loop invariant of line 5-14 is that for each $u \in U(G')$, $deg(u, G') \geq \alpha$. To see why, firstly, when G' enters the loop, all the nodes in $U(G')$ with degree less than α are removed at line 2-4. Secondly, whenever there exists some node in $U(G')$ whose degree becomes less than α during the loop, it is removed at line 11. Therefore, the node sets of G' at line 5 always consists of an (α, β)-core where $\beta = \min_{v \in V(G')} \mathsf{deg}(v, G')$. In other words, for $u \in U(G')$, $\beta_{\max,\alpha}(u) \geq \beta$.

Iteration	$\beta_{\max,\alpha}(u_1)$					$\alpha_{\max,\beta}(v_4)$				
	1	2	3	4	5	1	2	3	4	5
1 $(\alpha = 1)$	5	0	0	0	0	0	0	0	0	0
2 $(\alpha = 2)$	5	3	0	0	0	0	0	0	0	0
3 $(\alpha = 3)$	5	3	2	0	0	0	0	0	0	0
4 $(\alpha = 4)$	5	3	2	2	0	0	0	0	0	0
5 $(\alpha = 5)$	5	3	2	2	2	0	0	0	0	0
6 $(\beta = 1)$	5	3	2	2	2	5	0	0	0	0
7 $(\beta = 2)$	5	3	2	2	2	5	5	0	0	0
8 $(\beta = 3)$	5	3	2	2	2	5	5	2	0	0
9 $(\beta = 4)$	5	3	2	2	2	5	5	2	1	0
10$(\beta = 5)$	5	3	2	2	2	5	5	2	1	1

Figure 3.6: Decomposition procedure of Algorithm 2, dark cells are the values updated in each iteration

.

Now we iteratively remove all the nodes $v \in V(G')$ with $deg(v, G') \leq \beta$ (line 7-13). If some node $u \in U(G')$ is removed during this process, we know that $\beta_{\max,\alpha}(u) = \beta$ (line 9). It is notable that the removal of any node in $U(G')$ does not affect the degree of other nodes in $U(G')$ but the nodes in $V(G')$. Similarly, the removal of any node in $V(G')$ will only affect the degree of nodes in $U(G')$. Therefore, we only need to check whether there is some node $u \in U(G')$ whose degree becomes less than α after removing some node v in $V(G')$ (line 8-9). If the degree of $u \in U(G')$ becomes less than α, we set $\beta_{\max,\alpha}(u)$ as β (line 10) and remove it from G' (line 11). When the while loop at line 11 terminates, if $G' \neq \emptyset$, the node sets in G' consists of a new (α, β)-core with larger β, which is the input of the next iteration. Thus, compute$\beta_{\max}$ correctly computes $\beta_{\max,\alpha}(*)$ for each α. Similarly, we know that compute$\alpha_{\max}$ correctly computes $\alpha_{\max,\beta}(*)$ for each β. Since Algorithm 2 invokes compute$\beta_{\max}$ (compute$\alpha_{\max}$) for each α (β) from 1 to dmax_U (dmax_V), Algorithm 2 conducts core decomposition correctly.

Theorem 4.1: *Given a bipartite graph G, Algorithm 2 runs in $O(\mathsf{dmax} \cdot m)$ time,* *where* $\mathsf{dmax} = \max\{\mathsf{dmax}_U, \mathsf{dmax}_V\}$.

Proof: The removal of node v in line 8 and node u in line 3 and 11 can be done in $O(\mathsf{deg}(v, G'))$ and $O(\mathsf{deg}(u, G'))$ time with the efficient data structure proposed in [KBST15]. Since each node is removed once, the time complex of $\mathsf{compute}\beta_{\max}$ is bounded by $O(m)$. Similarly, the running time $\mathsf{compute}\alpha_{\max}$ is also $O(m)$. Thus, the time complexity of $\mathsf{BasicDecom}$ is $O(\mathsf{dmax} \cdot m)$. □

3.4.2 A Computation-sharing Core Decomposition Algorithm

Algorithm 5 processes the nodes in $U(G)$ and $V(G)$ independently and has to conduct $O(\mathsf{dmax})$ iterations to complete the core decomposition. However, dmax could be very large in real graphs [BA99], which makes Algorithm 2 impractical. In this section, we reduce the number of iterations to 2δ by exploring computation-sharing opportunities coherently during processing the nodes in $U(G)$ and $V(G)$, where δ is the maximum value such that $\mathcal{C}_{\delta,\delta}$ is nonempty and is bounded by $\sqrt{m}$.

In Algorithm 2, when finishing the process of a specific α, we actually have computed all the $\mathcal{C}_{\alpha',\beta}$ with $\alpha' \leq \alpha$ in G. Meanwhile, for a node $v \in V(G)$ and a given β, $\alpha_{\max,\beta}(v)$ is the maximum value of α such that v is contained in the corresponding $\mathcal{C}_{\alpha,\beta}$. Therefore, when finishing the process of a specific α in Algorithm 2, we can also obtain $\alpha_{\max,\beta}(v) \leq \alpha$ for $v \in V(G)$ for free. Similarly, we can obtain $\beta_{\max,\alpha}(u) \leq \beta$ for $u \in U(G)$ after processing a specific β. Moreover, let δ be the maximum value such that the corresponding $\mathcal{C}_{\delta,\delta}$ is nonempty, we have:

Lemma 4.1: *Given a bipartite graph G, $\alpha_{\max,\beta}(v) \leq \delta$, for all $\beta > \delta$ and $v \in V(G)$; $\beta_{\max,\alpha}(v) \leq \delta$, for all $\alpha > \delta$ and $u \in U(G)$.*

Proof: Suppose that there exists some $v \in V(G)$ and $\beta > \delta$ such that

$\alpha_{\max,\beta}(v) > \delta$, based on the definition of $\alpha_{\max,\beta}(v)$, $\mathcal{C}_{\delta+1,\delta+1}$ must be nonempty, which contradicts the definition of δ. Thus, $\alpha_{\max,\beta}(v) \leq \delta$, for all $\beta > \delta$ and $v \in V(G)$. Similarly, the second part is correct. □

Based on Lemma 4.1, if we iterate α from 1 to δ in Algorithm 2, besides computing $\beta_{\max,\alpha}(u)$ for each $\alpha \leq \delta$ and each $u \in U(G)$ in line 14, we can actually also obtain $\alpha_{\max,\beta}(v)$ for each $\beta > \delta$ and each $v \in V(G)$. Similarly, if we iterate β from 1 to δ, we can obtain not only $\alpha_{\max,\beta}(v)$ for each $\beta \leq \delta$ and each $v \in V(G)$ but also $\beta_{\max,\alpha}(u)$ for each $\alpha > \delta$ and each $u \in U(G)$.

Algorithm 5: ComShrDecom

```
Input: $G = (U \cup V, E)$
Output: $\beta_{\max,\alpha}(u)$ for $u \in U(G)$, $\alpha_{\max,\beta}(v)$ for $v \in V(G)$
1 $\delta \leftarrow$ the maximum value such that $\mathcal{C}_{\delta,\delta} \neq \emptyset$;
2 for each $\alpha = 1$ to $\delta$ do
3     compute$\beta_{\max}(G, \alpha)$;
4 end for
5 for each $\beta = 1$ to $\delta$ do
6     compute$\alpha_{\max}(G, \beta)$;
7 end for
```

Algorithm. Following above idea, our computation-sharing algorithm, ComShrDecom, is shown in Algorithm 5. In Algorithm 5, ComShrDecom first computes δ of G. δ can be achieved based on its definition by increasing δ step by step starting from 1 while iteratively removing nodes from G whose degree is less than δ. When G is empty, δ is obtained and it can be done in $O(m)$ time. Then, ComShrDecom iterates α and β from 1 to δ to compute $\beta_{\max,\alpha}(u)$ for $u \in U(G)$ and $\alpha_{\max,\beta}(v)$ for $v \in V(G)$ by invoking compute$\beta^{+}_{\max}$ and compute$\alpha^{+}_{\max}$, respectively

Procedure compute$\beta^{+}_{\max}(G, \alpha)$

```
1  $G' \leftarrow G$;
2  while $\exists u \in U(G') : \mathsf{deg}(u, G') < \alpha$ do
3      remove $u$ and its incident edges from $G'$;
4  end while
5  while $G' \neq \emptyset$ do
6      $\beta \leftarrow \min_{v \in V(G')} \mathsf{deg}(v, G')$;
7      while $\exists v \in V(G') : \mathsf{deg}(v, G') \leq \beta$ do
8          remove $v$ and its incident edges from $G'$;
9          for each $i = 1$ to $\beta$ do
10             if $\alpha_{\max,i}(v) < \alpha$ then
11                 $\alpha_{\max,i}(v) \leftarrow \alpha$;
12             end if
13         end for
14         while $\exists u \in U(G') : \mathsf{deg}(u, G') < \alpha$ do
15             $\beta_{\max,\alpha}(u) \leftarrow \beta$;
16             remove $u$ and its incident edges from $G'$;
17         end while
18     end while
19 end while
```

(line 2-7).

The main difference between procedure compute$\alpha^{+}_{\max}$ and procedure compute$\alpha_{\max}$ is that compute$\alpha_{\max}$ updates $\beta_{\max,\alpha}(u)$ and $\alpha_{\max,\beta}(v)$ simultaneously based on the computation result of previous iterations. More specifically, when compute$\beta^{+}_{\max}$ removes a node $v \in V(G)$ and its incident edges from G' (line 8), for

Procedure compute$\alpha^+_{\max}(G, \beta)$

```
1  $G' \leftarrow G$;
2  while $\exists v \in V(G')$ : deg$(v, G') < \beta$ do
3      remove $v$ and its incident edges from $G'$;
4  end while
5  while $G' \neq \emptyset$ do
6      $\alpha \leftarrow \min_{u \in U(G')}$ deg$(u, G')$;
7      while $\exists u \in U(G')$ : deg$(u, G') \leq \alpha$ do
8          remove $u$ and its incident edges from $G'$;
9          for each $i = 1$ to $\alpha$ do
10             if $\beta_{\max,i}(u) < \beta$ then
11                 $\beta_{\max,i}(u) \leftarrow \beta$;
12             end if
13         end for
14         while $\exists v \in V(G')$ : deg$(v, G') < \beta$ do
15             $\alpha_{\max,\beta}(v) \leftarrow \alpha$;
16             remove $v$ and its incident edges from $G'$;
17         end while
18     end while
19 end while
```

each i from 1 to β, if $\alpha_{\max,i}(v) < \alpha$, it updates the corresponding $\alpha_{\max,i}(v)$ as α (line 9-13). This is because when v is removed, v is in a $\mathcal{C}_{\alpha,\beta}$, thus $\alpha_{\max,i}(v)$ is at least α. After compute$\beta^+_{\max}$ finishes, the $\alpha_{\max,\beta}(v) \leq \alpha$ for nodes $v \in V(G)$ are obtained. Procedure compute$\alpha^+_{\max}$ conducts the process symmetrically as compute$\beta^+_{\max}$.

Example 4.3: Figure 3.7 shows the procedure of ComShrDecom to compute

Iteration	$\beta_{\max,\alpha}(u_1)$					$\alpha_{\max,\beta}(v_4)$				
	1	2	3	4	5	1	2	3	4	5
1 ($\alpha = 1$)	5	0	0	0	0	1	1	1	1	1
2 ($\alpha = 2$)	5	3	0	0	0	2	2	2	1	1
3 ($\beta = 1$)	5	3	1	1	1	5	2	2	1	1
4 ($\beta = 2$)	5	3	2	2	2	5	5	2	1	1

Figure 3.7: Decomposition procedure of Algorithm 5. Dark cells are the values updated in each iteration

$\beta_{\max,\alpha}(u_1)$ and $\alpha_{\max,\beta}(v_4)$. ComShrDecom first computes $\delta = 2$, thus it needs 4 iterations to finish the decomposition. Compared with BasicDecom, IterOptDecom updates both $\beta_{\max,\alpha}(u_1)$ and $\alpha_{\max,\beta}(v_4)$ simultaneously in a single iteration. In iteration 2, it invokes compute$\beta^{+}_{\max}$ with $\alpha = 2$ and finds that both u_1 and v_4 are removed when $\beta = 3$. Thus IterOptDecom updates $\beta_{\max,2}(u_1)$ to 3 and $\alpha_{\max,1}(v_4)$, $\alpha_{\max,2}(v_4)$, $\alpha_{\max,3}(v_4)$ to 2. $\square$

Correctness. Following Lemma 4.1, if we iterate α from 1 to δ and invoke compute$\beta^{+}_{\max}$ for the specific α, we can obtain $\beta_{\max,\alpha}(u)$ regarding $\alpha \leq \delta$ for $u \in U(G)$ and $\alpha_{\max,\beta}(v)$ regarding $\beta > \delta$ for $v \in V(G)$. Similarly, by iterating β from 1 to δ and invoking compute$\alpha^{+}_{\max}$ for the specific β, we can obtain $\alpha_{\max,\beta}(v)$ regarding $\beta \leq \delta$ for $v \in V(G)$ and $\beta_{\max,\alpha}(u)$ regarding $\alpha > \delta$ for $u \in U(G)$. Thus, Algorithm 5 conducts the core decomposition correctly.

Theorem 4.2: *Given a bipartite graph G, the time complexity of Algorithm 5 is $O(\delta \cdot m)$, where $\delta \leq \lceil \sqrt{m} \rceil$.*

Proof: The difference between compute$\beta_{\max}$ and compute$\beta^{+}_{\max}$ lies in line 9-13. Since the maximum possible value of β in line 9 can be no larger than $\mathsf{deg}(v, G)$, the time complexity of line 9-13 is $O(\mathsf{deg}(v, G))$. Hence, compute$\beta^{+}_{\max}$ runs in $O(m)$ time. Similarly, compute$\alpha^{+}_{\max}$ also runs in $O(m)$ time. Thus, Algorithm 5 also runs in $O(\delta \cdot m)$.

Let g denote the subgraph induced by $\mathcal{C}_{\delta,\delta}$. Based on the definition of (α, β)-core , there are at least δ nodes in g and the degree of each node is at least δ. Thus, we have $\delta \cdot \delta \leq E(g) \leq m$. Therefore, $\delta \leq \sqrt{m}$. □

Remark. In fact, the number of iterations in Algorithm 5, which equals to $2 \cdot \delta$, is within a constant factor of 2 to the optimal number of iterations we can achieve. This is because essentially during the decomposition process we needs to compute each nonempty (α, β)-core at least once. Hence, we should at least iterate α from 1 to δ or iterate β from 1 to δ to compute all the (α, β)-cores whose $\alpha \leq \delta \wedge \beta \leq \delta$. In other words, the lower bound of the number of iterations required to conduct the decomposition is δ. Therefore, the number of iterations in Algorithm 5 is within a constant factor of 2 to the optimal number of iterations.

3.4.3 Index Construction Algorithm

After obtaining the core decomposition result, we can construct BiCore-Index based on its structure directly. The construction algorithm is shown in Algorithm 8. For $\mathbb{I}^U$, it first constructs $\mathbb{I}^U$.NB (line 2-7) and sorting all the node blocks based on their associated (α, β) value (line 8). After that, the address of the α-th array is stored in $\mathbb{I}^U[\alpha]$ (line 10). $\mathbb{I}^U[\alpha][\beta]$ stores the address of the first node block (α, β') such that $\beta' \geq \beta$ (line 12-14). $\mathbb{I}^V$is constructed symmetrically in line 15 and $\mathbb{I}$ is returned in line 16.

Theorem 4.3: *Given a bipartite graph G, the running time of Algorithm 8 is $O(m)$.*

Proof: Since the size of NB is bounded by $O(m)$, line 4-5 is executed in $O(m)$ times. Furthermore, the running time of line 9 to 14 is bounded by $O(m)$ because the size of SPT is also bounded by $O(m)$ and we only need to visit each element in

Algorithm 8: IndexCon (G)

```
Input: Core decomposition result of $G$
Output: $\mathbb{I}$ of $G$
1  $\mathbb{I} \leftarrow \emptyset$;
2  for each each $u \in U(G)$ do
3      for each $\alpha = 1$ to $\mathsf{deg}(u, G)$ do
4          nb $\leftarrow$ node block associated with $(\alpha, \beta_{\max,\alpha}(u))$;
5          nb $\leftarrow$ nb $\cup\, u$;
6      end for
7  end for
8  sort blocks in $\mathbb{I}^U$.NB based on their associated $(\alpha, \beta)$ value;
9  for each $\alpha = 1$ to $\alpha_{\mathsf{max}}$ do
10     $\mathbb{I}^U[\alpha] \leftarrow$ address of the $\alpha$-th array in SPT;
11 end for
12 for each $\beta = 1$ to $\beta_{\max,\alpha}$ do
13     $\mathbb{I}^U[\alpha][\beta] \leftarrow$ address of nb satisfying SPT in Section 3.3.1;
14 end for
15 construct $\mathbb{I}^V$similarly as line 2-14;
16 return $\mathbb{I}$
```

SPT once in line 13. Similarly, line 15 also runs in $O(m)$. Hence, the running time of Algorithm 8 is $O(m)$. □

3.5 Parallel Algorithms for Index Construction

Our index construction algorithm ComShrDecom can be easily extended to run in parallel. We illustrate the parallel algorithm in Algorithm 9. We first compute

δ similar as ComShrDecom (line 1). Then we dynamically allocate each α and β between 1 and δ to some thread (line 2-6). Specifically, for index construction, we modify line 2 and line 5 of ComShrDecom such that it only computes α or β equal to the allocated value. To avoid race condition, we keep a copy of $\beta_{\max,\alpha}(*, G)$ and $\alpha_{\max,\beta}(*, G)$ for each thread. At last, for index construction, $\beta_{\max,\alpha}(*, G)$ and $\alpha_{\max,\beta}(*, G)$ are set as the largest value computed among all threads.

Algorithm 9: ParallelDecom

```
1 $\delta \leftarrow$ the maximum value such that $\mathcal{C}_{\delta,\delta} \neq \emptyset$;
2 for each $\alpha = 1$ to $\delta$ do
3     dynamically run compute$\beta^{+}_{\max}$ $(G, \alpha)$ in parallel;
4 end for
5 for each $\beta = 1$ to $\delta$ do
6     dynamically run compute$\alpha^{+}_{\max}$ $(G, \beta)$ in parallel;
7 end for
8 merge results by selecting the largest value computed in all threads;
```

3.6 Performance Studies

This section presents our experimental results. All experiments are performed under a Linux operating system on a machine with an Intel Xeon 3.4GHz CPU and 64GB RAM.

Dataset. We evaluate the algorithms on ten real graphs and two synthetic graphs. All the real graphs are downloaded from KONECT[1]. For the synthetic graphs, we generate a power-law graph (PL) in which edges are randomly added such that the

[1]`http://konect.uni-koblenz.de/networks`

degree distribution follows a power-law distribution and a uniform-degree graph (UD) in which all edges are added with the same probability. The details of these graphs are shown in Table 3.2. Note that we remove isolated nodes and duplicate edges in graphs and their sizes listed are based on the processed graphs.

Algorithms. We implement and compare following algorithms:

- (α, β)-core decomposition algorithms.
    - Baseline: the state-of-the-art existing solution proposed in [DLHM17] (introduced in Section 3.2).
    - QueryOPT: Our (α, β)-core query processing algorithm (Algorithm 1).
    - BasicDecom: Our proposed index construction algorithm based on basic decomposition algorithm (Algorithm 2 + Index construction algorithm in Section 3.4.3).
    - ComShrDecom: Our proposed index construction algorithm based on computation-sharing decomposition algorithm (Algorithm 5 + Index construction algorithm in Section 3.4.3).
- Parallel algorithms for (α, β)-core computation.
    - ParallelDecom: Our parallel algorithm for index construction (Algorithm 5 implemented with Algorithm 9).

All algorithms are implemented in C++, using gcc compiler at -O3 optimization level. The time cost is measured as the amount of wall-clock time elapsed during the program's execution. All the experiments are repeated 5 times and we report the average time.

Table 3.2: Statistic for the graphs

Dataset	Type	$\|U\|$	$\|V\|$	$\|E\|$	$\|G\|$(MB)	$\|\mathbb{I}\|$ (MB)	dmax	$\sqrt{m}$	δ
WC (Wikipedia-en)	Text	1.85M	0.18M	3.80M	45.56	30.10	11,593	1,948	19
FG (Flickr)	Social	0.40M	0.10M	8.55M	70.66	70.61	34,989	2,923	148
EP (Epinions)	Rating	0.12M	0.76M	13.67M	113.63	117.27	162,169	3,697	152
DE (Wikipedia-de)	Authorship	0.43M	3.20M	26.01M	231.50	220.13	278,998	5,100	156
RE (Reuters)	Text	0.78M	0.28M	60.57M	481.52	532.30	345,056	7,782	192
TR (TREC)	Text	0.56M	1.17M	83.63M	666.87	748.74	457,437	9,144	509
DUI (Delicious)	Folksonomy	0.83M	33.78M	101.80M	1,065.71	799.81	29,240	10,089	184
LJ (LiveJournal)	Social	3.20M	10.69M	112.31M	985.93	931.60	1,053,676	10,597	109
WT (Web Trackers)	Hyperlink	27.67M	12.76M	140.61M	1,414.34	1,492.43	11,571,952	11,858	438
OG (Orkut)	Affiliation	2.78M	8.73M	327.04M	2,644.93	2,645.46	318,240	18,084	467
PL (Power Law)	Power-law	5M	5M	1,012M	8,080.00	8,003.39	40,354	31,812	374
UD (Uniform Degree)	Uniform-degree	5M	5M	1,067M	8,102.00	8,000.62	277	32,665	169

3.6.1 Performance of Querying Processing

In this section, we evaluate the performance of our proposed (α, β)-core query processing algorithm QueryOPT with the state-of-the-art algorithm Baseline. The running time we report is based on answering the query 10 times. We first test the algorithms on all the twelve datasets with the same query $Q_{10,10}$. Then, we report the performance of the algorithms to process $Q_{\alpha,\beta}$ when varying α (β) regarding fixed β (α).

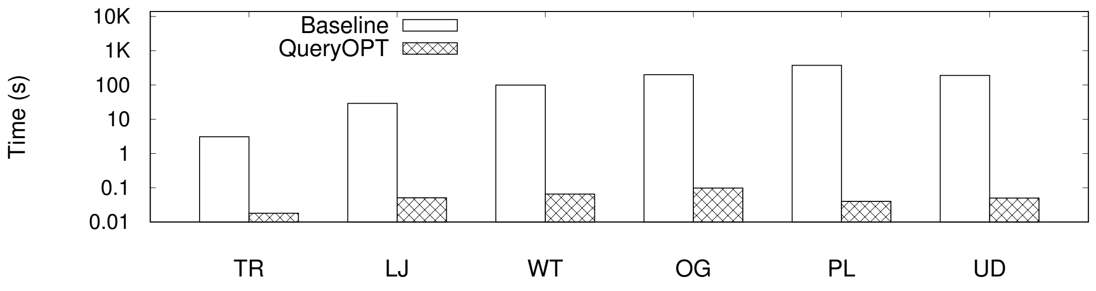
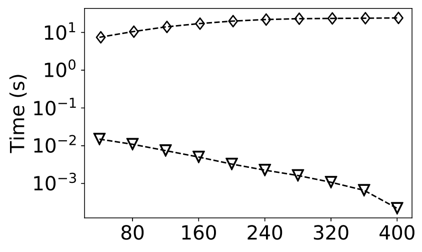


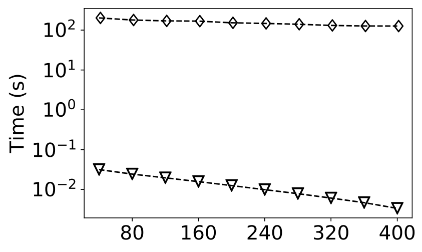
Figure 3.8: Query performance on different datasets

Exp-1: Query performance on different datasets. Figure 3.8 shows the running time of two query processing algorithms to process $Q_{10,10}$. We only show the results on the six largest datasets due to the similar trends. Since QueryOPT is optimal, it is always the fastest algorithm in all cases. For example, on DUI, the running time of QueryOPT is 0.04s, which achieves three order of magnitude improvement compared with Baseline (86.8s).

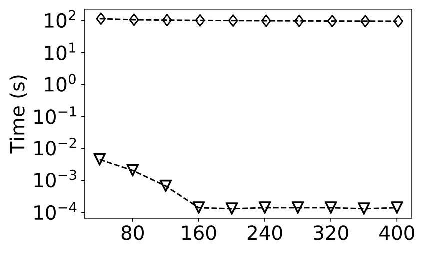
Exp-2: Varying α (β). The running time of Baseline and QueryOPT when varying α (β) is reported in Figure 3.9. We just show the results on four real graphs due to the similar trends. As shown in Figure 3.9, QueryOPT is far more efficient than Baseline on all datasets under every α (outperforms Baseline by 3-7 orders of magnitude). This is because QueryOPT is a time-optimal algorithm. As α grows,



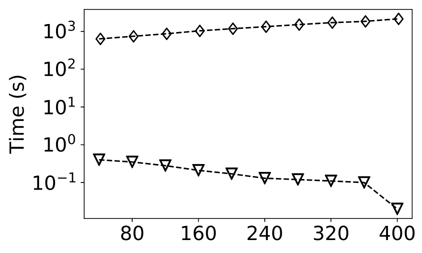
(a) TR ($\beta = 400$)



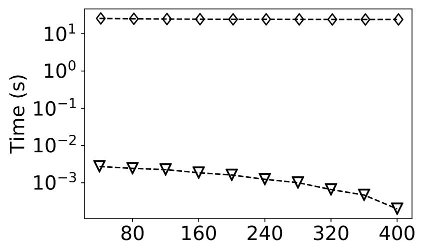
(b) OG ($\beta = 400$)



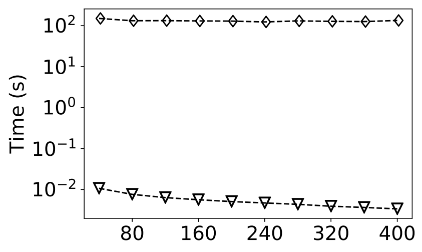
(c) WT ($\beta = 400$)



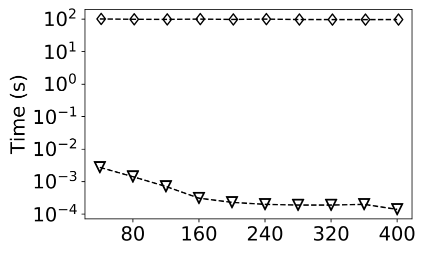
(d) PL ($\beta = 400$)



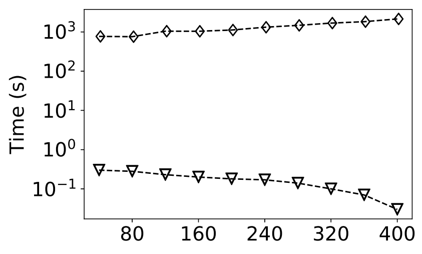
(e) TR ($\alpha = 400$)



(f) OG ($\alpha = 400$)



(g) WT ($\alpha = 400$)



(h) PL ($\alpha = 400$)

Figure 3.9: Query time for different α (β)

the time cost of Baseline is relatively stable since no matter what α is, Baseline needs to visit the entire graph. The gap between QueryOPT and Baseline increases as α grows. This is because as α grows, the size of $\mathcal{C}_{\alpha,\beta}$ decreases and the running time of QueryOPT depends on the size of $\mathcal{C}_{\alpha,\beta}$ while that of Baseline depends on the size of input graph. The results when varying β is similar to varying α.

3.6.2 Performance of Index Construction

In this section, we report the size of BiCore-Index for the datasets and evaluate the performance of two index algorithms BasicDecom and ComShrDecom. In this set of experiments, we set the maximum running time for each test as 48 hours. If a test does not stop within the time limit, we denote its processing time as INF.

Exp-3: Index size $|\mathbb{I}|$. The BiCore-Index size $|\mathbb{I}|$ of all the datasets is reported in Table 3.2. For ease of comparison, we also report the graph size in Table 3.2. As shown in Table 3.2, the size of BiCore-Index is linear to the size of its corresponding graph. For example, the size of OG is $2,644.93$ MB while the size of its BiCore-Index is $2,645.46$ MB. The results are consistent with our theoretical analysis in Section 3.3.3.

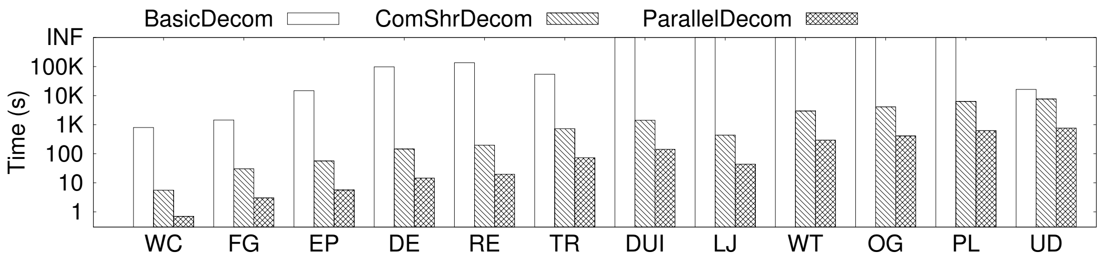
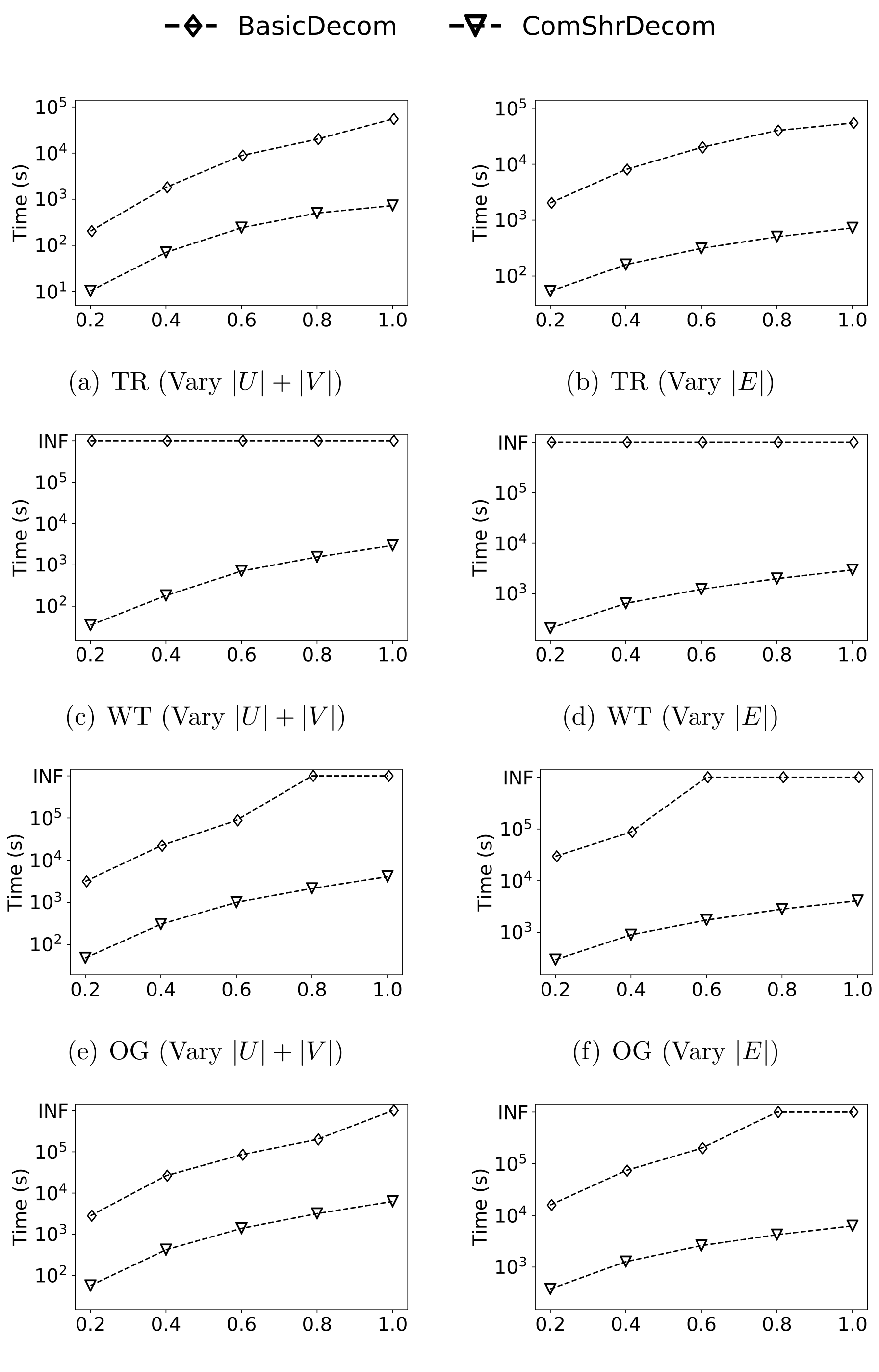


Figure 3.10: Index construction time for different datasets

Exp-4: Index construction time for different datasets. In this experiment, we evaluate the time cost for constructing BiCore-Index on different datasets using BasicDecom and ComShrDecom. The results are reported in Figure 3.10. As shown in Figure 3.10, ComShrDecom is faster than BasicDecom on all datasets and on average achieves over 1000x improvement. For example, in EP, ComShrDecom spends 56 seconds while BasicDecom spends 14,818 seconds.

Exp-5: Comparison of dmax, $\sqrt{m}$ and δ. To better demonstrate the efficiency of BasicDecom and ComShrDecom, we report dmax, $\sqrt{m}$ and δ in Table 3.2 as these values directly relates to their running time. As shown in Table 3.2, δ is at least two order of magnitude smaller than dmax for all datasets, which explains the outstanding performance of ComShrDecom. Furthermore, δ is much smaller than $\sqrt{m}$ on real and power-law graphs, which means ComShrDecom hardly runs in worst case and is very efficient in practice. The results confirm our analysis in Section 3.4 and are consistent with the results in Exp-4.

Exp-6: Scalability of index construction. In this experiment, we evaluate the scalability of BasicDecom and ComShrDecom. To test the scalability, we vary the number of nodes and the number of edges by randomly sampling nodes and edges respectively from 20% to 100% and keeping the induced subgraphs as the input graphs. We only show the results on TR, WT, OG, and PL in Figure 3.11 since trends are similar on all other datasets. As shown in Figure 3.11, when varying the number of nodes, the running time for both algorithms stably increases. ComShrDecom has better performance in all cases and outperforms BasicDecom over two orders of magnitude on average. For example, on WT, the running time of ComShrDecom increases from 35s to 2,953s while BasicDecom cannot terminate within 48 hours for all cases of WT. Varying the number of edges has a similar trend as varying the number of nodes. The results verify that ComShrDecom has a good scalability in practice.



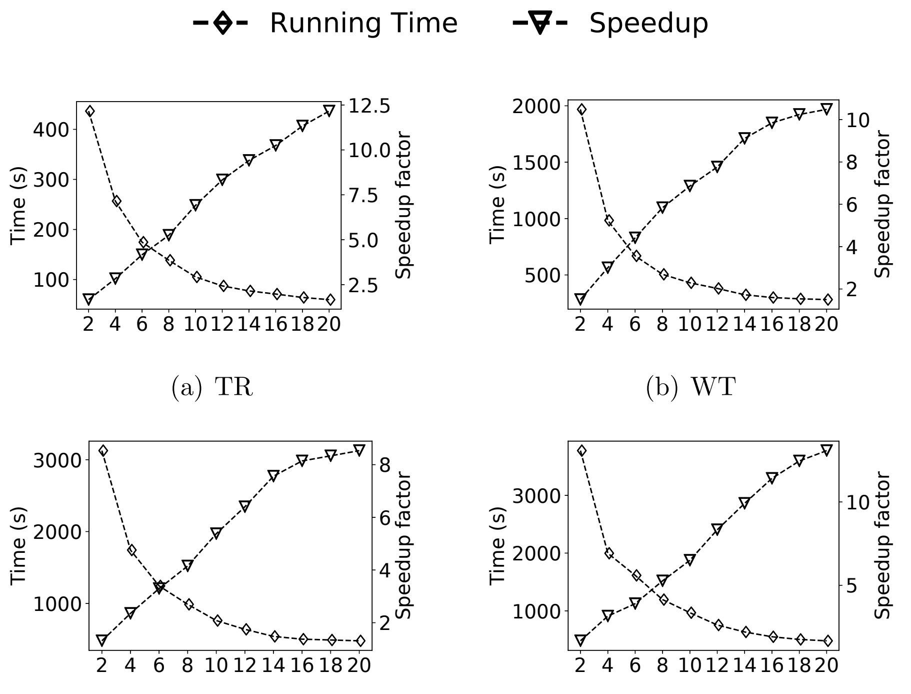
(a) TR (Vary $|U|+|V|$) (b) TR (Vary $|E|$)

(c) WT (Vary $|U|+|V|$) (d) WT (Vary $|E|$)

(e) OG (Vary $|U|+|V|$) (f) OG (Vary $|E|$)

(g) PL (Vary $|U|+|V|$) (h) PL (Vary $|E|$)

Figure 3.11: Scalability of index construction algorithms



(a) TR (b) WT

(c) OG (d) PL

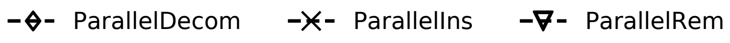
Figure 3.12: ParallelDecom with varying number of cores

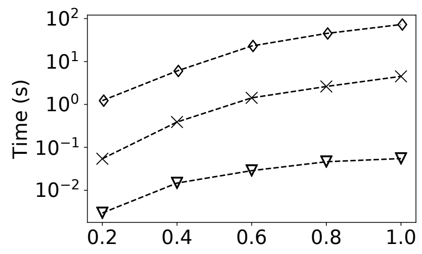
3.6.3 Parallel Index Construction

In this section, we implement parallel index construction and maintenance algorithms ParallelDecom using C++11 thread class and test them with 12 cores in default.

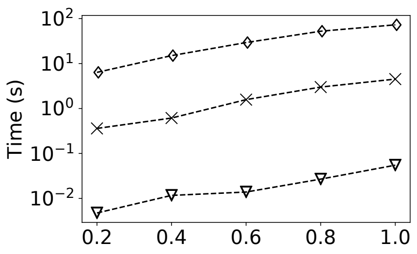
Exp-7: ParallelDecom on different datasets. The running time of ParallelDecom on all datasets is reported in Figure 3.10. ParallelDecom achieves one order magnitude improvement compared with its non-parallel partners, e.g., ComShrDecom. For example, on OG, the running time of ParallelDecom is 732s while ComShrDecom costs 4,103s.

Exp-8: ParallelDecom with varying cores. We report performance of ParallelDecom on TR, WT, OG, and PL with different number of cores in Fig-

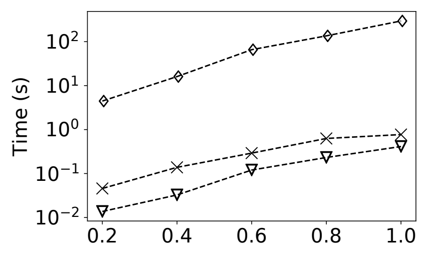




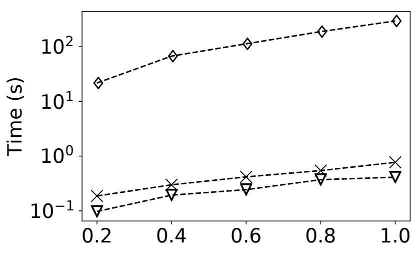
(a) TR (Vary $|U| + |V|$)



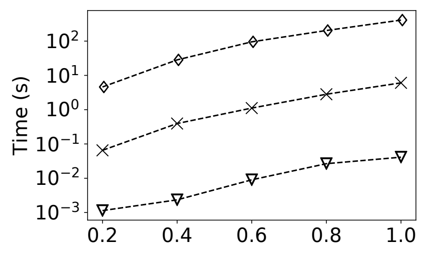
(b) TR (Vary $|E|$)



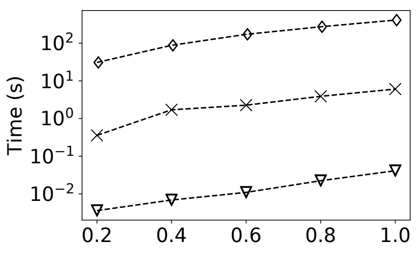
(c) WT (Vary $|U| + |V|$)



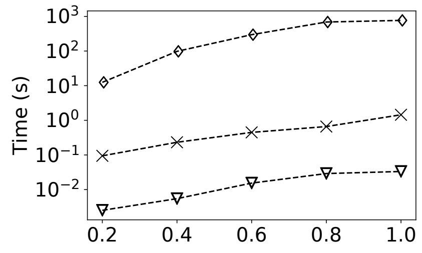
(d) WT (Vary $|E|$)



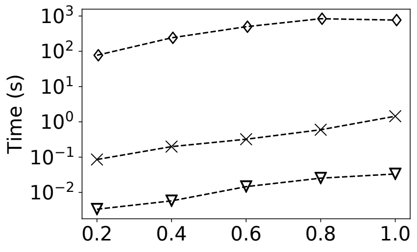
(e) OG (Vary $|U| + |V|$)



(f) OG (Vary $|E|$)



(g) PL (Vary $|U| + |V|$)



(h) PL (Vary $|E|$)

Figure 3.13: Scalability of parallel algorithms

ure 3.12. For ease of comparison, we also draw the speedup factors in each figure. The experiment results show that the running time of ParallelDecom decreases as the number of cores increases. For example, for index construction on OG, the running time reduces from 3000s to 500s as the number of cores increases from 2 to 20. The running time of ParallelDecom is almost inversely proportional to the number of cores, which shows that it is efficient in practice.

Exp-9: Scalability of ParallelDecom. We evaluate the scalability of ParallelDecom on TR, WT, OG, and PL in Figure 3.13. All the graphs are sampled in the same way as Exp-6. As shown in this experiment, the time cost of ParallelDecom increases when varying the number of nodes or edges. Furthermore, the growth trends of ParallelDecom is similar to its non-parallel partners in Exp-6, which verifies that ParallelDecom performs well as the graph size grows.

3.7 Conclusion

In this chapter, we study the problem of efficient (α, β)-core computation. We devise a compact index BiCore-Index whose size can be bounded by $O(m)$. Based on BiCore-Index, we propose an optimal algorithm for (α, β)-core computation and investigate efficient algorithms to construct the index. Moreover, we also discuss about how to construct BiCore-Index with parallel algorithm. The experimental results demonstrate the efficiency of our proposed algorithms.

Chapter 4

(α, β)-Core Maintenance in Bipartite Graphs

4.1 Introduction

Although, BiCore-Index is useful in bipartite graphs for online group recommendation and frustrater detection [LYL+19], in real applications, such as online social networks [KNT10], web graph [OZZ07], and collaboration network [AHL12], graphs are generally dynamic, i.e., the graphs are frequently updated by node/edge insertion/deletion. For example, Facebook has more than 1.3 billion users and approximately 5 new users join Facebook every second [OMK15]; Twitter has more than 300 million users and 3 new users join Twitter every second [OMK15]. Therefore, supporting graph updates efficiently is important for the practical applicability of a graph algorithm in real applications. In the literature, numerous studies on the fundamental graph problems on dynamic graphs have been conducted, such as core maintenance problem in unipartite graphs [SGJS+13, ZYZQ17], reachability [FLL+11], densest subgraphs [ELS15], and pattern matching [ZLWX14].

Motivated by this, we aim to develop efficient BiCore-Index maintenance algorithms in dynamic graphs. Furthermore, as today's graphs grow in scale [LGHB07, DBS18] and current commodity servers are generally equipped with multi-cores [SB13], it is natural to solve graph problems in parallel [DBS18, SRM14]. Therefore, we also investigate the problem of implementing our algorithms in parallel.

Challenges and our solutions. As graphs are frequently updated in many applications, our index should support efficient maintenance when the graph is dynamic. The state-of-art core maintenance algorithms on unipartite graphs (general graphs) require extra neighbor information for each node and auxiliary data structures [ZYZQ17] to maintain an order of nodes. Although the state-of-art core maintenance algorithms only need to use $O(n)$ extra space on general graphs, extending the techniques for general graphs to maintain index in bipartite graphs makes the space cost reach $O(\mathsf{dmax} \cdot n)$ because the containment relationship of (α, β)-core is more complicate than general k-core. Hence, it is a challenge to devise efficient algorithms that can maintain BiCore-Index without extra space cost. Moreover, all the existing k-core maintenance algorithms [LYM13, ZYZQ17, WQZ^{+}16] focus on single-core computation because the insertion/removal of edges spreads influence among vertices in a complicate way and it is hard to predict the change without processing vertices in a certain order. Therefore, it is a challenge to maintain BiCore-Index in a parallel manner. In summary, we need to answer the following two questions:

- How to update BiCore-Index in dynamic graphs efficiently?
- Can we develop effective parallel algorithms for BiCore-Index maintenance?

Regarding the first question, we first propose an algorithm to maintain BiCore-Index in dynamic graphs by reducing unnecessary computation in the procedure of updating BiCore-Index. Then, we show that we can decide whether a node in BiCore-Index should be updated or not by visiting its neighbors locally. Based on this locality property, we further devise a locality-based algorithm that updates BiCore-Index locally. Regarding the second question, we find that the updating process can be split into independent subprocesses which can be executed based on the BiCore-Index before update. To update BiCore-Index, we merge the results by selecting the largest value computed among all subprocess. Moreover, we discuss about how to maintain BiCore-Index when a batch of edges are updated.

Contributions. For core maintenance in bipartite graphs, our main contributions in this chapter are summarized below.

1. *Efficient index maintenance algorithm for dynamic graphs.* We develop a locality-based algorithm to update BiCore-Index, which decide whether a node in BiCore-Index should be updated or not by visiting its neighbors locally. Moreover, we discuss about how to maintain BiCore-Index when a batch of edges are updated.

2. *Efficient parallel maintenance algorithm.* We devise an efficient parallel index maintenance algorithms by splitting the updating process into independent subprocesses and merging the results by selecting the largest value computed among all subprocess.

3. *Extensive experiments on real datasets.* We demonstrate the efficiency of our proposed algorithm with ten real graphs and two synthetic graphs. The experimental results show that our algorithm achieves up to 4 orders of magnitude speedup for index maintenance compared with existing techniques.

Outline. Section 4.2 gives the problem definition and the backgrounds of BiCore-Index. In Section 4.3, we present efficient algorithms to maintain BiCore-Index in dynamic graphs. Section 4.4 discusses how to extend our proposed algorithms when a batch of edges are inserted/removed. Section 4.5 discusses parallel implementation of the BiCore-Index construction and maintenance algorithm. Section 4.6 evaluates our algorithms using extensive experiments and Section 4.7 concludes this chapter.

4.2 Preliminaries

A bipartite graph $G = (U, V, E)$ is a graph consisting of two disjoint sets of nodes U and V such that every edge from $E \subseteq U \times V$ connects one node of U and one node of V. We use $U(G)$ and $V(G)$ to denote the two disjoint node sets of G and $E(G)$ to represent the edge set of G. We denote the number of nodes in $U(G)$ and $V(G)$ as n_U and n_V, the total number of nodes as n and the number of edges in $E(G)$ as m. The degree of a node $u \in U(G) \cup V(G)$, denoted by $\mathsf{deg}(u, G)$, is the number of neighbors of u in G. We also use $\mathsf{dmax}_U(G)$ ($\mathsf{dmax}_V(G)$) to denote the maximum degree among all the nodes in $U(G)$ ($V(G)$), i.e., $\mathsf{dmax}_U(G) = \max\{\mathsf{deg}(u, G) | u \in U(G)\}$ ($\mathsf{dmax}_V(G) = \max\{\mathsf{deg}(v, G) | v \in V(G)\}$). For simplicity, we omit G in the notations if the context is self-evident. For a bipartite graph G and two node sets $U' \subseteq U(G)$ and $V' \subseteq V(G)$, the bipartite subgraph induced by U' and V' is the subgraph G' of G such that $U(G') = U'$, $V(G') = V'$ and $E(G') = E(G) \cap (U' \times V')$.

Definition 2.1: **((α, β)-core)** Given a bipartite graph G and two integers α and β, the (α, β)-core of G, denoted by $\mathcal{C}_{\alpha,\beta}$, consists of two node sets $\mathcal{U} \subseteq U(G)$ and $\mathcal{V} \subseteq V(G)$ such that the bipartite subgraph g induced by $\mathcal{U} \cup \mathcal{V}$ is the maximal subgraph of G in which all the nodes in $\mathcal{U}$ have degree at least α and all the nodes in $\mathcal{V}$ have degree at least β, i.e., $\forall u \in \mathcal{U}, \mathsf{deg}(u, \mathrm{g}) \geq \alpha \wedge \forall v \in \mathcal{V}, \mathsf{deg}(v, \mathrm{g}) \geq \beta$.

Definition 2.2:

Table 4.1: Summary of Notations

$G = (U, V, E)$	a bipartite graph with two node sets U and V, an edge set E
$U(G), V(G), E(G)$	node sets U, V and edge set E of G
n, m	number of nodes and edges of G
$\mathsf{deg}(u, G)$	the degree of node u in $U(G) \cup V(G)$
G^+, G^-	new bipartite graph with the insertion/removal of some edge
(u, v)	an edge incident to node u and v
$\mathcal{C}^+_{\alpha,\beta}$	new (α, β)-core after the insertion of some edge
$\mathcal{C}^-_{\alpha,\beta}$	new (α, β)-core after the removal of some edge
$\beta_{\max,\alpha}(*, G/G^+/G^-)$	all the $\beta_{\max,\alpha}(u)$ values regarding nodes in $U(G/G^+/G^-)$
$\alpha_{\max,\beta}(*, G/G^+/G^-)$	all the $\alpha_{\max,\beta}(v)$ values regarding nodes in $V(G/G^+/G^-)$
δ	the maximum value s.t. $\mathcal{C}_{\delta,\delta} \neq \emptyset$

1. **$\beta_{\max,\alpha}(u)$.** Given a bipartite graph G and an integer α, for each node $u \in U(G) \cup V(G)$, $\beta_{\max,\alpha}(u)$ is the maximum value of β such that u is contained in the corresponding $\mathcal{C}_{\alpha,\beta}$. If no such β, $\beta_{\max,\alpha}(u) = 0$.

2. **$\alpha_{\max,\beta}(u)$.** Given a bipartite graph G and an integer β, for each node $u \in U(G) \cup V(G)$, $\alpha_{\max,\beta}(u)$ is the maximum value of α such that u is contained in the corresponding $\mathcal{C}_{\alpha,\beta}$. If no such α, $\alpha_{\max,\beta}(u) = 0$.

In order to support efficient (α, β)-core queries, we develop BiCore-Index in previous chapter, which is a three-level tree structure with two parts for nodes in $U(G)$ and $V(G)$ respectively, denoted by $\mathbb{I}^U$and $\mathbb{I}^V$. The vertices in BiCore-Index are arranged based on their $\beta_{\max,\alpha}(*)$ and $\alpha_{\max,\beta}(*)$ values. Please refer to Section 3.3 for details.

Problem Statement. In this paper, we study the problem of efficient maintenance of BiCore-Index when the underlying bipartite graphs are dynamically updated.

The notations that will be used in this chapter are summarized in Table 4.1.

4.3 Index Maintenance In Dynamic Graphs

In this section, we introduce the algorithms for maintaining BiCore-Index in dynamic graphs where nodes and edges are inserted or deleted. We mainly focus on the edge insertion and deletion, because node insertion/deletion can be regarded as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated node.

In Section 4.3.1, we first propose basic algorithms for index maintenance. It is based on the fact that for a given $\alpha(\beta)$, after an edge (u, v) is inserted/removed, we only need to recompute those (α, β)-cores whose $\beta(\alpha)$ is no less/larger than the minimum value among $\beta_{\max,\alpha}(u)$ and $\beta_{\max,\alpha}(v)$ ($\alpha_{\max,\beta}(u)$ and $\alpha_{\max,\beta}(v)$). In Section 4.3.2, we improve the basic algorithms in two folds. First, we show that for a given $\alpha(\beta)$ we only need to recompute one (α, β)-core . Second, we study the locality properties of those nodes that will be influenced after an edge being inserted/removed and show that those nodes can be found through a local search.

4.3.1 Basic Algorithms For Index Maintenance

When a bipartite graph G is updated, a straightforward solution to maintain BiCore-Index is reconstructing it from scratch on the updated graph. However, since the graph is typically large and frequently updated, this approach is impractical due to its inefficiency. In this section, we design an incremental algorithm to maintain BiCore-Index in dynamic graphs. For the ease of presentation, we use G^+/G^- to represent the updated graph after edge (u, v) is inserted/removed and $\mathcal{C}_{\alpha,\beta}$, $\mathcal{C}^+_{\alpha,\beta}$, and $\mathcal{C}^-_{\alpha,\beta}$ to denote the $(\alpha,\ \beta)$-core in G, G^+, and G^-, respectively. Without lose of generality, we assume that $u \in U$ and $v \in V$. We also use $\beta_{\max,\alpha}(*, G/G^+/G^-)$ and $\alpha_{\max,\beta}(*, G/G^+/G^-)$ to represent these values for an arbitrary node not specified in G, G^+ and G^-, respectively.

Algorithm 10: BiCore-Index-Ins

```
   Input: $G$, $\mathbb{I}$ and an inserted edge $(u,v)$
   Output: $\mathbb{I}$ of $G^+$
1  $G^+ \leftarrow$insert $(u,v)$ into $G$;
2  $\delta \leftarrow$ the maximum value such that $\mathcal{C}^+_{\delta,\delta} \neq \emptyset$;
3  $\beta_{\max,\alpha}(w,G^+) \leftarrow \beta_{\max,\alpha}(w,G)$ for $w \in \mathcal{U}$;
4  $\alpha_{\max,\beta}(w,G^+) \leftarrow \alpha_{\max,\beta}(w,G)$ for $w \in \mathcal{V}$;
5  for each $\alpha = 1$ to $\delta$ do
6      $\tau_\alpha \leftarrow \min\{\beta_{\max,\alpha}(u,G), \beta_{\max,\alpha}(v,G)\}$;
7      update$\beta_{\max}$Ins $(G^+,\alpha,\tau_\alpha)$;
8  end for
9  for each $\beta = 1$ to $\delta$ do
10     $\tau_\beta \leftarrow \min\{\alpha_{\max,\beta}(v,G), \alpha_{\max,\beta}(u,G)\}$;
11     update$\alpha_{\max}$Ins $(G^+,\beta,\tau_\beta)$;
12 end for
13 IndexCon$(G^+)$ (Algorithm 8);
```

Edge Insertion

Based on the discussion in Section 3.4, the most time-consuming part to construct BiCore-Index is to compute $\beta_{\max,\alpha}(*)$ for each node in U and $\alpha_{\max,\beta}(*)$ for each node in V. Therefore, the key to incrementally maintain BiCore-Index is to identify those nodes whose $\beta_{\max,\alpha}(*)$ or $\alpha_{\max,\beta}(*)$ are the same in G and G^+, and avoid the re-computation of these values for these nodes.

Given an inserted edge (u,v) on G, for an integer α, let τ_α =

Procedure update$\alpha_{\max}$Ins(G^+, α, τ_α)

```
for each $\tau > \tau_\alpha$ do
    $C \leftarrow \mathcal{C}^+_{\alpha,\tau} - \mathcal{C}_{\alpha,\tau}$;
    for each each $w \in C \wedge w \in U$ do
        $\beta_{\max,\alpha}(w, G^+) \leftarrow \tau$;
    end for
    for each $w \in C \wedge w \in V$ do
        for each $i = 1$ to $\tau$ do
            if $\alpha_{\max,i}(w, G^+) < \alpha$ then
                $\alpha_{\max,i}(w, G^+) \leftarrow \alpha$;
            end if
        end for
    end for
end for
```

$\min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$. Since both u and v are contained in $\mathcal{C}_{\alpha,\tau_\alpha}$, $\mathcal{C}_{\alpha,\beta}$ with $\beta \leq \tau_\alpha$ will not change after the insertion of (u, v). Therefore, we have:

Lemma 3.1: *Given an inserted edge (u, v) on G, for an integer α, let $\tau_\alpha = \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$, if $\beta_{\max,\alpha}(*, G) < \tau_\alpha$, then $\beta_{\max,\alpha}(*, G) = \beta_{\max,\alpha}(*, G^+)$; for an integer β, let $\tau_\beta = \min\{\alpha_{\max,\beta}(u, G), \alpha_{\max,\beta}(v, G)\}$, if $\alpha_{\max,\beta}(*, G) < \tau_\beta$, then $\alpha_{\max,\beta}(*, G) = \alpha_{\max,\beta}(*, G^+)$.*

According to Lemma 3.1, $\beta_{\max,\alpha}(*)$ $(\alpha_{\max,\beta}(*))$ may increase only if its value is no less than τ_α (τ_β). Therefore, for a given $\alpha(\beta)$, we only need to re-compute (α, β)-cores whose $\beta > \tau_\alpha(\alpha > \tau_\beta)$. Following this idea, we design BiCore-Index-Ins to handle edge insertion, which is shown in Algorithm 10.

To support the efficient incremental index maintenance, we keep all $\beta_{\max,\alpha}(w, G)$

Procedure update$\beta_{\max}$Ins(G^+, β, τ_β)

```
1  for each $\tau > \tau_\beta$ do
2      $C \leftarrow \mathcal{C}^+_{\tau,\beta} - \mathcal{C}_{\tau,\beta}$;
3      for each each $w \in C \wedge w \in V$ do
4          $\alpha_{\max,\beta}(w, G^+) \leftarrow \tau$;
5      end for
6      for each $w \in C \wedge w \in U$ do
7          for each $i = 1$ to $\tau$ do
8              if $\beta_{\max,i}(w, G^+) < \beta$ then
9                  $\beta_{\max,i}(w, G^+) \leftarrow \beta$;
10             end if
11         end for
12     end for
13 end for
```

for $w \in U(G)$ ($\alpha_{\max,\beta}(w, G)$ for $w \in V(G)$). Note that the total size of these values for all nodes can be bounded by $O(m)$. Thus, the extra space consumption does not affect the space complexity of BiCore-Index. BiCore-Index-Ins follows a similar framework as Algorithm 5. It first inserts edge (u, v) into G and computes δ value of G^+ (line 1-2). Then, for all possible α from 1 to δ, it computes τ_α based on Lemma 3.1 and invokes update$\beta_{\max}$Ins to update $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ for each node w whose $\beta_{\max,\alpha}(w, G) >= \tau_\alpha$ (line 5-8). β is processed similarly (line 9-12). At last, it updates the BiCore-Index based on the updated $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ (line 13). Note that in line 6, $\beta_{\max,\alpha}(v, G)$ is not kept but it can be computed online through kept $\alpha_{\max,\beta}(v, G)$.

Procedure update$\beta_{\max}$Ins follows a similar framework as compute$\beta^+_{\max}$ in Algorithm 5. Since it only concerns the nodes with $\beta_{\max,\alpha}(*, G) >= \tau_\alpha$, it computes $\mathcal{C}^+_{\alpha,\tau}$ for each $\tau > \tau_\alpha$ (line 1-2). Note that $\mathcal{C}_{\alpha,\tau}$ can be retrieved from BiCore-Index. After that, it updates $\beta_{\max,\alpha}(w, G^+)$ to τ for each $w \in U$ which is newly added to $\mathcal{C}^+_{\alpha,\tau}$ (line 3-5). For each newly added node $w \in V$, it updates $\alpha_{\max,i}(w, G^+)$ to α if $\alpha_{\max,i}(w, G^+) < \alpha$ and $i \leq \tau$ (line 6-12). Procedure update$\alpha_{\max}$Ins implements symmetrical procedure as update$\beta_{\max}$Ins.

Theorem 3.1: *When an edge(u, v) is inserted into graph G,* BiCore-Index-Ins *updates* BiCore-Index *correctly.*

Proof: Based on Lemma 3.1 and Lemma 4.1, in line 5-8, BiCore-Index-Ins updates $\beta_{max,\alpha}(u)$ with $\alpha \leq \delta$ and $\alpha_{max,\beta}(v)$ with $\beta > \delta$. In line 9-12, BiCore-Index-Ins updates $\alpha_{max,\beta}(v)$ with $\beta \leq \delta$ and $\beta_{max,\alpha}(u)$ with $\alpha > \delta$. Hence, BiCore-Index-Ins updates BiCore-Index correctly. □

Edge Removal

Following the similar idea for handling edge insertion, for edge removal, we have:

Lemma 3.2: *Given a removed edge (u, v) on G, for an integer α, let $\tau_\alpha = \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$, if $\beta_{\max,\alpha}(*, G) > \tau_\alpha$, then $\beta_{\max,\alpha}(*, G) = \beta_{\max,\alpha}(*, G^-)$; for an integer β, let $\tau_\beta = \min\{\alpha_{\max,\beta}(u, G), \alpha_{\max,\beta}(v, G)\}$, if $\alpha_{\max,\beta}(*, G) > \tau_\beta$, then $\alpha_{\max,\beta}(*, G) = \alpha_{\max,\beta}(*, G^-)$.*

According to Lemma 3.2, $\beta_{\max,\alpha}(*)$ ($\alpha_{\max,\beta}(*)$) may decrease only if its value is no more than τ_α (τ_β). Therefore, for a given $\alpha(\beta)$, we only need to re-compute (α, β)-cores whose $\beta \leq \tau_\alpha (\alpha \leq \tau_\beta)$. Based on this, we design the algorithm, BiCore-Index-Rem, to handle the edge removal case, which is shown in Algorithm 13.

BiCore-Index-Rem follows the same framework as edge insertion case (Algorithm 10) except the procedure update$\beta_{\max}$Rem and update$\alpha_{\max}$Rem.

Algorithm 13: BiCore-Index-Rem

```
   Input: $G$, $\mathbb{I}$ and a removed edge $(u, v)$
   Output: $\mathbb{I}$ of $G^-$
1  $G^- \leftarrow$remove $(u, v)$ from $G$;
2  $\delta \leftarrow$ the maximum value such that $\mathcal{C}^-_{\delta,\delta} \neq \emptyset$;
3  $\beta_{\max,\alpha}(w, G^-) \leftarrow \beta_{\max,\alpha}(w, G)$ for $w \in \mathcal{U}$;
4  $\alpha_{\max,\beta}(w, G^-) \leftarrow \alpha_{\max,\beta}(w, G)$ for $w \in \mathcal{V}$;
5  for each $\alpha = 1$ to $\delta$ do
6      $\tau_\alpha \leftarrow \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$;
7      update$\beta_{\max}$Rem $(G^-, \alpha, \tau_\alpha)$;
8  end for
9  for each $\beta = 1$ to $\delta$ do
10     $\tau_\beta \leftarrow \min\{\alpha_{\max,\beta}(v, G), \alpha_{\max,\beta}(u, G)\}$;
11     update$\beta_{\max}$Rem $(G^-, \beta, \tau_\beta)$;
12 end for
13 IndexCon$(G^-)$ (Algorithm 8);
```

update$\beta_{\max}$Rem iterates τ from τ_α to 1 for the specific α and computes $\mathcal{C}^-_{\alpha,\tau}$ (line 1-2). For each $w \in U$ which is removed from $\mathcal{C}_{\alpha,\tau}$, we set $\beta_{\max,\alpha}(w, G^-)$ as $\tau - 1$ (line 3-5) because w is no longer contained in (α, τ)-core and the largest possible value of $\beta_{\max,\alpha}(w, G^-)$ is $\tau - 1$. Similarly, for each $w \in V$ which is removed from $\mathcal{C}_{\alpha,\tau}$, we set $\alpha_{\max,i}(w, G^-)$ to $\alpha - 1$ if $\alpha_{\max,i}(w) > \alpha - 1$ and $i \geq \tau$ (line 6-12) because the largest possible value of $\alpha_{\max,i}(w, G^-)$ is $\alpha - 1$. Procedure update$\alpha_{\max}$Rem implements similar procedure as update$\beta_{\max}$Rem.

Theorem 3.2: *When an edge (u, v) is removed from G,* BiCore-Index-Rem *updates*

Procedure update$\beta_{\max}$Rem(G^-, α, τ_α)

```
for each $\tau \leq \tau_\alpha$ do
    $C \leftarrow \mathcal{C}_{\alpha,\tau} - \mathcal{C}^-_{\alpha,\tau}$;
    for each $w \in C \wedge w \in U$ do
        $\beta_{\max,\alpha}(w, G^-) \leftarrow \tau - 1$;
    end for
    for each $w \in C \wedge w \in V$ do
        for each $i = \tau$ to deg$(w, G^+)$ do
            if $\alpha_{\max,i}(w, G^-) > \alpha - 1$ then
                $\alpha_{\max,i}(w, G^-) \leftarrow \alpha - 1$;
            end if
        end for
    end for
end for
```

BiCore-Index *correctly.*

Proof: Based on Lemma 3.1 and Lemma 4.1, in line 5-8, BiCore-Index-Rem updates $\beta_{max,\alpha}(u)$ with $\alpha \leq \delta$ and $\alpha_{max,\beta}(v)$ with $\beta > \delta$. In line 9-12, BiCore-Index-Rem updates $\alpha_{max,\beta}(v)$ with $\beta \leq \delta$ and $\beta_{max,\alpha}(u)$ with $\alpha > \delta$. Hence, BiCore-Index-Rem updates BiCore-Index correctly. □

Complexity analysis

For a given $\alpha(\beta)$, we can compute all the (α, β)-core with a continuous range of $\beta(\alpha)$, e.g., $\alpha \geq \tau_\alpha$ or $\alpha \leq \tau_\alpha$ by visiting the entire graph once. Therefore, the computation complexity of both BiCore-Index-Ins and BiCore-Index-Rem are approximately the same as ComShrDecom since they need to visit the entire graph once

Procedure update$\alpha_{\max}$Rem(G^-, β, τ_β)

```
1  for each $\tau \leq \tau_\beta$ do
2      $C \leftarrow \mathcal{C}_{\tau,\beta} - \mathcal{C}^-_{\tau,\beta}$;
3      for each $w \in C \wedge w \in V$ do
4          $\alpha_{\max,\beta}(w, G^-) \leftarrow \tau - 1$;
5      end for
6      for each $w \in C \wedge w \in U$ do
7          for each $i = \tau$ to deg$(w, G^+)$ do
8              if $\beta_{\max,i}(w, G^-) > \beta - 1$ then
9                  $\beta_{\max,i}(w, G^-) \leftarrow \beta - 1$;
10             end if
11         end for
12     end for
13 end for
```

for each α and β from 1 to δ. The efficiency of BiCore-Index-Ins and BiCore-Index-Rem comes from the fact that they update $\beta_{\max,\alpha}(*, G^+/G^-)$ and $\alpha_{\max,\beta}(*, G^+/G^-)$ based on a continuous subrange of $\beta(\alpha)$ for each $\alpha(\beta)$. As shown in our experiment Exp-8, they are typically faster than ComShrDecom which computes BiCore-Index from scratch. The space cost of both algorithms is $O(m)$ because they only require extra space to store $\beta_{\max,\alpha}(*, G)$ for each node in U and $\alpha_{\max,\beta}(*, G)$ for each node in V.

4.3.2 Locality-based Algorithm For Index Maintenance

The shortcoming of both BiCore-Index-Ins and BiCore-Index-Rem is that they always need to re-compute (α, β)-core of the entire graph for each α or β. To overcome this defect, in this section, we propose a locality-based algorithm to handle edge insertion or removal. We first prove that although the insertion of an edge affects many (α, β)-cores, we actually only need to care about one specific $\beta(\alpha)$ for a given $\alpha(\beta)$. Next, we discuss how to update $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ for each $\alpha(\beta)$ by locally visiting a subgraph instead of the entire graph. Note that the state-of-the-art core maintenance algorithm [ZYZQ17] is not suitable for (α, β)-core maintenance due to the extra space cost. We discuss this at the end of this section.

Locality-based Edge Insertion

According to Lemma 3.1, for a given $\alpha(\beta)$, we need to re-compute all the $\mathcal{C}^+_{\alpha,\beta}$ whose $\beta > \tau_\alpha(\alpha > \tau_\beta)$. This is because the insertion of an edge can change more than one (α, β)-core for a given $\alpha(\beta)$.

Example 3.1: For the bipartite graph G shown in Figure 4.1, edge (u_4, v_6) is inserted. We can see that for $\alpha = 1$, both $(1, 4)$-core and $(1, 5)$-core are changed. $(1, 4)$-core has one more node u_4 and $(1, 5)$-core is newly formed. □

Hence, to improve the efficiency of insertion algorithm, an intuitive way is to re-compute as few (α, β)-cores as possible for a given $\alpha(\beta)$. In the following part, we mainly focus on updating $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ for some given α as the case of β can be analyzed similarly.

Suppose that edge (u, v) is inserted into graph G. For some integer α, let $\eta = \max\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$. It is easy to see that we have $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for any $\beta \geq \eta + 2$. The reason is that v is not contained in $\mathcal{C}_{\alpha,\eta+1}$ and if both u and v are contained in $\mathcal{C}^+_{\alpha,\eta+2}$, by deleting edge (u, v) from G^+, v must be contained in $\mathcal{C}_{\alpha,\eta+1}$. Therefore, for a given α, we only need to re-compute these (α, β)-cores which satisfy $\tau_\alpha < \beta \leq \eta + 1$.

Example 3.2: In Figure 4.1, edge (u_4, v_6) is inserted. $\tau_\alpha = 3$ and $\eta = 4$ when $\alpha = 1$

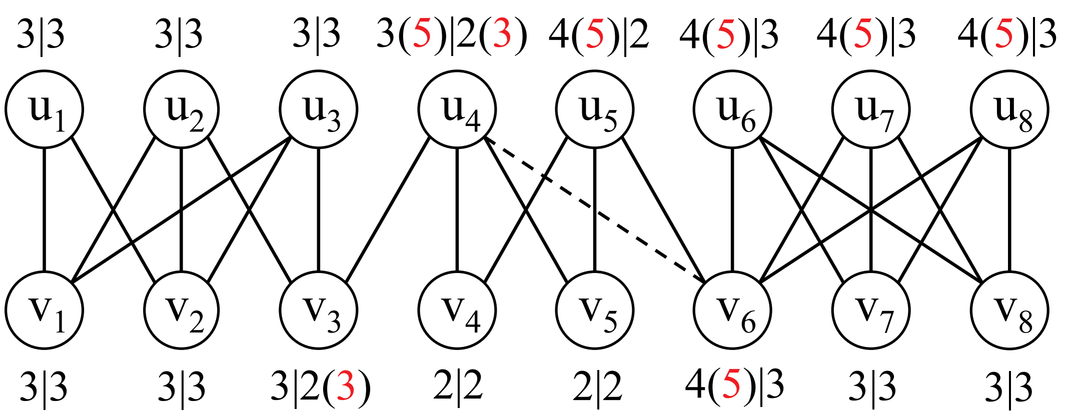


Figure 4.1: Illustration of $\beta_{\max,1}(*)$ (the first value) and $\beta_{\max,2}(*)$ (the second value) before and after the insertion of edge (u_4, v_6). Red number is the value (if changed) after insertion.

because $\beta_{\max,1}(u_4, G) = 3$ and $\beta_{\max,1}(v_6, G) = 4$. Hence, we need to re-compute $(1, 4)$-core and $(1, 5)$-core for $\alpha = 1$. After the computation, we find that u_4 is newly added to $(1, 4)$-core and u_4, u_5, u_6, u_7, u_8, and v_6 are newly added to $(1, 5)$-core. Thus, $\beta_{\max,1}(*)$ of u_4, u_5, u_6, u_7, u_8, and v_6 should be 5 after the insertion of edge (u_4, v_6). □

We have shown that for a given α, the β value of (α, β)-cores which need to be re-computed is between τ_α and $\eta + 1$. However, this range can still be very large. For instance, in Figure 4.1, if the degree of v_6 is 2000, $\beta_{\max,1}(v_6, G)$ is 2000 and the range of β is from 4 to 2001 for $\alpha = 1$. To further reduce computation, we have the following lemma:

Lemma 3.3: *Given an inserted edge (u, v) on G, for any integer α, let $b_\alpha = \max\ x$ s.t. $|\{w | w \in \mathsf{nbr}(u, G^+) \wedge w \in \mathcal{C}_{\alpha,x}\}| \geq \alpha$, then $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} \cup \{u\}$ for each $\beta_{\max,\alpha}(u, G) \leq \beta \leq b_\alpha$; for any integer β, let $b_\beta = \max\ x$ s.t. $|\{w | w \in \mathsf{nbr}(v, G^+) \wedge w \in \mathcal{C}_{x,\beta}\}| \geq \beta$, then $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} \cup \{v\}$ for each $\alpha_{\max,\beta}(v, G) \leq \alpha \leq b_\beta$.*

Proof: We prove the first part as the second part can be proved similarly. Firstly, it should be noted that $b_\alpha \geq \beta_{\max,\alpha}(u, G)$ because u must have at least α neighbors in $(\alpha, \beta_{\max,\alpha}(u, G))$-core. b_α may be larger than $\beta_{\max,\alpha}(u, G)$ because v becomes u's neighbor in G^+. It is easy to see that u must be contained in $\mathcal{C}^+_{\alpha,b_\alpha}$ because u has at least α neighbors in G^+ which are contained in $\mathcal{C}_{\alpha,b_\alpha}$. On the other hand, the existence of edge (u, v) has no influence on other nodes in $\mathcal{C}^+_{\alpha,\beta}$ for $\beta \leq b_\alpha$. The reason is that all the neighbors of u in $\mathcal{C}^+_{\alpha,\beta}$ are already contained in $\mathcal{C}_{\alpha,\beta}$. Hence, we have $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} \cup \{u\}$ for each $\beta_{\max,\alpha}(u, G) \leq \beta \leq b_\alpha$. □

Example 3.3: In Figure 4.1, $\beta_{\max,1}(v_6, G) = 4$. After edge (u_4, v_6) is inserted, we find that $b_1 = 4$ which means that $(1, 4)$-core will only have one more node u_4. □

It is worth noticing that for a given α, u cannot be contained in $\mathcal{C}^+_{\alpha,b_\alpha+2}$. Suppose that u is contained in $\mathcal{C}^+_{\alpha,b_\alpha+2}$. If we remove u from G^+, all the neighbors of u in G^+ will be contained in $(\alpha, b_\alpha + 1)$-core. Hence, u must have at least α neighbors in G^+ which are contained in $\mathcal{C}_{\alpha,b_\alpha+1}$, which means that $b_\alpha = b_\alpha + 1$. Based on this fact, we further induce the following lemma:

Lemma 3.4: *Given an inserted edge (u, v) on G, for any integer α, let $\phi_\alpha = \min\{b_\alpha, \beta_{\max,\alpha}(v, G)\}$, we only need to re-compute $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$; for any integer β, let $\phi_\beta = \min\{\alpha_{\max,\beta}(u, G), b_\beta\}$, we only need to re-compute $\mathcal{C}^+_{\phi_\beta+1,\beta}$.*

Proof: For the first part, we discuss two cases.

Case-1 ($b_\alpha > \beta_{\max,\alpha}(u, G)$): Because u has only one new neighbor v in G^+, we must have $\beta_{\max,\alpha}(u, G) < \beta_{\max,\alpha}(v, G)$, otherwise b_α must be equal to $\beta_{\max,\alpha}(u, G)$. According to the definition of b_α, we have $b_\alpha \leq \beta_{\max,\alpha}(v, G)$. Hence, $\phi_\alpha = b_\alpha$. Because u cannot be contained in $\mathcal{C}^+_{\alpha,b_\alpha+2}$, we know that the insertion of edge (u, v) will not change $\mathcal{C}^+_{\alpha,\beta}$ for $\beta > \phi_\alpha + 1$. Combined with Lemma 3.3, we only need to re-compute $\mathcal{C}_{\alpha,\phi_\alpha+1}$ because for those $\mathcal{C}^+_{\alpha,\beta}$ whose $\beta \neq \phi_\alpha + 1$ they are either unchanged ($\beta \geq \phi_\alpha + 2$ or $\beta \leq \beta_{\max,\alpha}(u, G)$) or contain only one more node u

$(\beta_{\max,\alpha}(u, G) < \beta \leq \phi_\alpha)$.

Case-2 ($b_\alpha = \beta_{\max,\alpha}(u, G)$): Under this case, we have $\phi_\alpha = \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$. If $\phi_\alpha = b_\alpha = \beta_{\max,\alpha}(u, G)$, u cannot be contained in $\mathcal{C}^+_{\alpha,b_\alpha+2}$, we have $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for any $\beta \neq \phi_\alpha + 1$. If $\phi_\alpha = \beta_{\max,\alpha}(v, G)$, v cannot be contained in $\mathcal{C}^+_{\alpha,\phi_\alpha+2}$, otherwise v must be contained in $\mathcal{C}_{\alpha,\phi_\alpha+1}$. Hence, we still have $\mathcal{C}^+_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for any $\beta \neq \phi_\alpha + 1$. Therefore, the first part of Lemma 3.4 is proved. The second part can be proved similarly. □

According to Lemma 3.4, when an edge (u, v) is inserted into G, we only need to re-compute $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ for each α, or symmetrically, $\mathcal{C}^+_{\phi_\beta+1,\beta}$ for each β. To further improve the efficiency, we aim to locally search all the newly added nodes in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$. Let U^* and V^* denote the newly added nodes from U and V, respectively. We have the following lemma:

Lemma 3.5: *Given an inserted edge (u, v) on G, let U^* and V^* denote the newly added nodes in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ from U and V, respectively, the induced subgraph by $U^* \cup V^*$ in G^+ is connected.*

Proof: If a node is newly added into $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$, either the node gains a new neighbor or at least one of its existing neighbors is also newly added into $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$. Applying this recursively, we know that the induced subgraph of $U^* \cup V^*$ in G^+ must be connected. □

Lemma 3.5 suggests that we can search for U^* and V^* in a small local region near the inserted edge. It also implies that if a node is added into $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ it must have enough neighbors which can participate in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ together with it.

Let w be the node in $U^* \cup V^*$ and w' be the neighbor of w. Since a node is contained in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ if it is already contained in $\mathcal{C}_{\alpha,\phi_\alpha+1}$, we know that w' must be w's neighbor in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ if $\beta_{\max,\alpha}(w', G) > \phi_\alpha$. Also, if $\beta_{\max,\alpha}(w', G) < \phi_\alpha$, we know that it cannot be w's neighbor in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$. This is because if u is removed from G^+, the rest nodes in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ must form a (α, ϕ_α)-core. In other words, the rest nodes must be contained in $\mathcal{C}_{\alpha,\phi_\alpha}$. Note that we have one exception, node u. According to Lemma 3.3, $\beta_{\max,\alpha}(u, G)$ may be smaller than ϕ_α but $\beta_{\max,\alpha}(u, G^+)$ may equal to $\phi_\alpha + 1$. This special case can be handled by setting $\beta_{\max,\alpha}(u, G)$ as b_α. The only thing that is difficult to tell is that whether w' is w's neighbor in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ when $\beta_{\max,\alpha}(w', G) = \phi_\alpha$ because w' may be added into $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ together with w. To handle this case, we define the local support of a node w as:

Definition 3.1: Local Support (Insertion) Given an node $w \in U \cup V$ and an inserted edge (u, v), for an integer α, the local support of w is defined as $sup(w) = |\{w' \in \mathsf{nbr}(w, G^+) \mid w' \in \mathcal{C}_{\alpha,\phi_\alpha+1} \vee w' \in U^* \cup V^*\}|$.

It is not hard to observe that a node $w \in \mathcal{C}^+_{\alpha,\phi_\alpha+1}.\mathcal{U}$ if and only if $sup(w) \geq \alpha$, and a node $w \in \mathcal{C}^+_{\alpha,\phi_\alpha+1}.\mathcal{V}$ if and only if $sup(w) \geq \phi_\alpha + 1$. Hence, when $\beta_{\max,\alpha}(w, G) = \phi_\alpha$, we can tell whether w belongs to $U^* \cup V^*$ or not based on its local support. Although the local support and $U^* \cup V^*$ recursively depend on each other, we can actually compute the upper bound of local support and decrease it until the local support reaches its true value. The locality-based algorithm BiCore-Index-Ins* which adopts this strategy is given in Algorithm 16.

The basic idea of BiCore-Index-Ins* is to compute local support for every visited node w in such a way that for each $w' \in \mathsf{nbr}(w, G^+)$ with $\beta_{\max,\alpha}(w', G) = \phi_\alpha$, unless BiCore-Index-Ins* makes sure that w' is not contained in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$, it assumes w' is in $U^* \cup V^*$. When BiCore-Index-Ins* finds a node which cannot be contained in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ but was assumed in $U^* \cup V^*$ before, it will decrease the local support of its neighbors by 1 and conducts a backward search to find more visited nodes that should not have been in $U^* \cup V^*$.

Specifically, for each α from 1 to δ, BiCore-Index-Ins* computes ϕ_α based on Lemma 3.4 (line 3-4) and sets $\beta_{\max,\alpha}(u, G^+)$ and $\beta_{\max,\alpha}(u, G)$ as b_α based on

Algorithm 16: BiCore-Index-Ins*

```
   Input: $G$, $\mathbb{I}$ and an inserted edge $(u, v)$
   Output: $\mathbb{I}$ of $G^+$
1  line 1-4 of Algorithm 10;
2  for each $\alpha = 1$ to $\delta$ do
3      $b_\alpha \leftarrow \max x \; s.t. \; |\{w | w \in \mathsf{nbr}(u, G^+) \wedge w \in \mathcal{C}_{\alpha,x}\}| \geq \alpha$;
4      $\phi_\alpha \leftarrow \min\{b_\alpha, \beta_{\max,\alpha}(v, G)\}$;
5      $\beta_{\max,\alpha}(u, G^+) \leftarrow b_\alpha$; $\beta_{\max,\alpha}(u, G) \leftarrow b_\alpha$;
6      $T \leftarrow \emptyset$; $C \leftarrow \emptyset$; $S \leftarrow$ empty stack;
7      $sup(w) \leftarrow 0$ for each $w \in U \cup V$;
       /* Assuming $\beta_{\max,\alpha}(u, G) \leq \beta_{\max,\alpha}(v, G)$ */
8      $S.push(u)$;
9      while $S \neq \emptyset$ do
10         $w \leftarrow S.pop()$; $S' \leftarrow S$;
11         $T.insert(w)$; $C.insert(w)$;;
12         for each $w' \in \mathsf{nbr}(w, G^+)$ do
13             if $\beta_{\max,\alpha}(w', G) > \phi_\alpha$ or $w' \in C$ then
14                 $sup(w) \leftarrow sup(w) + 1$;
15             end if
16             else if $w' \notin T \wedge \beta_{\max,\alpha}(w', G) = \phi_\alpha$ then
17                 $sup(w) \leftarrow sup(w) + 1$;
18                 $S'.push(w')$;
19             end if
20         end for
21         if $sup(w) \geq \alpha \wedge w \in U$ or $sup(w) \geq \phi_\alpha + 1 \wedge w \in V$ then
22             $S \leftarrow S'$;
23         end if
24         else
25             RemoveCandidates$(w, \alpha, \phi_\alpha, T, C, G^+)$;
26         end if
27     end while
28     line 3-12 of Procedure update$\alpha_{\max}$Ins;
29 end for
30 for each $\beta = 1$ to $\delta$ do
31     line 3-28 by swapping $u$, $\mathcal{U}$, and $\alpha$ with $v$, $\mathcal{V}$, and $\beta$;
32 end for
33 IndexCon$(G^+)$ (Algorithm 8);
```

```
Procedure RemoveCandidates($w, \alpha, \phi_\alpha, T, C, G^+$)
1 $C.remove(w)$;
2 for each $w' \in \mathsf{nbr}(w, G^+) \land w' \in C$ do
3     $sup(w') \leftarrow sup(w') - 1$;
4     if $sup(w') < \alpha \land w' \in U$ or $sup(w') < \phi_\alpha + 1 \land w' \in V$ then
5         RemoveCandidates($w', \phi_\alpha, T, C, G^+$);
6     end if
7 end for
```

Lemma 3.3 (line 5). Then it issues a DFS to locally compute U^* and V^*. During the DFS, only nodes whose $\beta_{\max,\alpha}(w', G) = \phi_\alpha$ will be visited since other nodes are either already in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ ($\beta_{\max,\alpha}(*, G) > \phi_\alpha$) or cannot be in $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$ ($\beta_{\max,\alpha}(*, G) < \phi_\alpha$). BiCore-Index-Ins* uses two sets T and C, and a stack S to record the visited nodes, candidates (nodes that may be added to $\mathcal{C}^+_{\alpha,\phi_\alpha+1}$), and nodes to be visited (line 6). The root node of DFS is selected between u and v depending on whose $\beta_{\max,\alpha}$ is smaller (line 8). For the node w that is currently being visited, BiCore-Index-Ins* uses S' to record all the unvisited neighbors of w which are assumed in candidates (line 10). It first marks w as visited and adds it into candidates (line 11). Then, it computes $sup(w)$ based on w's neighbors. For each $w' \in \mathsf{nbr}(w, G^+)$, if $\beta_{\max,\alpha}(w', G) > \phi_\alpha$ or w' is in candidates, it increases $sup(w)$ by one (line 12-15). If $\beta_{\max,\alpha}(w', G) = \phi_\alpha$ and w' has not been visited yet, BiCore-Index-Ins* increases $sup(w)$ by one and adds w' to S' for future verification (line 16-19). After $sup(w)$ is computed, if $sup(w)$ is large enough to support w as a possible candidate, it will set S as S' to find more candidates (line 21-22). Otherwise, RemoveCandidates is invoked to recursively remove candidates (line 24-26). Finally, it will update $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ as Algorithm 10 does (line

28). β from 1 to δ is processed similarly as α (line 30-32).

RemoveCandidates first removes w from candidates (line 1) and decreases $sup(w')$ by 1 for each $w' \in \mathsf{nbr}(w, G^+)$ and w' in candidates (line 2-3). Then, it will check whether w' should be removed from candidates (line 4). If the answer is yes, it recursively invokes RemoveCandidates to remove w' from candidates (line 4).

Theorem 3.3: *When an edge(u, v) is inserted into graph G,* BiCore-Index-Ins* *updates* BiCore-Index *correctly.*

Proof: For any given α, the induced subgraph of candidate set C in G^+ is connected and either u or v is contained in C if C is nonempty. Also, BiCore-Index-Ins* will check each node w whose $\beta_{\max,\alpha}(w, G) = \phi_\alpha$ if it is a neighbor of some node in C. Therefore, BiCore-Index-Ins* will not miss any possible node in $U^* \cup V^*$. Furthermore, each node w in C satisfies $sup(w) \geq \alpha$ if $w \in U$, or $sup(w) \geq \phi_\alpha + 1$ if $w \in V$. Therefore, $C = U^* \cup V^*$ when the while loop terminates. Combined with the proof of Theorem 3.2, we know that BiCore-Index-Ins* updates BiCore-Index correctly. □

Example 3.4: We explain the procedure of BiCore-Index-Ins* for $\alpha = 2$ when edge (u_4, v_6) is inserted into the graph in Figure 4.1. When $\alpha = 2$, $\phi_2 = 2$ because $b_2 = 2$ and $\beta_{\max,2}(v_6, G) = 3$. Starting from u_4, BiCore-Index-Ins* first adds u_4 to C and T. It computes $sup(u_4) = 4 > \alpha$ and pushes v_3, v_4, v_5 into stack S because v_3, v_4, v_5 are neighbors of u_4 and $\beta_{\max,2}(v_3/v_4/v_5, G) = \phi_2$. Note that $\beta_{\max,2}(v_6, G) = 3 > \phi_2$ thus v_6 is not pushed into S. Then BiCore-Index-Ins* pops v_5 from S and adds v_5 to C and T. It computes $sup(v_5) = 2 < \phi_2 + 1$ because v_5 has only two neighbors u_4, u_5 whose $\beta_{\max,2}(u_4/u_5, G) = \phi_2$. Hence, BiCore-Index-Ins* invokes RemoveCandidates to remove v_5 from C and decrease $sup(u_4)$ by 1. v_4 is processed similarly as v_5. At this moment, $sup(u_4)$ has decreased from 4 to 2 but it is still no less than α and remains in C. Next, BiCore-Index-Ins* pops v_3 from S and adds v_3 to C

and T. It computes $sup(v_3) = 3 \geq \phi_2 + 1$. Because v_3 has only one neighbor u_4 whose $\beta_{\max,2}(u_4, G) = \phi_2$ and u_4 is already in T, $\mathsf{BiCore\text{-}Index\text{-}Ins}^*$ terminates the while loop. Since only u_4 and v_3 remain in C, $\beta_{\max,2}(u_4, G^+)$ and $\beta_{\max,2}(v_3, G^+)$ are updated as 3. In fact, $\mathsf{BiCore\text{-}Index\text{-}Ins}^*$ will not update $\beta_{\max,2}(v_3, G^+)$. Instead based on line 28, it will update $\alpha_{\max,3}(v_3)$ as 2, which is not presented in Figure 4.1 for ease of presentation. □

Locality-based Edge Removal

Following the similar idea for handling edge insertion, for edge removal, we have:

Lemma 3.6: *Given a removed edge (u, v) on G, for any integer α, let $b_\alpha \leftarrow \max\ x\ s.t.\ |\{w | w \in \mathsf{nbr}(u, G^-) \wedge w \in \mathcal{C}_{\alpha,x}\}| \geq \alpha$, if $b_\alpha + 1 < \beta_{\max,\alpha}(u, G)$, then $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} - \{u\}$ for $b_\alpha < \beta < \beta_{\max,\alpha}(u, G)$; for any integer β, let $b_\beta \leftarrow \max\ x\ s.t.\ |\{w | w \in \mathsf{nbr}(u, G^-) \wedge w \in \mathcal{C}_{x,\beta}\}| \geq \beta$, if $b_\beta + 1 < \alpha_{\max,\beta}(v, G)$, then $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} - \{v\}$ for $b_\beta < \alpha < \alpha_{\max,\beta}(v, G)$.*

Proof: If $b_\alpha < \beta_{\max,\alpha}(u, G)$, we have $\beta_{\max,\alpha}(u, G^-) = b_\alpha$ because u has at least α neighbors in G^- which are also contained in $\mathcal{C}^-_{\alpha,b_\alpha}$ and doesn't have more than α neighbors in any (α, β)-core if $\beta > b_\alpha$. Now, suppose that we insert edge (u, v) back into G^-, according to Lemma 3.3 and Lemma 3.4, we have $\mathcal{C}_{\alpha,\beta} = \mathcal{C}^-_{\alpha,\beta} \cup \{u\}$ for each $b_\alpha < \beta < \beta_{\max,\alpha}(u, G)$. Hence, we have $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta} - \{u\}$ for each $b_\alpha < \beta < \beta_{\max,\alpha}(u, G)$. The second part can be proved similarly. □

Note that the precondition for Lemma 3.6 is $\beta_{\max,\alpha}(u, G) \leq \beta_{\max,\alpha}(v, G)$. If $\beta_{\max,\alpha}(u, G) > \beta_{\max,\alpha}(v, G)$, we have $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for $\beta \neq \beta_{\max,\alpha}(v, G)$. The reason is that the removal of edge (u, v) does not affect $\mathcal{C}^-_{\alpha,\beta}$ for any $\beta > \beta_{\max,\alpha}(v, G)$ and v must be contained in $\mathcal{C}^-_{\alpha,\beta}$ for any $\beta < \beta_{\max,\alpha}(v, G)$. Specifically, we have the following lemma:

Lemma 3.7: *Given an removed edge (u, v) on G, for any integer α, let $\tau_\alpha = \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$, we only need to re-compute $\mathcal{C}^-_{\alpha,\tau_\alpha}$; for any integer β, let $\tau_\beta = \min\{\alpha_{\max,\beta}(u, G), \alpha_{\max,\beta}(v, G)\}$, we only need to re-compute $\mathcal{C}^-_{\tau_\beta,\beta}$.*

Proof: Firstly, we have $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for any $\beta > \tau_\alpha$ since either u or v is not contained in $\mathcal{C}_{\alpha,\beta}$ for $\beta > \tau_\alpha$. Secondly, if $\tau_\alpha = \beta_{\max,\alpha}(u, G)$, according to Lemma 3.6, we know that for each $\mathcal{C}^-_{\alpha,\beta}$ whose $\beta < \tau_\alpha$, it is either unchanged ($\beta \leq b_\alpha$) or losing only one node u ($b_\alpha < \beta < \tau_\alpha$). If $\tau_\alpha = \beta_{\max,\alpha}(v, G)$, we have $\mathcal{C}^-_{\alpha,\beta} = \mathcal{C}_{\alpha,\beta}$ for any $\beta \neq \tau_\alpha$. Therefore, for any integer α, we only need to re-compute $\mathcal{C}^-_{\alpha,\tau_\alpha}$. The second part can be proved similarly. □

According to Lemma 3.7, for a given α, we only need to compute $\mathcal{C}^-_{\alpha,\tau_\alpha}$. Let $V^\#$ and $U^\#$ denote the set of nodes from U and V that will be removed from $\mathcal{C}_{\alpha,\tau_\alpha}$ after the deletion of edge (u, v). We define the local support for each node $w \in U \cup V$ as:

Definition 3.2: Local Support (Removal) Given an node $w \in U \cup V$ and an inserted edge (u, v), for an integer α, the local support of w is defined as $sup(w) = |\{w' \in \mathsf{nbr}(w, G^-) \mid w' \in \mathcal{C}_{\alpha,\tau_\alpha} \wedge w' \notin U^\# \cup V^\#\}|$.

According to Definition 3.2, the local support is the number of w's neighbors that will be in $\mathcal{C}^-_{\alpha,\tau_\alpha}$. Therefore, a node $w \in \mathcal{C}^-_{\alpha,\tau_\alpha}.\mathcal{U}$ if and only if $sup(w) \geq \alpha$, and a node $w \in \mathcal{C}^-_{\alpha,\tau_\alpha}.\mathcal{V}$ if and only if $sup(w) \geq \tau_\alpha$. Similarly to the idea in edge insertion, we can compute the upper bound of local support for each node w and decrease it until it is smaller than α or τ_α. The locality-based algorithm for edge removal BiCore-Index-Rem* is given in Algorithm 16.

The basic idea of BiCore-Index-Rem* is that it first assumes each node w whose $\beta_{\max,\alpha}(w, G) = \tau_\alpha$ is not in $U^\# \cup V^\#$. After $sup(w)$ is computed, if w is found to be in $U^\# \cup V^\#$, BiCore-Index-Rem* decreases the local support of its neighbors by 1 and checks whether they should be added to $U^\# \cup V^\#$.

Algorithm 18: BiCore-Index-Rem*

```
Input: $G$, $\mathbb{I}$ and an removed edge $(u, v)$
Output: $\mathbb{I}$ of $G^+$
1  line 1-4 of Algorithm 13;
2  for each $\alpha = 1$ to $\delta$ do
3      $b_\alpha \leftarrow \max x \; s.t. \; |\{w | w \in \mathsf{nbr}(u, G^-) \wedge w \in \mathcal{C}_{\alpha,x}\}| \geq \alpha$;
4      $\tau_\alpha \leftarrow \min\{\beta_{\max,\alpha}(u, G), \beta_{\max,\alpha}(v, G)\}$;
5      $T \leftarrow \emptyset$; $C \leftarrow \emptyset$; $S \leftarrow$ empty stack;
6      $sup(w) \leftarrow 0$ for each $w \in U \cup V$;
7      if $\beta_{\max,\alpha}(u, G) = \tau_\alpha$ then
8          $S.push(u)$; ;
9      end if
10     line 7-8 by replacing $u$ with $v$;
11     while $S \neq \emptyset$ do
12         $w \leftarrow S.pop()$; $S' \leftarrow S$;
13         $T.insert(w)$;
14         for each each $w' \in \mathsf{nbr}(w, G^-)$ do
15             if $\beta_{\max,\alpha}(w', G) \geq \tau_\alpha \wedge w' \notin C$ then
16                 $sup(w) \leftarrow sup(w) + 1$;
17             end if
18         end for
19         if $sup(w) < \alpha \wedge w \in U$ or $sup(w) < \tau_\alpha \wedge w \in V$ then
20             AddCandidates($w, S', \alpha, \tau_\alpha, T, C, G^-$);
21             $S \leftarrow S'$;
22         end if
23     end while
24     line 3-12 of Procedure update$\beta_{\max}$Rem;
25     if $\beta_{\max,\alpha}(u, G^-) > b_\alpha$ then
26         $\beta_{\max,\alpha}(u, G^-) \leftarrow b_\alpha$;
27     end if
28 end for
29 for each $\beta = 1$ to $\delta$ do
30     line 3-26 by swapping $u$, $\mathcal{U}$, and $\alpha$ with $v$, $\mathcal{V}$, and $\beta$;
31 end for
32 IndexCon($G^-$) (Algorithm 8);
```

BiCore-Index-Rem* uses two sets T and C, and a stack S to record the visited nodes, candidates (nodes that are in $U^{\#} \cup V^{\#}$), and nodes to be visited (line 5).

Procedure AddCandidates($w, S', \alpha, \tau_\alpha, T, C, G^-$)

```
C.insert(w);
for each $w' \in \mathsf{nbr}(w, G^-)$ do
    if $w' \notin T \wedge \beta_{\max,\alpha}(w', G) = \tau_\alpha \wedge w' \notin S'$ then
        $S'.push(w')$;
    end if
    else if $w' \in T \wedge w' \notin C$ then
        $sup(w') \leftarrow sup(w') - 1$;
        if $sup(w') < \alpha \wedge w' \in U$ or $sup(w') < \tau_\alpha \wedge w' \in V$ then
            AddCandidates($w', S', \alpha, \tau_\alpha, T, C, G^-$);
        end if
    end if
end for
```

Since u and v may not be connected in G^-, both u and v may be pushed into S (line 7-10). Different from BiCore-Index-Ins*, for node w which is currently being visited, BiCore-Index-Rem* only marks w as visited but does not add w to candidates immediately (line 13) because C only contains nodes which are surely to be in $U^{\#} \cup V^{\#}$. For each neighbor w' of w, $sup(w)$ is increased by 1 if $\beta_{\max,\alpha}(w', G) \geq \tau_\alpha$ and $w' \notin C$ (line 14-16). After $sup(w)$ is computed, if $sup(w) < \alpha$ for $w \in U$ or $sup(w) < \tau_\alpha$ for $w \in V$, w must be in $U^{\#} \cup V^{\#}$ since it does not have enough neighbors in $\mathcal{C}^-_{\alpha,\tau_\alpha}$. Hence, BiCore-Index-Rem* invokes AddCandidates to add w into C and use S' to record possible candidates which need to be further verified (line 19-21). Finally, it will update $\beta_{\max,\alpha}(*, G^-)$ and $\alpha_{\max,\beta}(*, G^-)$ for each node in G^- as BiCore-Index-Rem does (line 24). Note that after all the $\beta_{\max,\alpha}(w, G^-)$ are updated, if $\beta_{\max,\alpha}(u, G^-) > b_\alpha$, it sets $\beta_{\max,\alpha}(u, G^-)$ as b_α based on Lemma 3.6

(line 29-31). β from 1 to δ is processed similarly as α (line 29-30).

AddCandidates first adds w into candidates (line 1). Then for each w's neighbor w' which has not been visited and $\beta_{\max,\alpha}(w', G) = \tau_\alpha$, w' is pushed into S' if w' is not in S' (line 2-4). Otherwise, if w' has been visited but not in candidates, $sup(w')$ is decreased by one (line 6-7). After that, if w' is found to be in candidates, AddCandidates is recursively invoked to add w' into candidates (line 8-8).

Theorem 3.4: *When an edge (u, v) is removed from graph G,* BiCore-Index-Rem* *updates* BiCore-Index *correctly.*

Proof: Each node in C is connected with either u or v in G^-. Also, BiCore-Index-Rem* will check each node w whose $\beta_{\max,\alpha}(w, G) = \tau_\alpha$ if it is a neighbor of some node in C, thus it will not miss any node in $U^\# \cup V^\#$. Furthermore, each node w in C satisfies $sup(w) < \alpha$ if $w \in U$, or $sup(w) < \tau_\alpha$ if $w \in V$. Therefore, $C = U^\# \cup V^\#$ when the while loop terminates. Combined with the proof of Theorem 3.2, BiCore-Index-Rem* updates BiCore-Index correctly. □

Complexity analysis

The time complexity of both BiCore-Index-Ins* and BiCore-Index-Rem* are approximately the same as ComShrDecom since they need to visit the entire graph for each $\alpha(\beta)$ in the worst case. However, both algorithms are efficient in practice because the subgraph they visit during the local search is usually much smaller than the entire graph. As shown in our experiments, they can achieve up to four order of magnitude improvement when compared with BiCore-Index-Ins and BiCore-Index-Rem. The space cost of BiCore-Index-Ins* and BiCore-Index-Rem* are $O(m)$ since there are at most n nodes in the node sets T, C and stack S.

Discussion. The state-of-the-art core maintenance algorithm for edge insertion in general graphs (unipartite graphs) is an order-based approach proposed in

[ZYZQ17]. This approach utilizes the order of nodes removed during core decomposition process, which is call k-order, to perform core maintenance. Each time when an edge is inserted, it reorders nodes such that the new order is still a k-order of the updated graph. It finds the nodes that need to be updated based on the new order. However, for (α, β)-core maintenance, this approach needs to maintain a k-order for each α, that is the order of nodes removed from graphs when α is fixed and β is increased from 1 to the largest possible value. Therefore, the space cost will reach $O(\mathsf{dmax} \cdot n)$. Even if we only iterate α from 1 to δ, the space cost is $O(\delta \cdot n)$, which is still much larger than the graph size in practice (see Table 3.2). Also, considering the hidden constant related to the data structure that supports fast reordering of k-order, the order-based approach is not suitable for (α, β)-core maintenance.

4.4 Batch Update.

In this section, we discuss how to update BiCore-Index when a sequence of edges are inserted/removed.

We first scan the sequence and remove all the operation pairs consisting of insertion then removal (removal then insertion) of the same edge as these operation pairs have no effect on the final result. For the remaining edges (*effective edges*), we rearrange the order such that all the removed edges come after inserted edges. Thus, we can treat batch update as first insert a set of edges then removing another set of edges.

When a set of edges is inserted, for an integer $\alpha(\beta)$, we set $\pi_\alpha(\pi_\beta)$ as the smallest $\beta_{\max,\alpha}(w, G)(\alpha_{\max,\beta}(w, G))$ where w is incident to at least one inserted edge. Since all the (α, β)-cores whose $\beta \le \pi_\alpha(\alpha \le \pi_\beta)$ will not change after the

Algorithm 20: BiCore-Index-Batch

```
Input: $G$, $\mathbb{I}$ and a sequence of edges $S$ to be removed/inserted
Output: $\mathbb{I}$ of $G^*$
1  $I \leftarrow$ the set of effective inserted edges in $S$;
2  $R \leftarrow$ the set of effective removed edges in $S$;
3  $G^* \leftarrow$ insert all the edges in $I$ into $G$;
4  $\delta \leftarrow$ the maximum value such that $\mathcal{C}_{\delta,\delta} \neq \emptyset$ in $G^*$;
5  for each $\alpha = 1$ to $\delta$ do
6      $\pi_\alpha \leftarrow \min\{\beta_{\max,\alpha}(w, G) \mid w$ is incident to $I\}$;
7      line 1-13 of Procedure update$\alpha_{\max}$Ins by replacing $\tau_\alpha$ with $\pi_\alpha$;
8  end for
9  for each $\beta = 1$ to $\delta$ do
10     line 6-7 by swapping $\alpha$ with $\beta$;
11 end for
12 $G^* \leftarrow$ remove $R$ from $G^*$;
13 $\delta \leftarrow$ the maximum value such that $\mathcal{C}_{\delta,\delta} \neq \emptyset$ in $G^*$;
14 for each $\alpha = 1$ to $\delta$ do
15     $\pi_\alpha \leftarrow \max\{\beta_{\max,\alpha}(w, G^*) \mid w$ is incident to $R\}$;
16     line 1-13 of Procedure update$\alpha_{\max}$Rem by replacing $\tau_\alpha$ with $\pi_\alpha$;
17 end for
18 for each $\beta = 1$ to $\delta$ do
19     line 15-16 by swapping $\alpha$ with $\beta$;
20 end for
21 IndexCon($G^*$) (Algorithm 8);
```

insertions, we only need to re-compute (α, β)-cores whose $\beta > \pi_\alpha(\alpha > \pi_\beta)$ and update corresponding values using method in BiCore-Index-Ins. Similarly, when a set of edges is removed, let $\pi_\alpha(\pi_\beta)$ be the largest $\beta_{\max,\alpha}(w, G)(\alpha_{\max,\beta}(w, G))$ where w is incident to at least one removed edge, we only need to re-compute (α, β)-cores whose $\beta \leq \pi_\alpha(\alpha \leq \pi_\beta)$ and update corresponding values using method in BiCore-Index-Rem.

The algorithm BiCore-Index-Batch is given in Algorithm 20. It first extracts effective inserted and removed edges into I and R, respectively (line 1-2). Then, it inserts all the edges from I into G (line 3). For each α from 1 to δ, π_α is set as the smallest value among $\beta_{\max,\alpha}(w, G)$ such that w is incident to at least one edge in I (line 5-6). For each node w newly added to (α, β)-core whose $\beta > \pi_\alpha$, BiCore-Index-Batch updates $\beta_{\max,\alpha}(w, G^*)$ or $\alpha_{\max,\beta}(w, G^*)$ similar as line 1-13 in update$\alpha_{\max}$Ins (line 7). β is processed similarly (line 9-10). After that, it removes all the edges in R from G^* (line 12). It sets π_α as the largest $\beta_{\max,\alpha}(w, G)$ such that w is incident to at least one edge in R, and updates corresponding values for each node similar as line 1-13 of update$\alpha_{\max}$Rem (line 14-16). β is processed similarly (line 18-19). Finally, IndexCon is invoked to reconstruct BiCore-Index (line 20).

4.5 Parallel Algorithms for Index Construction

Our index maintenance algorithms, i.e., BiCore-Index-Ins* and BiCore-Index-Rem* can be easily extended to run in parallel. We illustrate the parallel framework in Algorithm 21. We first compute δ according to the specific algorithm (line 1). Then we dynamically allocate each α and β between 1 and δ to some thread (line 2-6). Specifically, for index maintenance, we modify line 2 and line 30 of BiCore-Index-Ins*, and line 2 and line 29 of BiCore-Index-Rem* such that each time they only compute the allocated value. To avoid race condition, we keep a copy of $\beta_{\max,\alpha}(*, G^+/G^-)$ and $\alpha_{\max,\beta}(*, G^+/G^-)$ for each thread. At last, for edge insertion, $\beta_{\max,\alpha}(*, G^+)$ and $\alpha_{\max,\beta}(*, G^+)$ are set as the largest value computed among all threads. For edge removal, $\beta_{\max,\alpha}(*, G^-)$ and $\alpha_{\max,\beta}(*, G^-)$ are set as the smallest value among the results (line 8).

Algorithm 21: ParallelFramework

```
1 $\delta \leftarrow$ the maximum value such that $\mathcal{C}_{\delta,\delta} \neq \emptyset$;
2 for each $\alpha = 1$ to $\delta$ do
3     dynamically run compute$\beta^+_{\max}$ $(G, \alpha)$ in parallel;
4 end for
5 for each $\beta = 1$ to $\delta$ do
6     dynamically run compute$\alpha^+_{\max}$ $(G, \beta)$ in parallel;
7 end for
8 merge results by selecting the largest value computed in all threads;
```

4.6 Performance Studies

This section presents our experimental results. All experiments are performed under a Linux operating system on a machine with an Intel Xeon 3.4GHz CPU and 64GB RAM.

Dataset. We evaluate the algorithms on ten real graphs and two synthetic graphs. All the real graphs are downloaded from KONECT[1]. For the synthetic graphs, we generate a power-law graph (PL) in which edges are randomly added such that the degree distribution follows a power-law distribution and a uniform-degree graph

[1]`http://konect.uni-koblenz.de/networks`

(UD) in which all edges are added with the same probability. The details of these graphs are shown in Table 3.2 (Section 3.6). Note that we remove isolated nodes and duplicate edges in graphs and their sizes listed are based on the processed graphs.

Algorithms. We implement and compare following algorithms:

- BiCore-Index maintenance algorithms.
    - BiCore-Index-Ins: Our basic algorithm for handling edge insertion (Algorithm 10).
    - BiCore-Index-Rem: Our basic algorithm for handling edge removal (Algorithm 13).
    - BiCore-Index-Ins*: Our locality-based algorithm for handling edge insertion (Algorithm 16).
    - BiCore-Index-Rem*: Our locality-based algorithm for handling edge removal (Algorithm 18).
    - BiCore-Index-Batch: Our algorithm for handling batch update (Algorithm 20).
- Parallel algorithms for BiCore-Index maintenance.
    - ParallelIns: Our parallel algorithm for handling edge insertion (Algorithm 16 implemented with Algorithm 21).
    - ParallelRem: Our parallel algorithm for handling edge removal (Algorithm 18 implemented with Algorithm 21).

All algorithms are implemented in C++, using gcc compiler at -O3 optimization level. The time cost is measured as the amount of wall-clock time elapsed during the program's execution. All the experiments are repeated 5 times and we report the average time.

4.6.1 Dynamic Maintenance

In this section, we test the performance of our index maintenance algorithms. We take the algorithm which invokes ComShrDecom to construct BiCore-Index from scratch for each update as the baseline solution. For the baseline solution, the running time is nearly the same for edge insertion and removal, therefore, we just show one result in the figures. **Exp-1: Index maintenance on different datasets.** For BiCore-Index-Rem and BiCore-Index-Rem*, we randomly remove 5000 distinct existing edges from the graph and report the average processing time for each edge removal. For BiCore-Index-Ins and BiCore-Index-Ins*, we insert the removed edges back into the graph one by one and report the average processing time for each edge insertion. For BiCore-Index-Batch, we randomly generate 5000 edges which are randomly chosen as insertion or removal and we report the average processing time for each edge. All the results are shown in Figure 4.2.

Generally, the average processing time of our proposed algorithms is much smaller than the baseline solution. For example, on WT, our proposed algorithms BiCore-Index-Ins and BiCore-Index-Ins* can handle edge insertion in 297s and 7.8s, respectively, while the baseline solution ComShrDecom requires 2,953s. Also, BiCore-Index-Rem and BiCore-Index-Rem* can handle edge removal on WT in 324s and 4.8s, respectively. This is because our proposed algorithms save lots of unnecessary computation. Compared with BiCore-Index-Ins and BiCore-Index-Rem, BiCore-Index-Ins* and BiCore-Index-Rem* achieve up to 1000x and 10000x speed up, respectively. This is because for each α and β, BiCore-Index-Ins* and BiCore-Index-Rem* only need to visit a local subgraph rooted at the nodes incident to the inserted or removed edge

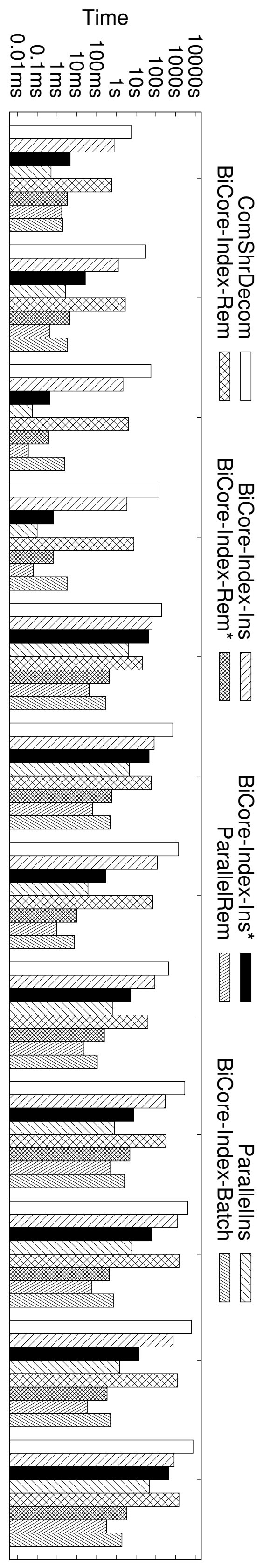


Figure 4.2: Time cost for index maintenance

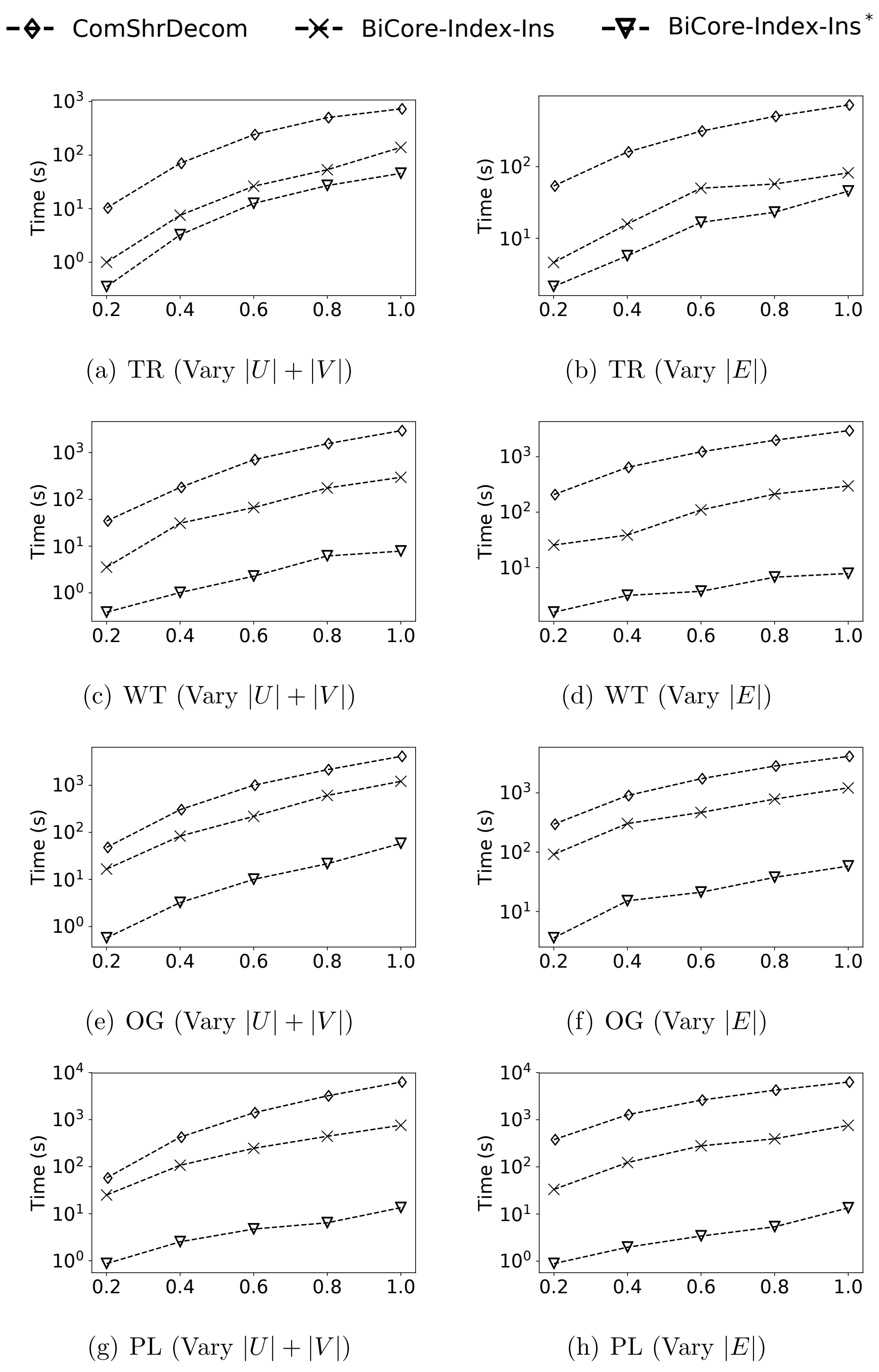
while BiCore-Index-Ins and BiCore-Index-Rem need to visit the entire graph. Moreover, BiCore-Index-Batch can handle batch update on WT and OG in 2.6s and 0.74s for each edge, respectively. Note that the efficiency of BiCore-Index-Batch benefits from the number of edges and it becomes inefficient if there are only a few inserted and removed edges.

BiCore-Index-Ins* doesn't achieve too much improvement compared with BiCore-Index-Ins in RE and TR. This is because the node set U in both bipartite graphs is subject to uniform degree distribution[2] [3] which means nodes in U have similar $\beta_{\max,\alpha}(*)$ value. Thus, BiCore-Index-Ins* needs to visit almost the entire graph before it can compute the nodes that need to be updated. Similar situation also appears in UD where both node sets U and V are subject to the uniform degree distribution.

Exp-2: Scalability of index maintenance. In this experiment, we evaluate the scalability of BiCore-Index-Ins, BiCore-Index-Rem, BiCore-Index-Ins* and BiCore-Index-Ins*. All the graphs are sampled in the same way as Exp-6. For ease of comparison, we also plotted ComShrDecom. We report the performance of BiCore-Index-Ins and BiCore-Index-Ins* in Figure 4.3, and performance of BiCore-Index-Rem and BiCore-Index-Rem* in Figure 4.4. We only show the results on TR, WT, OG, and PL since trends are similar on other datasets. As shown in Figure 4.3 and Figure 4.4, the running time of all four algorithms grows when varying the number of nodes or edges. BiCore-Index-Rem* always performs better than BiCore-Index-Rem in all cases and achieves at least two-order magnitude improvement. BiCore-Index-Ins* outperforms BiCore-Index-Ins by at least one-order magnitude except for TR, which is due to the uniform degree distribution of node set U. Nevertheless, the

[2]`http://konect.uni-koblenz.de/networks/reuters`

[3]`http://konect.uni-koblenz.de/networks/gottron-trec`



(a) TR (Vary $|U| + |V|$)

(b) TR (Vary $|E|$)

(c) WT (Vary $|U| + |V|$)

(d) WT (Vary $|E|$)

(e) OG (Vary $|U| + |V|$)

(f) OG (Vary $|E|$)

(g) PL (Vary $|U| + |V|$)

(h) PL (Vary $|E|$)

Figure 4.3: Scalability of edge insertion algorithms

experiment results show that our proposed index maintenance algorithms have a good scalability in practice.

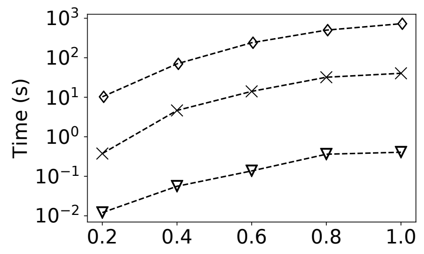
Discussion. As shown in our experiments, the real performance of our maintenance algorithms is from 2 to 4 orders of magnitude faster than the static (α, β)-core decomposition algorithm. Compared with the static method, our algorithm can handle the graph update regarding the (α, β)-core query with as less time as possible and return the query results timely based on the updated graph. Therefore, our dynamic (α, β)-core algorithm is suitable for the online application scenarios requiring to return the query result timely and react to the changes quickly, in which re-running the static (α, β)-core method periodically cannot achieve the same goal.

4.6.2 Parallel Index Maintenance Algorithms

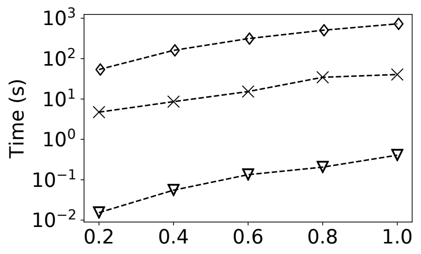
In this section, we implement parallel index construction and maintenance algorithms ParallelIns and ParallelRem using C++11 thread class and test them with 12 cores in default.

Exp-3: Parallel maintenance algorithms on different datasets. The running time of ParallelIns and ParallelRem are reported in Figure 4.2. ParallelIns and ParallelRem achieve one order magnitude improvement compared with their non-parallel partners, e.g., BiCore-Index-Ins* and BiCore-Index-Rem*. For example, ParallelIns and ParallelRem cost 7.07s and 0.068s for index maintenance on OG, respectively, while BiCore-Index-Ins* and BiCore-Index-Rem* cost 58s and 0.44s, respectively.

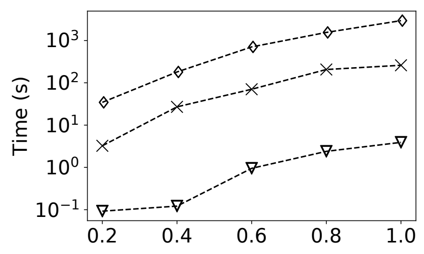
Exp-4: Parallel maintenance algorithms with varying cores. We report performance of ParallelIns, and ParallelRem on TR, WT, OG, and PL with different number of cores in Figure 4.5 and Figure 4.6, respectively. For ease of comparison, we also draw the speedup factors in each figure. The experiment results show



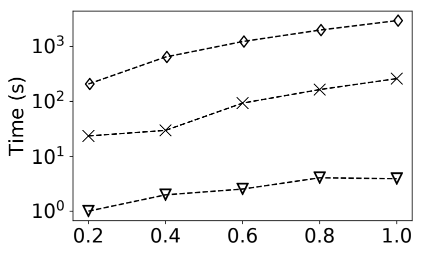
(a) TR (Vary $|U| + |V|$)



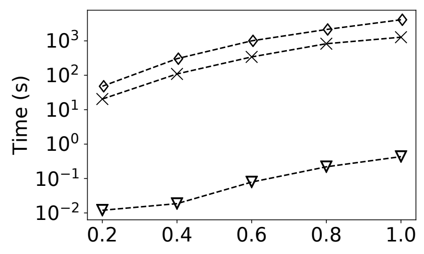
(b) TR (Vary $|E|$)



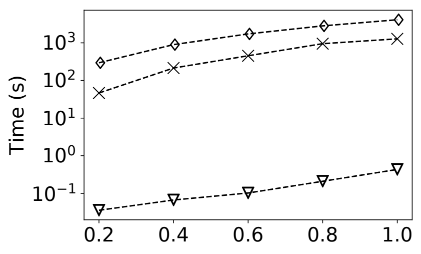
(c) WT (Vary $|U| + |V|$)



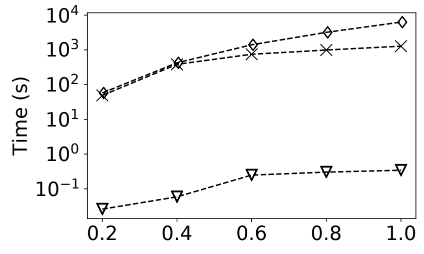
(d) WT (Vary $|E|$)



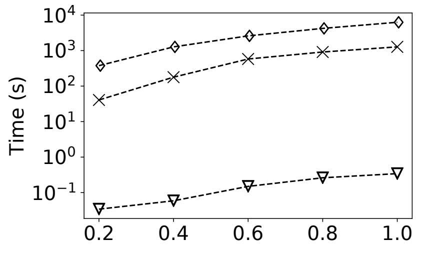
(e) OG (Vary $|U| + |V|$)



(f) OG (Vary $|E|$)



(g) PL (Vary $|U| + |V|$)



(h) PL (Vary $|E|$)

Figure 4.4: Scalability of edge removal algorithms

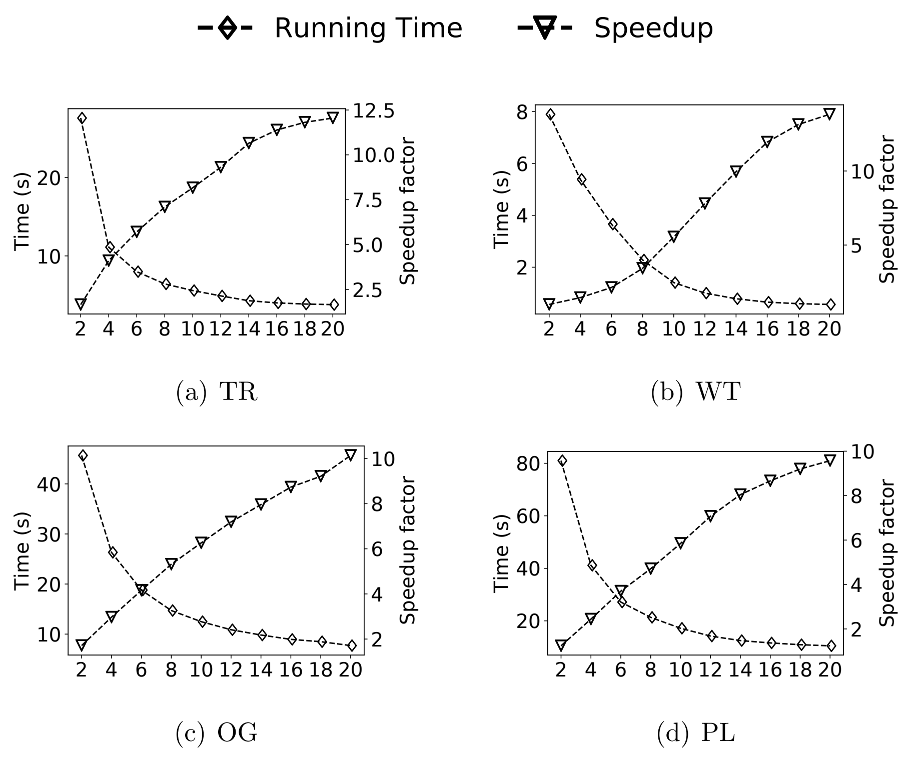
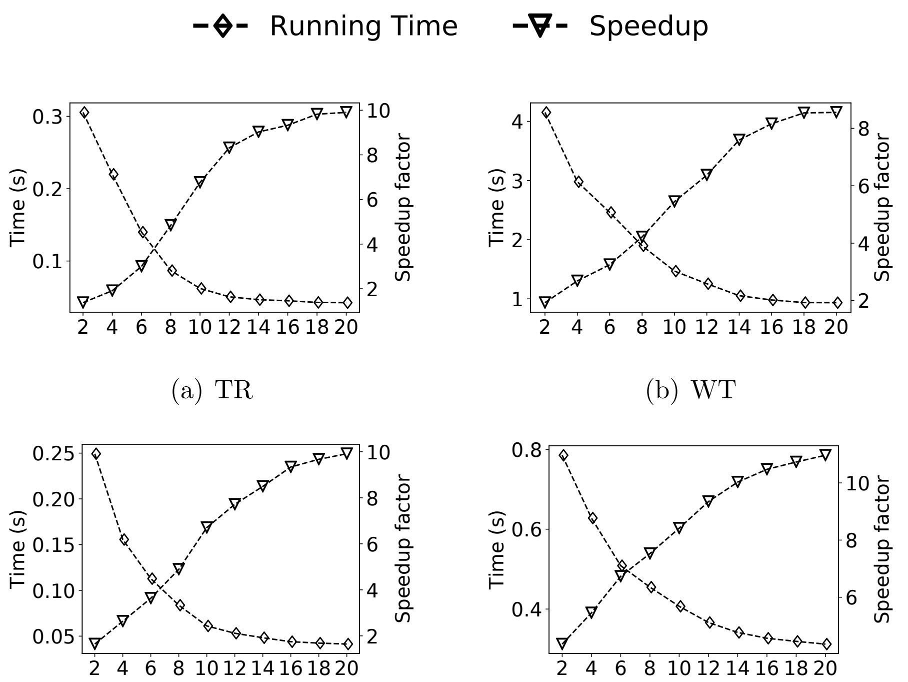


Figure 4.5: ParallelIns with varying number of cores

that the running time of all three algorithms decreases as the number of cores increases. Our parallel algorithms are useful for edge insertion update as they are relatively time consuming compared with edge removal update. For example, for index maintenance on OG, the running time reduces from 47s to 8s as the number of cores increases from 2 to 20. The running time of both algorithms is almost inversely proportional to the number of cores, which shows that they are efficient in practice.

Exp-5: Scalability of parallel maintenance algorithms. We evaluate the scalability of ParallelIns and ParallelRem on TR, WT, OG, and PL in Figure 4.7. To test the scalability, we vary the number of nodes and the number of edges by randomly sampling nodes and edges respectively from 20% to 100% and keeping



(a) TR (b) WT

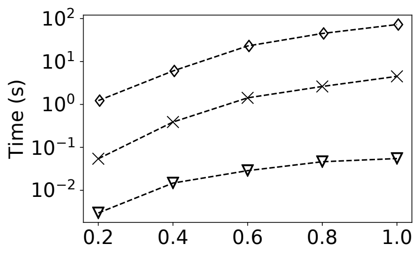
(c) OG (d) PL

Figure 4.6: ParallelRem with varying number of cores

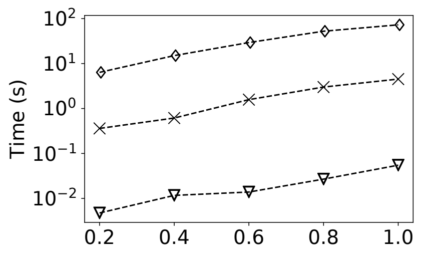
the induced subgraphs as the input graphs. As shown in this experiment, the time cost of all three parallel algorithms increases when varying the number of nodes or edges. Furthermore, the growth trends of both parallel algorithms are similar to their non-parallel partners in Exp-3, which verifies that our parallel algorithms perform well as the graph size grows.

4.7 Conclusion

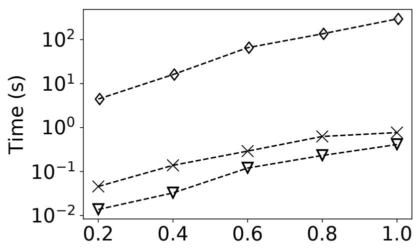
In this chapter, we study the problem of efficient (α, β)-core maintenance. We develop a locality-based algorithm to update BiCore-Index, which decide whether a node in BiCore-Index should be updated or not by visiting its neighbors locally. Moreover, we illustrate how to maintain BiCore-Index when a batch of edges are



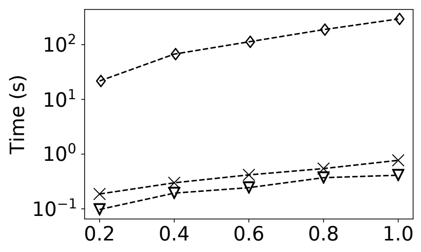
(a) TR (Vary $|U| + |V|$)



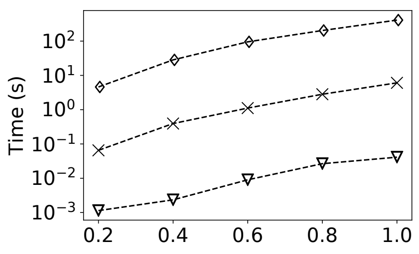
(b) TR (Vary $|E|$)



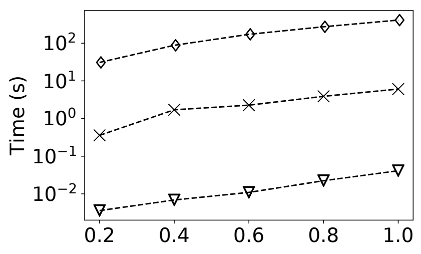
(c) WT (Vary $|U| + |V|$)



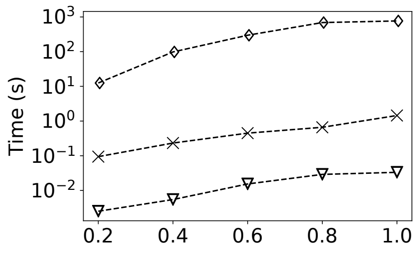
(d) WT (Vary $|E|$)



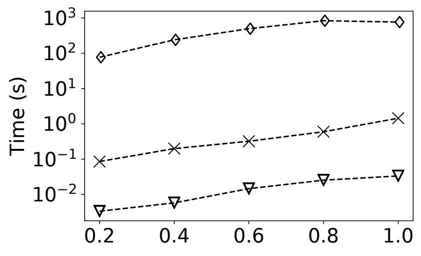
(e) OG (Vary $|U| + |V|$)



(f) OG (Vary $|E|$)



(g) PL (Vary $|U| + |V|$)



(h) PL (Vary $|E|$)

Figure 4.7: Scalability of parallel algorithms

updated. Finally, we discuss how to implement our BiCore-Index maintenance algorithms in parallel. The experimental results demonstrate the efficiency of our proposed algorithms.

Chapter 5

CoreCube: Core Decomposition in Multilayer Graphs

5.1 Introduction

In real-life networks, there are usually multiple types of interactions (edges) among entities (nodes), e.g., the relationship between two users in a social network can be friends, colleagues, relatives and so on. The entities and interactions are usually modelled as a multilayer graph, where each layer records a certain type of interaction among entities [DMR16]. Because of the strong modeling paradigm to handle various interactions among a set of entities, there are significant existing studies of multilayer graphs, e.g., [BGHS12, LSQ$^+$18]. Previous works usually focus on mining dense structures from multilayer graphs according to given parameters, e.g., [ZZL18]. Nevertheless, graph decomposition, as a fundamental graph problem [WQZ$^+$16], remains largely unexplored on multilayer graphs.

Core decomposition (or k-core decomposition), as one of the most well-studied graph decomposition, is to compute the core number for every node in the graph [Sei83]. It is a powerful tool in modeling the dynamic of user engagement in social networks. In practice, a user u tends to adopt a new behavior if there are a considerable number of friends (e.g., the core number of u) in the group who also adopted the same behavior [MV13]. Core decomposition is also theoretically supported by Nash equilibrium in game theory [BKL+15]. It has a variety of applications, e.g., graph visualization [AHDBV05a], internet topology [CHK+07] and user engagement [ZLZ+18, ZZZ+17]. Extending the single-layer core decomposition to multilayer graphs is a critical task which can benefit a lot of applications considering the various real-world interactions between entities.

Given a multilayer graph, the multilayer k-core on a set of layers is defined as a set of nodes whose minimum degree in the induced subgraph of each layer is at least k. The core number of a node on a set of layers is the largest k such that the multilayer k-core on these layers contains the node. Multilayer core decomposition on a set of layers is to compute the core number for each node on these layers. In this paper, we propose CoreCube which records the core numbers of each node for every combination of layers in a multilayer graph. In the following, we show the details for some application examples.

User Engagement Evaluation. In social networks, users may participate in multiple groups with different themes, where each group forms a layer in the multilayer graph. For instance, the authors in a coauthor network have different coauthor relationship on different venues (conferences or journals). For any given user-interested combination of venues (correspond to layers), CoreCube of the coauthor network can immediately answer the engagement level for each author, i,e, the core numbers [MV13]. Given a degree constraint k, we can also immediately retrieve a cohesive user group from CoreCube, i.e., the multilayer k-core.

Biological Module Analysis. In biological networks, different interactions between

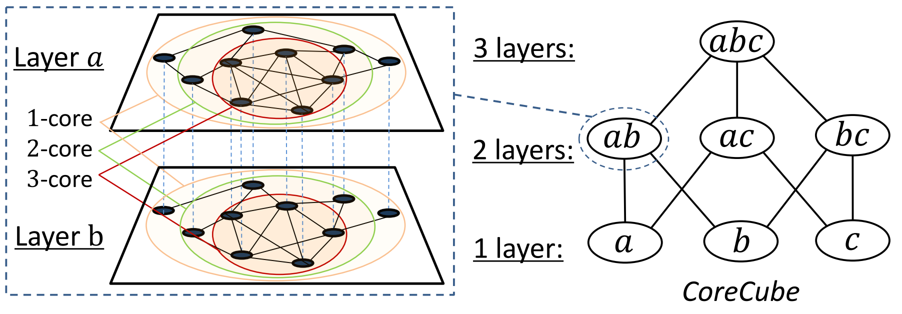


Figure 5.1: Multilayer Core Decomposition and CoreCube of a Graph

the modules are detected with different methods due to data noise and technical limitations [HYH+05]. Analyzing module structure according to single method, i.e., on a single layer, may not be accurate. CoreCube allows us to study the connections between modules for any combination of potential methods. Thus, we can find co-expression clusters and verify the effectiveness of detection methods.

Figure 5.1 shows an example of CoreCube on a graph G with three layers and depicts the multilayer core decomposition on layer a and b. The 3-core on layer a and b contains 5 nodes where each node has a degree of at least 3 in each layer. There are 7 different combinations of layers in CoreCube of G. For each combination, we compute its multilayer core decomposition and record the core numbers in CoreCube. CoreCube can immediately answer a query for core numbers on any set of layers including the traditional single layer graph.

Challenges and Contributions. Although core decomposition on a single-layer graph can be computed in linear time, it becomes very challenging on a multilayer graph because the combination number of layers is exponential to the number of layers. In the general case, no polynomial-time algorithm may exist for computing the CoreCube. To the best of our knowledge, there is only one similar work [GBG17] where the algorithms can be adapted to compute the CoreCube while it is hard to share the computation among different combination of layers. The algorithms proposed in this paper can largely speed up the computation of CoreCube. We

Table 5.1: Summary of Notations

Notation	**Definition**
$G = (V, E, L)$	a multilayer graph, where V is a set of nodes, L is a set of layers, and $E \subseteq (V \times V \times L)$ is a set of edges
$V(G)$	the node set of G
L'; l	$L' \subseteq L$ is a subset of L; $l \in L$ is a layer in L
$E_{L'}$	the edge set in L', i.e., $E_{L'} = E \cap (V \times V \times L')$
u, v	a node in the graph
$\|V\|$, $\|E\|$, $\|L\|$	the number of nodes, edges, and layers in G, respectively
$N_G(v, l)$	the set of adjacent nodes of v in layer l of G
$deg_G(v, l)$	the number of adjacent nodes of v in layer l of G
d_{max}	the maximum degree, i.e., $d_{max} = \max\{deg_G(v, l) \,\|\, v \in V \wedge l \in L\}$
$C^k_{L'}$	the multilayer k-core on a set of layers L'
$\mathcal{C}_{L'}(v)$	the core number of v on a set of layers L'
$\mathcal{C}_{L'}$	the multilayer core decomposition result on a set of layers L'
$\mathcal{C}$	the CoreCube of G, i.e., $\mathcal{C} = \{\mathcal{C}_{L'} \mid L' \subseteq L\}$

summarize our contributions as follows:

- We propose efficient algorithms to compute the CoreCube. Several theorems reveal the inner characteristics of multilayer core decomposition. (Section 5.3)
- We devise a hybrid storage method which has a superior trade-off between query processing time and storage size. (Section 5.4)
- Extensive experiments demonstrate that our CoreCube computation and query processing are faster than baselines by more than one order of magnitude. (Section 5.5)

5.2 Problem Definition

In this section, we give some notations and formally define CoreCube. The notations are summarized in Table 5.1.

We consider an unweighted and undirected multilayer graph $G = (V, E, L)$, where V represents the set of nodes in G, L represents the set of layers, and

$E \subseteq (V \times V \times L)$ represents the set of edges. We use $|V|$, $|E|$, and $|L|$ to denote the number of nodes, edges, and layers, respectively. $N_G(v,l)$ is the set of adjacent nodes of v in layer l. We say a node u is incident to an edge, or an edge is incident to u, if u is one of the endpoints of the edge. We use $deg_G(v,l)$ to denote the number of adjacent nodes of u in layer l. When the context is clear, we omit the the input graph in notations, such as $deg(v,l)$ for $deg_G(v,l)$.

Definition 2.1: Multilayer k-core. Given a multilayer graph $G = (V, E, L)$, a set of layers $L' \subseteq L$ and an integer k, the multilayer k-core of G on L', denoted by $C^k_{L'}$, is the maximum node set such that every node v in the subgraph H induced by $C^k_{L'}$ satisfies $deg_H(v,l) \geq k$ on each $l \in L'$.

Let k_{max} be the maximum possible k such that a multilayer k-core of G on L' exists. The multilayer k-core for all $1 \leq k < k_{max}$ has the following partial containment property:

Property 2.1: *Given a multilayer graph $G = (V, E, L)$ and a set of layers L', $C^{k+1}_{L'} \subseteq C^k_{L'}$ for all $1 \leq k < k_{max}$.*

Next, we define the core number for each $v \in V$.

Definition 2.2: Core Number. Given a multilayer graph $G = (V, E, L)$ and a set of layers $L' \subseteq L$, the core number of v on L', denoted by $\mathcal{C}_{L'}(v)$, is the largest k such that v is contained in multilayer k-core on L', i.e., $\mathcal{C}_{L'}(v) = \max\{k \mid v \in C^k_{L'}\}$.

Based on Property 2.1 and Definition 2.2, we can easily derive following lemma:

Lemma 2.1: *Given a multilayer graph $G = (V, E, L)$, a set of layers L', and an integer k, we have $C^k_{L'} = \{v \in V \mid \mathcal{C}_{L'}(v) \geq k\}$.*

Definition 2.3: Multilayer Core Decomposition. Given a multilayer graph

$G = (V, E, L)$ and a set of layers $L' \subseteq L$, the multilayer core decomposition, denoted by $\mathcal{C}_{L'}$, computes $C^k_{L'}$ for all $1 \leq k \leq k_{max}$.

According to Lemma 2.1, multilayer core decomposition on L' is equivalent to computing the core number $\mathcal{C}_{L'}(v)$ for each $v \in V$. Finally, we give the formal definition of CoreCube and the problem we tackle in this paper.

Definition 2.4: CoreCube. Given a multilayer graph $G = (V, E, L)$, the CoreCube of G, denoted as $\mathcal{C}$, computes multilayer core decomposition on all the subsets of L, i.e., $\mathcal{C} = \{\mathcal{C}_{L'} \mid L' \subseteq L\}$.

Problem Statement. In this paper, we study the problem of efficiently computing and compactly storing CoreCube of multilayer graphs.

5.3 CoreCube Computation

In this section, we present our basic CoreCube computation algorithm and then discuss how to improve the algorithm by sharing computation among multilayer core decomposition on different sets of layers.

5.3.1 Basic CoreCube Algorithm

Based on Property 2.1, given a multilayer graph $G = (V, E, L)$ and a set of layers $L' \subseteq L$, the multilayer core decomposition on L' can be computed in a bottom up manner following the paradigm used for single layer graphs [BZ03], which increases k step by step and iteratively removing nodes whose degree are less than k. We give this algorithm `Core-BU` in Algorithm 22. `Core-BU` computes multilayer core decomposition in increasing order of k. Each time, k is selected as the minimum degree (line 3). Whenever there exists a node v whose degree is no larger than k in some layer $l \in L'$ (line 4), we know that the core number of v is k (line 5) and we remove v with all its incident edges from the graph (line 6). The core numbers are returned in line 9. With the help of bin sort and the efficient data structure proposed in [KBST15] to maintain the minimum degree, `Core-BU` can achieve a time complexity of $O(|E_{L'}| + |V|)$.

The algorithm `CoreCube-BU` which computes CoreCube with `Core-BU` is shown in Algorithm 23. In Algorithm 23, CoreCube is computed level-by-level. Each time, we generate all the subsets of L with the same size z (line 3) and compute multilayer core decomposition on each subset (line 4-5). CoreCube is returned in line 6.

Complexity. Since there are $2^{|L|} - 1$ (expect $\emptyset$) subsets of L need to be processed and `Core-BU` runs in $O(|E_{L'}| + |V|)$ for any subset L', the complexity of `CoreCube-BU` is $O(2^{|L|} \cdot (|E| + |V|))$.

5.3.2 Computation-sharing CoreCube Algorithm

`Core-BU` needs to remove all the edges in $E_{L'}$ when computing multilayer core decomposition on L'. This is because the core number of a node v is obtained only when v is removed. Therefore, `CoreCube-BU` computes each multilayer core decomposition independently. To improve the efficiency of CoreCube computation, we aim at devising an algorithm that shares computation among multilayer core decomposition on different sets of layers. We first extend the locality property of k-core in single layer graphs [MDPM13] to multilayer graphs.

Theorem 3.1: *Given a multilayer graph $G = (V, E, L)$ and a set of layers $L' \subseteq L$, we have the following recursive equations for core number $\mathcal{C}_{L'}(v)$ of a node $v \in V$:*

$$\forall l \in L' \quad M_l(v) = \max\ k\ s.t.\ |\{u \in N(v, l) \mid \mathcal{C}_{L'}(u) \geq k\}| \geq k \tag{5.1}$$

$$\mathcal{C}_{L'}(v) = \min\{M_l(v) \mid l \in L'\} \tag{5.2}$$

Algorithm 22: **Core-BU**(G, L')

```
Input: $G = (V, E, L)$ : a multilayer graph, $L'$ : a subset of $L$
Output: $\mathcal{C}_{L'}$ : the multilayer core decomposition on $L'$
1 $G' \leftarrow G_{L'}$;
2 while $G' \neq \emptyset$ do
3     $k \leftarrow \min\{deg_{G'}(v, l) \mid v \in V(G') \wedge l \in L'\}$;
4     while $\exists v \in V(G')$ and $l \in L'$ : $deg_{G'}(v, l) \leq k$ do
5         $\mathcal{C}_{L'}(v) \leftarrow k$;
6         remove $v$ and its incident edges from $G'$;
7     end while
8 end while
9 return $\mathcal{C}_{L'}$
```

Algorithm 23: **CoreCube-BU**(G)

```
Input  : $G$ : a multilayer graph
Output: $\mathcal{C}$ : the CoreCube of $G$
1 $\mathcal{C} \leftarrow \emptyset$;
2 for each $z = 1$ to $|L|$ do
3     $Z \leftarrow$ {all the subsets of $L$ whose size are $z$};
4     for each $L' \in Z$ do
5         $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{Core-BU}(G, L')\}$;
6     end for
7 end for
8 return $\mathcal{C}$
```

where $N(v, l)$ is the set of adjacent nodes of v in layer l.

Proof. (i) Let $k_c = \min\{M_l(v) \mid l \in L'\}$ and S be the multilayer k_c-core on L'.

Firstly, S must be nonempty as there exists some node u satisfying $\mathcal{C}_{L'}(u) \geq k_c$. According to Equation 5.1 and 5.2, we have $\forall l \in L'$, $|\{u \in N(v,l) \,|\, \mathcal{C}_{L'}(u) \geq k_c\}| \geq k_c$. Therefore, in each layer $l \in L'$, v has at least k_c adjacent nodes in S, which means $v \in S$. Hence, $\mathcal{C}_{L'}(v) \geq k_c$. (ii) On the other hand, according to Equation 5.1 and 5.2, there must exist some $l_0 \in L'$ in which $|\{u \in N(v,l_0) \,|\, \mathcal{C}_{L'}(u) \geq k_c + 1\}| < k_c+1$. Therefore, $\mathcal{C}_{L'}(v) < k_c+1$. Combining the conclusion in (i) and (ii) together, it holds that $\mathcal{C}_{L'}(v) = \min\{M_l(v) \mid l \in L'\}$. □

Following Theorem 3.1, we devise the algorithm `Core-TD` which computes multilayer core decomposition on L' in a top down manner. `Core-TD` iteratively reduces the upper bound of core number for each node. Initially, each node v is assigned an arbitrary upper bound of core number (e.g. the minimum degree of v in L'). Then `Core-TD` keeps updating the upper bound using Equation 5.1 and 5.2 until convergence. The pseudocode of `Core-TD` is given in Algorithm 24. Here, we use $\overline{\mathcal{C}}_{L'}(v)$ to denote the upper bound of $\mathcal{C}_{L'}(v)$. We also use $sup(v,l)$ (support of v) to denote the number of adjacent nodes of v in layer l whose upper bound is no less than $\overline{\mathcal{C}}_{L'}(v)$. That is

$$sup(v,l) = |\{u \in N(v,l) \,|\, \overline{\mathcal{C}}_{L'}(u) \geq \overline{\mathcal{C}}_{L'}(v)\}| \tag{5.3}$$

Note that if $sup(v,l) < \overline{\mathcal{C}}_{L'}(v)$, Equation 5.1 does not hold for v in layer l. Therefore, we can determine whether $\overline{\mathcal{C}}_{L'}(v)$ needs to be updated by comparing $\overline{\mathcal{C}}_{L'}(v)$ with $sup(v,l)$ for each $l \in L'$ instead of scanning all the adjacent nodes of v.

`Core-TD` first initializes $sup(v,l)$ for every node based on Equation 5.3 in line 1. Then it updates node v whose upper bound violates Equation 5.1 in some layer r (line 2). c_0 records the value of $\overline{\mathcal{C}}_{L'}(v)$ before being updated (line 3). `Core-TD` updates $\overline{\mathcal{C}}_{L'}(v)$ according to Equation 5.1 and 5.2 (line 4-9). Then, for each layer $l \in L'$, it recomputes $sup(v,l)$ and updates $sup(u,l)$ for each adjacent node u of v

(line 10-17). $sup(u,l)$ is decreased by 1 if v once contributed to $sup(u,l)$ but not anymore after $\overline{\mathcal{C}}_{L'}(v)$ being updated (line 13-15). Finally, after all the upper bound converges, `Core-TD` sets $\mathcal{C}_{L'}(v)$ as $\overline{\mathcal{C}}_{L'}(v)$ for each node $v \in V$ in line 19 and returns $\mathcal{C}_{L'}$ in line 20.

Complexity. In `Core-TD`, each time when the upper bound of some node v is updated, line 2-18 takes $O(\sum_{l \in L'}(deg(v,l)))$. Since $\overline{\mathcal{C}}_{L'}(v)$ is at least decreased by 1 whenever being updated, the time complexity of `Core-TD` is $O(\sum_{v \in V}(\overline{\mathcal{C}}_{L'}(v) \cdot \sum_{l \in L'} deg(v,l)))$, which is bounded by $O(d_{max} \cdot |E_{L'}|)$ as the maximum degree d_{max} can always serve as an upper bound for any node.

Correctness. The correctness of `Core-TD` is based on Theorem 3.1. When `Core-TD` terminates, Equation 5.1 and Equation 5.2 are satisfied for each node. On the other hand, the value computed for each node cannot be smaller than the core number because it is always an upper bound of the core number. Hence, `Core-TD` correctly computes core number for each node.

The key issue with `Core-TD` is how to initialize the upper bound tight enough such that it can quickly converge. To deal with this issue, we introduce the following lemma: **Lemma 3.1:** *Given a multilayer graph $G = (V, E, L)$ and a node $v \in V$, it holds that $\mathcal{C}_{L_1}(v) \geq \mathcal{C}_{L_2}(v)$ if $L_1 \subseteq L_2$.*

Proof. Let $k = \mathcal{C}_{L_2}(v)$. Based on the definition of core number, there exists a set of nodes $S \subseteq V$ such that each node v in the subgraph H induced by S satisfies $deg_H(v,l) \geq k$ for $l \in L_2$. Since $L_1 \subseteq L_2$, we have $\mathcal{C}_{L_1}(v) \geq k = \mathcal{C}_{L_2}(v)$. $\square$

According to Lemma 3.1, the core number of a node v on L' can serve as an upper bound of v's core number on any superset of L'. Note that if we compute CoreCube level-by-level, we will obtain core numbers on all the subsets of L' when computing multilayer core decomposition on L'. Therefore we can exploit previous

Algorithm 24: **Core-TD**(G, L', $\overline{\mathcal{C}}_{L'}$)

```
Input: $G = (V, E, L)$ : a multilayer graph, $L'$ : a subset of $L$, $\overline{\mathcal{C}}_{L'}$ : upper
       bound of core number on $L'$ for each node in $V$
Output: $\mathcal{C}_{L'}$ : the multilayer core decomposition
1  $sup(v, l) \leftarrow |\{u \in N(v, l) \mid \overline{\mathcal{C}}_{L'}(u) \geq \overline{\mathcal{C}}_{L'}(v)\}|$ for each $v \in V$ and $l \in L'$;
2  while $\exists v \in V'$ and $r \in L'$: $sup(v, r) < \overline{\mathcal{C}}_{L'}(v)$ do
3      $c_0 \leftarrow \overline{\mathcal{C}}_{L'}(v)$;
4      for each $l \in L'$ do
5          $M_l(v) = \max\ k\ s.t.\ |\{u \in N(v, l) \mid \overline{\mathcal{C}}_{L'}(u) \geq k\}| \geq k$;
6          if $\overline{\mathcal{C}}_{L'}(v) > M_l(v)$ then
7              $\overline{\mathcal{C}}_{L'}(v) \leftarrow M_l(v)$;
8          end if
9      end for
10     for each $l \in L'$ do
11         $sup(v, l) \leftarrow |\{u \in N(v, l) \mid \overline{\mathcal{C}}_{L'}(u) \geq \overline{\mathcal{C}}_{L'}(v)\}|$;
12         for each $u \in N(v, l)$ do
13             if $\overline{\mathcal{C}}_{L'}(u) \leq c_0$ and $\overline{\mathcal{C}}_{L'}(u) > \overline{\mathcal{C}}_{L'}(v)$ then
14                 $sup(u, l) \leftarrow sup(u, l) - 1$;
15             end if
16         end for
17     end for
18 end while
19 $\mathcal{C}_{L'}(v) \leftarrow \overline{\mathcal{C}}_{L'}(v)$ for every $v \in V$;
20 return $\mathcal{C}_{L'}$
```

Algorithm 25: **CoreCube-TD(G)**

```
Input: $G$ : a multilayer graph
Output: $\mathcal{C}$ : the CoreCube of $G$
1  $\mathcal{C} \leftarrow \emptyset$;
2  for $z = 1$ to $|L|$ do
3      $Z \leftarrow$ {all the subsets of $|L|$ whose size are $z$};
4      for each $L' \in Z$ do
5          for each $v \in V$ do
6              if $z = 1$ then
7                  $\overline{\mathcal{C}}_{L'}(v) \leftarrow deg(v, l)$ where $l \in L'$;
8              end if
9              else
10                 $\overline{\mathcal{C}}_{L'}(v) \leftarrow \min\{\mathcal{C}_D(v) \mid D \subset L' \wedge |D| = |L'| + 1\}$;
11             end if
12         end for
13         $\mathcal{C} \leftarrow \mathcal{C} \cup \{$Core-TD $(G, L', \overline{\mathcal{C}}_{L'})\}$;
14     end for
15 end for
16 return $\mathcal{C}$
```

computation as much as possible by initializing $\overline{\mathcal{C}}_{L'}(v)$ with the minimum core number of v on all the subsets of L', i.e., $\overline{\mathcal{C}}_L(v) = \min\{\mathcal{C}_P(v)|P \subset L'\}$. Furthermore, based on Lemma 3.1, we actually only need to consider the subsets whose size is only one smaller than $|L'|$ because any the subset of L' whose size is smaller than $|L'| - 1$ must be contained in some subset of L' whose size is $|L'| - 1$.

The algorithm `CoreCube-TD` which computes CoreCube with `Core-TD` is shown

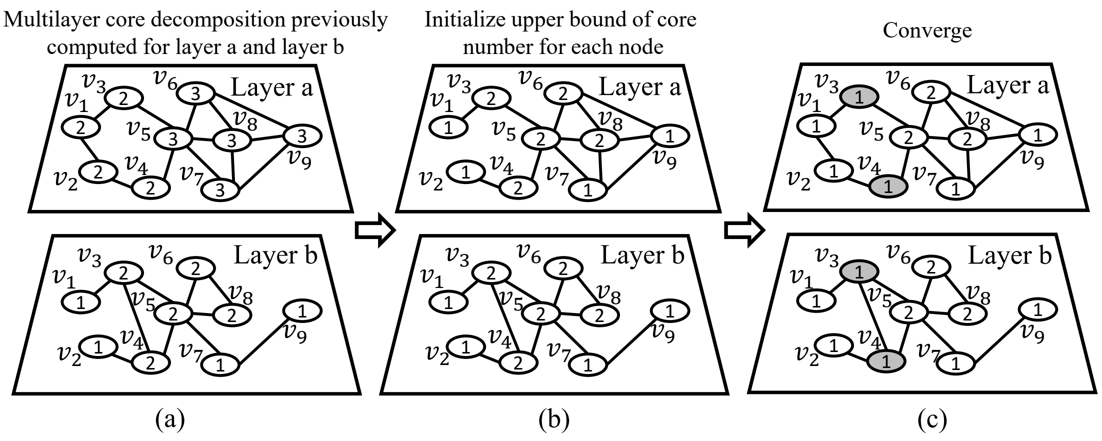

Multilayer core decomposition previously computed for layer a and layer b
Initialize upper bound of core number for each node
Converge
Layer a
Layer b
(a)
(b)
(c)

Figure 5.2: Computing multilayer core decomposition in `CoreCube-TD`

in Algorithm 25. Each time before it invokes `Core-TD` for a set of layers L', it sets the upper bound of core number for each node according to Lemma 3.1 (line 10). If $|L'|$ is 1, it sets the upper bound as the node degree (line 7). Finally, the CoreCube of G is returned in line 16.

Complexity. In `CoreCube-TD`, since the number of subsets D processed in line 10 is $|L'|$, line 5-12 takes $O(|L'| \cdot |V|)$. Considering that there are $2^{|L|} - 1$ subsets of L and `Core-TD` is invoked for each subset, the time complexity of `CoreCube-TD` is bounded by $O(2^{|L|} \cdot (|L| \cdot |V| + d_{max} \cdot |E|))$. Though the time complexity is apparently worse than that of `CoreCube-BU`, we find that much less nodes are visited in our experiments, especially when the number of layers is large. This is because the upper bound is initialized very close to the core number and converges quickly in `Core-TD`.

Example 3.1: We show the procedure of `CoreCube-TD` for computing multilayer core decomposition on the set of layers $\{a, b\}$ in Figure 5.2. Based on the multilayer core decomposition previously computed for each single layer (Figure 5.2 (a)), `CoreCube-TD` initializes the upper bound for each node with the minimum core number in layer a and layer b (Figure 5.2 (b)). Then it invokes `Core-TD` to compute multilayer core decomposition on $\{a, b\}$. As $sup(v_3, a)$ and $sup(v_4, a)$ are 1, which are smaller than their upper bound 2, `Core-TD` updates their upper bound as 1 (Figure 5.2 (c)). Finally, Figure 5.2 (c) is returned as the multilayer core decomposition result. In Figure 5.2, only v_3 and v_4's upper bound are updated since the upper bound of the rest nodes are already equal to their core number before `CoreCube-TD` invokes `Core-TD`. □

5.4 CoreCube Storage

In this section, we devise a method for compactly storing CoreCube and discuss how to process queries for core numbers on any set of layers. A straightforward method is storing core numbers on each set of layers in separate files. Given a core number query, we can directly retrieve the result from the disk. However, this method requires large disk space as we need to store each node in every file. To reduce space usage, we propose two optimization strategies.

Firstly, many nodes' core number on a set of layers L' can be zero when $|L'|$ is large because the core number of a node v is zero if $deg(v, l)$ in some layer $l \in L'$ equals to 0. Therefore, we do not record the node whose core number is zero. Secondly, the core number on L' can remain unchanged when a new layer l is added to L' if the core number on L' is small or the distribution of core number on l is nearly the same as that in L'. Hence, we can store the difference between core numbers on different sets of layers instead of directly storing core number for each node. Here, we call the file that stores nonzero core numbers as absolute storage and the file that stores the difference as relative storage. The algorithm `Hybrid-Storage` which uses both absolute storage and relative storage is given in Algorithm 26.

Algorithm 26: **Hybrid-Storage**(G, $\mathcal{C}$)

```
Input: $G = (V, E, L)$: a multilayer graph, $\mathcal{C}$: the CoreCube of $G$
Output: the files that stores $\mathcal{C}$
1  $Z \leftarrow \{$all the subsets of $|L|\}$;
2  for each $L' \in Z$ do
3      create a new file $F$;
4      if $|L'| = 1$ then
5          for each $v \in V$ and $\mathcal{C}_{L'}(v) \neq 0$ do
6              write $v$ and $\mathcal{C}_{L'}(v)$ into $F$;
7          end for
8      end if
9      else
10         $n_1 \leftarrow$ the number of non zero values in $\mathcal{C}_{L'}$;
11         $P \leftarrow$ the subset of $L'$ s.t. $|\{v \in V \mid \mathcal{C}_P(v) \neq \mathcal{C}_{L'}(v)\}|$ is minimum $\wedge |P| = |L'| - 1$ ;
12         $n_2 \leftarrow |\{v \in V \mid \mathcal{C}_P(v) \neq \mathcal{C}_{L'}(v)\}|$;
13         if $n_1 \leq n_2$ then
14             for each $v \in V$ and $\mathcal{C}_{L'}(v) \neq 0$ do
15                 write $v$ and $\mathcal{C}_{L'}(v)$ into $F$;
16             end for
17         end if
18         else
19             write $P$ as the predecessor into $F$;
20             for each $v \in V$ and $\mathcal{C}_P(v) - \mathcal{C}_{L'}(v) \neq 0$ do
21                 write $v$ and $\mathcal{C}_P(v) - \mathcal{C}_{L'}(v)$ into $F$;
22             end for
23         end if
24     end if
25 end for
```

`Hybrid-Storage` creates a file F for each subset of L (line 3). For the subset consists of single layer, it uses absolute storage to store the nonzero core number

Algorithm 27: **Core-Retrieve**(G, L')

```
Input: $G = (V, E, L)$: a multilayer graph, $L'$: a subset of layers
Output: $\mathcal{C}_{L'}$: the multilayer core number on $L'$
1  $\mathcal{C}_{L'}(v) \leftarrow 0$ for each $v \in V$;
2  $flag \leftarrow true$; $P \leftarrow L'$;
3  while $flag$ do
4      load the file $F$ corresponding to $P$ from disk;
5      if $F$ is relative storage then
6          $P \leftarrow$ the predecessor in $F$;
7      end if
8      else
9          $flag \leftarrow false$;
10     end if
11     $\mathcal{C}_{L'}(v) \leftarrow \mathcal{C}_{L'}(v) + F(v)$ for each $v \in V$;
12 end while
13 return $\mathcal{C}_{L'}$
```

for each node (line 4-8). For other subsets L', it first counts the number of nonzero core number in $\mathcal{C}_{L'}$ as n_1 (line 10). Then, it finds the subset P of L' such that the number of different values between $\mathcal{C}_{L'}$ and $\mathcal{C}_P$ is minimum (line 11) and refers this number as n_2 (line 12). If $n_1 \leq n_2$, `Hybrid-Storage` uses absolute storage (line 13-17). Otherwise, it uses relative storage that stores all the difference between $\mathcal{C}_{L'}$ and $\mathcal{C}_P$ (line 20-22). It also records P as the predecessor (line 19) so that we can know from which subset the difference is made when answering queries.

The algorithm which processes queries for core numbers on a set of layers L' is shown in Algorithm 27. `Core-Retrieve` keeps loading files from disk according to

Table 5.2: Statistics of Datasets

Dataset	**nodes**	**Edges**	**Layers**	**Domain**
`Homo`	18,223	153,922	7	genetic
`SacchCere`	6,571	247,152	7	genetic
`Twitter`	2,281,260	3,827,964	3	social
`Amazon`	410,237	8,132,506	4	co-purchasing
`DBLP`	2,175,466	8,221,193	10	co-authorship
`Flickr`	2,302,927	23,350,524	10	social
`StackOverflow`	6,024,272	28,978,914	10	social
`Wiki`	25,323,885	132,693,853	10	hyperlinks

the predecessors (line 4-7) until it meets absolute storage (line 8-10). Meanwhile, `Core-Retrieve` computes core numbers by summing up the difference stored in each file (line 11). Note that we use $F(v)$ to represent the value (core number or difference) associated with node v stored in file F. Finally, core numbers are returned in line 13. Note that `Core-Retrieve` loads at most $|L'|$ files.

5.5 Experimental Evaluation

5.5.1 Experimental Setting

Datasets. Eight real-life networks were deployed in our experiments. Table 5.2 shows the statistics of the 8 datasets, listed in increasing order of their edge numbers. `Home` and `SacchCere` are networks describing different types of genetic interactions between genes. `Twitter` represents different types of social interaction among Twitter users. `Amazon` is a co-purchasing temporal network, containing four snapshots between March and June 2003. `DBLP` is a co-author network. `Flickr` is a social network represents Flickr users and their friendship connections. `StackOverflow` is a temporal network represents different types of interactions on the website Stack Overflow. `Wiki` contains users and pages from Wikipedia, connected by edit events.

Algorithms. We test 4 algorithms for CoreCube computation. `CoreCube-BU` and `CoreCube-TD` are our algorithms, e.g., Algorithm 23 and Algorithm 25.

`ML-DFS` and `ML-Hybrid` are two state-of-the-art existing solutions proposed in [GBG17]. They compute cores for all the coreness vector $\boldsymbol{k}$, where $\boldsymbol{k}$ is a $|L|$-dimension vector and the value k in each dimension represents that the degree of each node is no less than k in the corresponding layer. `ML-DFS` searches the space of $\boldsymbol{k}$ through depth-first search strategy. `ML-Hybrid` adopts both depth-first and breath-first search strategy. In our experiments, we compute CoreCube by using cores whose $\boldsymbol{k}$ has the same value in every nonzero dimension. For the sake of fairness, *we extract and report the time spent on computing these cores* in `ML-DFS` and `ML-Hybrid` instead of the total running time.

To the best of our knowledge, no existing work investigates the storage of CoreCube. We test three algorithms `Naive-Storage`, `Nonzero-Storage` and `Hybrid-Storage`. `Naive-Storage` stores core numbers without any optimization strategies. `Nonzero-Storage` only stores nonzero core numbers. `Hybrid-Storage` uses both absolute storage and relative storage, i.e., Algorithm 26.

`Core-Retrieve` is our algorithm for answering core number queries, i.e., Algorithm 27. `CoreScratch` computes core numbers from scratch for each query. We divide `CoreScratch` into two procedures, `CoreScratch-Load` and `CoreScratch-Comp`. `CoreScratch-Load` is the procedure that loads the graph from disk into main memory. `CoreScratch-Comp` is the procedure that computes core numbers. For `CoreScratch-Comp`, we test both `Core-BU` and `Core-TD`, and report the running time based on the faster one.

All algorithms are implemented in C++ with -O2 optimization level and tested on an server equipped with Intel Xeon CPU at 2.8GHz and 128GB main memory.

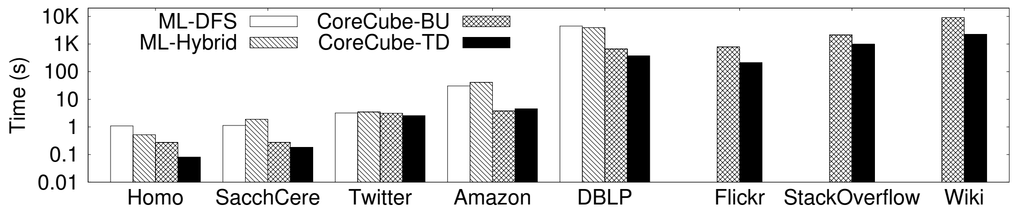


Figure 5.3: CoreCube computation time in all datasets

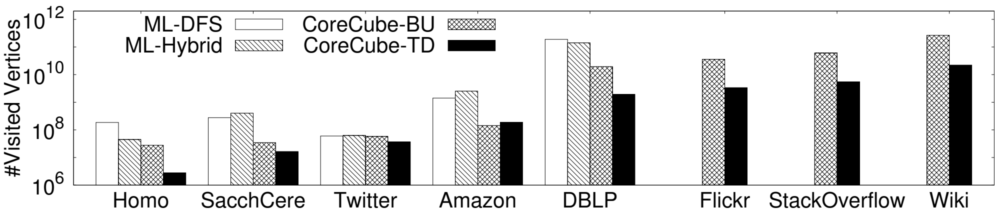


Figure 5.4: The number of visited nodes in CoreCube computation in all datasets

5.5.2 CoreCube Computation

In this set of experiments, we set the maximum running time for each test as 48 hours. If an algorithm cannot stop within the time limit, we omit its running time.

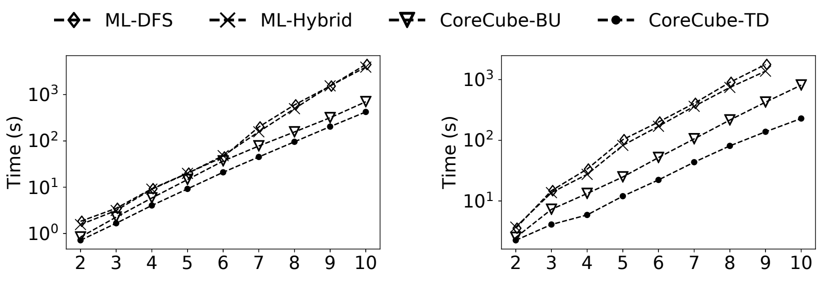
Exp-1: CoreCube Computation Time on Different Datasets. We report the time cost for computing CoreCube on different datasets in Figure 5.3. As shown in Figure 5.3, our proposed algorithm `CoreCube-TD` is the fastest algorithm in all datasets except `Amazon` and achieves one order of magnitude improvement on average compared with existing solutions `ML-DFS` and `ML-Hybrid`. For example, in `DBLP`, `CoreCube-BU` and `CoreCube-TD` spend 662s and 375s respectively while `ML-DFS` and `ML-Hybrid` spend 4487s and 3932s respectively. In the three largest datasets, `ML-DFS` and `ML-Hybrid` cannot terminate within 48 hours.

Exp-2: The Number of Visited nodes in CoreCube Computation. To better demonstrate performance of the four CoreCube computation algorithms, we report the number of visited nodes in Figure 5.4. The number of visited nodes represents how many times the value related to a node is modified or accessed, e.g., removing an edge or decreasing upper bound. For `ML-DFS` and `ML-Hybrid`, the number of visited nodes is collected during the computation of cores that are used for computing CoreCube. As shown in Figure 5.4, the number of visited nodes in `CoreCube-TD` is smallest in all datasets except for `Amazon`. This is because the core numbers in `Amazon` vary a lot on different sets of layers, which leads to slow convergence in `Core-TD`. Compared with our algorithms, the number of visited nodes in `ML-DFS` and `ML-Hybrid` is much larger. The reason is that they need to generate a subgraph that contains some core before computing it.

Exp-3: Scalability of CoreCube Computation. In this experiment, we evaluate the performance of four CoreCube computation algorithms with varying the number of layers. We show results on `DBLP` and `Flickr` in Figure 5.5. The trends are similar in other datasets. As shown in Figure 5.5, the running time of four algorithms stably increases. The gap between existing algorithms and our proposed algorithms becomes larger as the number of layers increases. Compared with existing algorithms, our proposed algorithm `CoreCube-TD` achieves at least 1 order of magnitude improvement when the number of layers excesses 7. Furthermore, the gap between `CoreCube-BU` and `CoreCube-TD` becomes larger with the increasing of layers, which shows that the advantages of `CoreCube-TD` is significant when the number of layers becomes large.

5.5.3 CoreCube Storage and Query Processing

Exp-4: Disk Usage under Different Storage Methods. In this experiment, we report the disk usage of storing CoreCube of all datasets in Figure 5.6. As shown in Figure 5.6, the disk usage of `Hybrid-Storage` is smallest in all datasets. For example, in `DBLP`, the disk usage of `Naive-Storage`, `Nonzero-Storage` and `Hybrid-Storage` are 21GB, 522MB and 302MB respectively. The gap between



(a) DBLP (Vary layers) (b) Flickr (Vary layers)

Figure 5.5: CoreCube computation time with varying number of layers

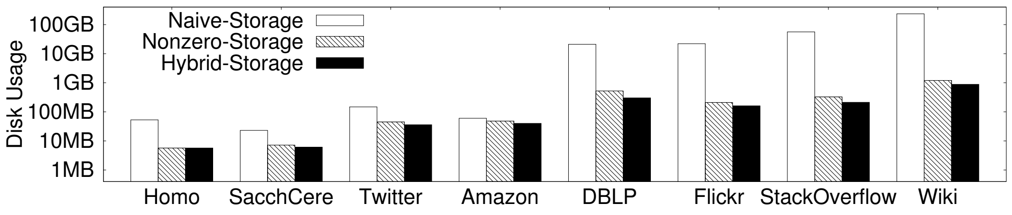


Figure 5.6: CoreCube storage in all datasets

`Naive-Storage` and `Nonzero-Storage` shows that many nodes have zero core number in CoreCube. `Hybrid-Storage` further reduces disk usage by storing the difference between core numbers on different subsets of layers.

Exp-5: Core Number Query Processing Time. In this experiment, we randomly generate 100 core number queries for each dataset. Each core number query asks for core numbers on a specific set of layers. The total running time of answering the 100 queries is reported in Figure 5.7. As shown in Figure 5.7, `Core-Retrieve` finishes 100 queries within 10ms in all datasets including the time spent on loading files from disk. `CoreScratch` spends more than 100s in the largest dataset even if the graph has already been loaded into memory. In real scenarios, graphs cannot always be kept in memory. The advantage of `Core-Retrieve` is more significant when considering the graph loading time in `CoreScratch-Load`.

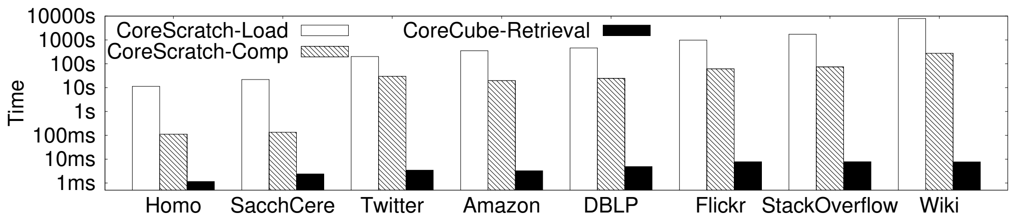


Figure 5.7: Core Number Query Processing Time

5.5.4 Case Study on DBLP

In this section, we test the effectiveness of multilayer core decomposition on DBLP. Here, the multilayer graph has two layers. One layer is the coauthor network of SIGMOD conference. Another one is the coauthor network of KDD conference. Two authors are connected if they collaborated on at least one paper. Both layers are extracted from data from 2013 to 2017.

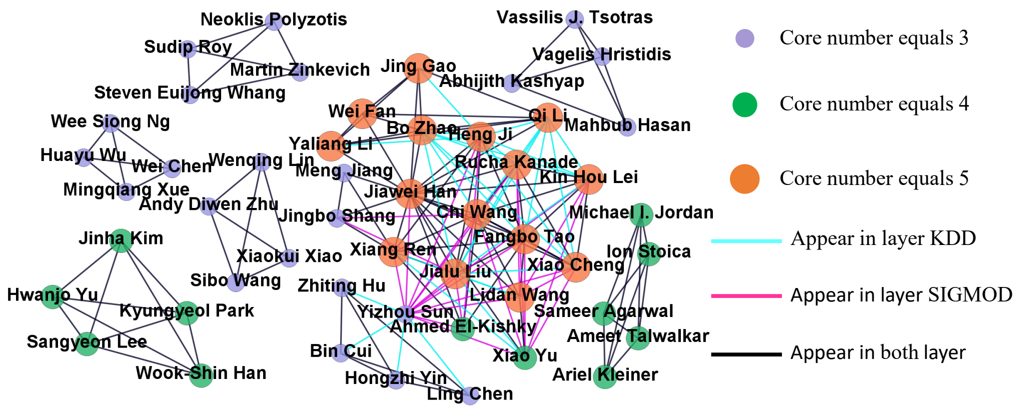


Figure 5.8: Multilayer core decomposition on DBLP

Exp-6: Case Study on DBLP. We show nodes with core number no less than 3 in Figure 5.8. Edges that appear exclusively in KDD and SIGMOD are colored with blue and red respectively. Edges that appear in both layers are colored with black. As shown in Figure 5.8, multilayer core decomposition effectively captures authors with different engagement level in both conferences. Note that the subgraph induced by multilayer k-core are not necessarily connected.

5.6 Conclusion

In this chapter, we study core decomposition on multilayer graphs and propose the CoreCube which records the multilayer core decomposition on every combination of layers. We devise algorithms for efficiently computing and compactly storing Core-Cube. The experimental results validate the efficiency of our proposed algorithms and effectiveness of multilayer core decomposition.

Chapter 6

Conclusion and Future Work

In this chapter, we provide a brief summarization of this thesis and show some possible directions. Specifically, the major contributions of this thesis are concluded in Section 6.1. Section 6.2 introduces several possible directions for future work.

6.1 Conclusions

In this thesis, we study three important problems on core computation in bipartite graphs and multilayer graphs.

Firstly, we study the problem of (α, β)-core computation, a fundamental problem in managing and analyzing bipartite graph data. The (α, β)-core of a bipartite graph $G = (U, V, E)$, denoted by $\mathcal{C}_{\alpha,\beta}$, consists of two node sets $\mathcal{U} \subseteq U(G)$ and $\mathcal{V} \subseteq V(G)$ such that the bipartite subgraph g induced by $\mathcal{U} \cup \mathcal{V}$ is the maximal subgraph of G in which all the nodes in $\mathcal{U}$ have degree at least α and all the nodes in $\mathcal{V}$ have degree at least β. In order to support realtime (α, β)-core query processing, we propose a non-trivial space-efficient index structure, BiCore-Index, with the size bounded by $O(m)$. BiCore-Index supports the optimal computation of (α, β)-core in bipartite graphs. Then, we present ComShrDecom to efficiently construct BiCore-Index. ComShrDecom shares the computation between two node sets of the bipartite graph when conducting the core decomposition. We prove that the time complexity of ComShrDecom is $O(\delta \cdot m)$, where δ is the maximum value such that the (δ, δ)-core in G is nonempty and is bounded by $\sqrt{m}$. As shown in our experiment, our algorithms achieve up to 5 orders of magnitude speedup for computing (α, β)-core and up to 3 orders of magnitude speedup for index construction, respectively, compared with existing techniques. Moreover, we discuss how to implement our index construction algorithms in parallel to further accelerate BiCore-Index construction.

Secondly, we study the problem of BiCore-Index maintenance when graphs are dynamically updated. We improve the performance of BiCore-Index maintenance algorithms in tow folds. First, we show that for a given $\alpha(\beta)$ we only need to recompute one (α, β)-core . Second, we study the locality properties of those nodes that will be influenced after an edge being inserted/removed and show that those nodes can be found through a local search. Our improved maintenance algorithms achieve up to four order of magnitude improvement compared with basic solutions. Then we propose BiCore-Index-Batch to handle the case when a batch of edges are inserted or removed. We also show that we can extend our BiCore-Index maintenance algorithms to run in parallel by splitting them into independent subprocesses and merging the results by selecting the largest/smallest value computed among all subprocess.

Finally, we formulate and investigate the problem of core decomposition on multilayer graphs. We proposed CoreCube which records the core decomposition results of each vertex for every combination of layers in a multilayer graph. Then, we analyze the inner characteristics of multilayer core decomposition and devise efficient algorithms to compute the CoreCube. Due to the result size is exponential to the number of layers, we devise a hybrid storage method which has a superior trade-off between query processing time and storage size. Extensive experiments demonstrate that our CoreCube computation and query processing are faster than baselines by more than one order of magnitude.

6.2 Directions for Future Work

The investigation on mining cohesive subgraph structure is still far from an end. New applications are posing new challenges. In this subsection, we propose several possible directions for future work.

More Cohesiveness Metrics. For structure cohesiveness, apart from the core model studied in this thesis, other models such as k-truss, nucleus, k-ECC, and clique have also been proposed in the literature. Thus, it would be interesting to extend such models to bipartite graphs and multilayer graphs. Furthermore, many real social networks contain keyword attributes or spatial attributes on the nodes. In addition to the network structure, community structure may contain some semantic information, such as attribute-related communities with keyword constraint, geo-social groups with spatial constraint. There are some studies finding cohesive subgraphs from attributed graphs like (k, r)-core [ZZQ+17], (k, d)-ECC [CLZ+18], and r-clique [KA11]. Nevertheless, these works are mainly focusing on unipartite graphs. Therefore, it would be interesting to extend (α, β)-core model and multilayer k-core model to attributed bipartite graphs and multilayer graphs.

I/O Efficient or Distributed Algorithms for Core Computation. In real applications (e.g., Facebook), the graphs may involve trillions of vertices and edges. For big graphs that cannot be kept by a single machine, existing cohesive subgraph detection algorithms based on core model may fail to process such real big graphs within reasonable time cost. Therefore, it would be interesting to develop algorithms based on distributed computation platforms (e.g., GraphX), which are able to process big graphs in a cluster. Moreover, to save memory space, we may keep the graph data on disk and design I/O-efficient query algorithms.

Bibliography

[ABF$^+$07] Adel Ahmed, Vladimir Batagelj, Xiaoyan Fu, Seok-Hee Hong, Damian Merrick, and Andrej Mrvar. Visualisation and analysis of the internet movie database. In *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*, pages 17–24. IEEE, 2007.

[AHDBV05a] J Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*, pages 41–50, 2005.

[AHDBV05b] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition: A tool for the visualization of large scale networks. *arXiv*, 2005.

[AHL12] Alireza Abbasi, Liaquat Hossain, and Loet Leydesdorff. Betweenness centrality as a driver of preferential attachment in the evolution of research collaboration networks. *Journal of Informetrics*, 6(3):403 – 412, 2012.

[AIB$^+$13] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Elisa Bertino, and Norman Foo.

Collusion detection in online rating systems. In Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang, editors, *Web Technologies and Applications*, pages 196–207, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[AKP17] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks*, page cnx001, 2017.

[ARS02] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.

[AUANK$^+$03] Md Altaf-Ul-Amine, Kensaku Nishikata, Toshihiro Korna, Teppei Miyasato, Yoko Shinbo, Md Arifuzzaman, Chieko Wada, Maki Maeda, Taku Oshima, Hirotada 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.

[AYRC$^+$09] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. Group recommendation: Semantics and efficiency. *Proc. VLDB Endow.*, 2(1):754–765, August 2009.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[BGHS12] Brigitte Boden, Stephan Günnemann, Holger Hoffmann, and Thomas Seidl. Mining coherent subgraphs in multi-layer graphs with edge labels. In *SIGKDD*, pages 1258–1266, 2012.

[BH03] Gary D Bader and Christopher WV Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics*, 4(1):2, 2003.

[BKL$^+$15] Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. Preventing unraveling in social networks: the anchored k-core problem. *SIAM Journal on Discrete Mathematics*, 29(3):1452–1475, 2015.

[BXG$^+$13] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*, pages 119–130. ACM, 2013.

[BZ03] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[CB15] Monika Cerinek and Vladimir Batagelj. Generalized two-mode cores. *Social Networks*, 42:80 – 87, 2015.

[CHK$^+$07] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. A model of internet topology using k-shell decomposition. *PNAS*, 104(27):11150–11154, 2007.

[CLZ$^+$18] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. Maximum co-located community search in large scale social networks. *Proceedings of the VLDB Endowment*, 11(10):1233–1246, 2018.

[CM13] Lucas Augusto Montalvão Costa Carvalho and Hendrik Teixeira Macedo. Users' satisfaction in recommendation systems for groups:

an approach based on noncooperative games. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 951–958. ACM, 2013.

[DBS18] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of SPAA*, pages 393–404, 2018.

[DFBG09] Carsten F Dormann, Jochen Fründ, Nico Blüthgen, and Bernd Gruber. Indices, graphs and null models: analyzing bipartite ecological networks. 2009.

[DJN$^+$13] Madelaine Daianu, Neda Jahanshad, Talia M. Nir, Arthur W. Toga, Clifford R. Jack Jr., Michael W. Weiner, and Paul M. Thompson. Breakdown of brain connectivity between normal aging and alzheimer's disease: A structural k-core network analysis. *Brain Connectivity*, 3(4):407–422, 2013.

[DLHM17] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. Efficient fault-tolerant group recommendation using alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 2047–2050, New York, NY, USA, 2017. ACM.

[DMR16] Mark E Dickison, Matteo Magnani, and Luca Rossi. *Multilayer social networks.* Cambridge University Press, 2016.

[ELS15] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of WWW*, pages 300–310, 2015.

[FCC$^+$17] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. Effective and efficient attributed community search. *The VLDB Journal*, 26(6):803–828, 2017.

[FCL$^+$17] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment*, 10(6):709–720, 2017.

[FCL$^+$18] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Long Yuan. Distributed computing connected components with linear communication cost. *Distributed and Parallel Databases*, 36(3):555–592, 2018.

[FCLH16] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment*, 9(12):1233–1244, 2016.

[FLL$^+$11] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 925–936. ACM, 2011.

[FWC$^+$18a] Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, and Xiaojun Chen. On spatial-aware community search. *IEEE TKDE*, 31(4):783–798, 2018.

[FWC$^+$18b] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. Effective and efficient community search over large directed graphs. *IEEE TKDE*, 2018.

[FYC$^+$19] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. Efficient algorithms for densest subgraph discovery. *Proceedings of the VLDB Endowment*, 12(11):1719–1732, 2019.

[GBG17] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. Core decomposition and densest subgraph in multilayer networks. In *CIKM*, pages 1807–1816, 2017.

[GL04] Jean-Loup Guillaume and Matthieu Latapy. Bipartite structure of all complex networks. *Information processing letters*, 90(5):215–221, 2004.

[GL06] Jean-Loup Guillaume and Matthieu Latapy. Bipartite graphs as models of complex networks. *Physica A: Statistical Mechanics and its Applications*, 371(2):795–813, 2006.

[GLRW13] Jagadeesh Gorla, Neal Lathia, Stephen Robertson, and Jun Wang. Probabilistic group recommendation via information matching. In *Proceedings of the 22nd international conference on World Wide Web*, pages 495–504. ACM, 2013.

[GMRS11] S. Gunnemann, E. Muller, S. Raubach, and T. Seidl. Flexible fault tolerant subspace clustering for data with missing values. In *2011 IEEE 11th International Conference on Data Mining*, pages 231–240, Dec 2011.

[GTV11a] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 201–210. IEEE, 2011.

[GTV11b] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 87–93. IEEE, 2011.

[GXL+10] Mike Gartrell, Xinyu Xing, Qin Lv, Aaron Beach, Richard Han, Shivakant Mishra, and Karim Seada. Enhancing group recommendation by incorporating social relationship interactions. In *Proceedings of the 16th ACM international conference on Supporting group work*, pages 97–106. ACM, 2010.

[Hoc98] Dorit S Hochbaum. Approximating clique and biclique problems. *Journal of Algorithms*, 29(1):174–200, 1998.

[HYH+05] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. In *International Conference on Intelligent Systems for Molecular Biology*, pages 213–221, 2005.

[JB15] Vinay Jethava and Niko Beerenwinkel. Finding dense subgraphs in relational graphs. In *ECML PKDD*, pages 641–654, 2015.

[KA11] Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment*, 4(10):681–692, 2011.

[KBST15] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[KGH+10] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. Identification of influential spreaders in complex networks. *Nature physics*, 6(11):888–893, 2010.

[KKND11] Mehdi Kaytoue, Sergei O Kuznetsov, Amedeo Napoli, and Sébastien Duplessis. Mining gene expression data with pattern structures in formal concept analysis. *Information Sciences*, 181(10):1989–2001, 2011.

[KNT10] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. *Structure and Evolution of Online Social Networks*, pages 337–357. Springer New York, New York, NY, 2010.

[KPPS14] Tamara G Kolda, Ali Pinar, Todd Plantenga, and Comandur Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, 2014.

[KTV97] Ravi Kannan, Prasad Tetali, and Santosh Vempala. Simple markov-chain algorithms for generating bipartite graphs and tournaments. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 193–200. Society for Industrial and Applied Mathematics, 1997.

[Ley02] Michael Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proc. Int. Symposium on String Processing and Information Retrieval*, pages 1–10, 2002.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[LLW10] Xiaowen Liu, Jinyan Li, and Lusheng Wang. Modeling protein interacting groups by quasi-bicliques: Complexity, algorithm, and application. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 7(2):354–364, April 2010.

[LP49] R Duncan Luce and Albert D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[LSE+18] Ricky Laishram, Ahmet Erdem Sariyüce, Tina Eliassi-Rad, Ali Pinar, and Sucheta Soundarajan. Measuring and improving the core resilience of networks. In *WWW*, pages 609–618, 2018.

[LSH08] Sune Lehmann, Martin Schwartz, and Lars Kai Hansen. Biclique communities. *Phys. Rev. E*, 78:016108, Jul 2008.

[LSLW] Jinyan Li, Kelvin Sim, Guimei Liu, and Limsoon Wong. *Maximal Quasi-Bicliques with Balanced Noise Tolerance: Concepts and Co-clustering Applications*, pages 72–83.

[LSLW08] Jinyan Li, Kelvin Sim, Guimei Liu, and Limsoon Wong. Maximal quasi-bicliques with balanced noise tolerance: Concepts and co-clustering applications. In *Proceedings of the 2008 SIAM International Conference on Data Mining*, pages 72–83. SIAM, 2008.

[LSQ+18] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. Persistent community search in temporal networks. In *ICDE*, pages 797–808, 2018.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, (1):76–80, 2003.

[Luc50] R Duncan Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[LYL$^+$19] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (α, β)-core computation: An index-based approach. In *Proceedings of WWW*, pages 1130–1141, 2019.

[LYM13] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2013.

[LZZ$^+$19] Boge Liu, Fan Zhang, Chen Zhang, Wenjie Zhang, and Xuemin Lin. Corecube: Core decomposition in multilayer graphs. In *WISE*, pages 694–710. Springer, 2019.

[MDFM19] Flaviano Morone, Gino Del Ferraro, and Hernán A Makse. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics*, 15(1):95, 2019.

[MDPM13] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.

[MV13] Fragkiskos D. Malliaros and Michalis Vazirgiannis. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*, pages 469–478, 2013.

[NOHA09] JC Nacher, T Ochiai, M Hayashida, and T Akutsu. A mathematical model for generating bipartite graphs and its application to protein networks. *Journal of Physics A: Mathematical and Theoretical*, 42(48):485005, 2009.

[NSNK12] Eirini Ntoutsi, Kostas Stefanidis, Kjetil Nørvåg, and Hans-Peter Kriegel. Fast group recommendations by applying user clustering. In *International Conference on Conceptual Modeling*, pages 126–140. Springer, 2012.

[NSRK14] Eirini Ntoutsi, Kostas Stefanidis, Katharina Rausch, and Hans-Peter Kriegel. "strength lies in differences": Diversifying friends for recommendations through subspace clustering. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 729–738, New York, NY, USA, 2014. ACM.

[OMK15] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. Efficient pagerank tracking in evolving networks. In *Proceedings of SIGKDD*, pages 875–884, 2015.

[OZZ07] Ricardo V. Oliveira, Beichuan Zhang, and Lixia Zhang. Observing the evolution of internet as topology. *SIGCOMM Comput. Commun. Rev.*, 37(4):313–324, August 2007.

[Pee03] René Peeters. The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131(3):651–654, 2003.

[PG09] Ardian Kristanto Poernomo and Vivekanand Gopalkrishnan. Towards efficient mining of proportional fault-tolerant frequent itemsets. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 697–706, New York, NY, USA, 2009. ACM.

[PZZ$^+$18] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin.

Efficient probabilistic k-core computation on uncertain graphs. In *Proceedings of ICDE*, pages 1192–1203. IEEE, 2018.

[SB13] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of PPoPP*, pages 135–146, 2013.

[SEF16] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Corescope: Graph mining using k-core analysis - patterns, anomalies and algorithms. In *ICDM*, pages 469–478, 2016.

[Sei83] Stephen B Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[SF78] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978.

[SGJS+13] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 2013.

[SGJS+16] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB JournalThe International Journal on Very Large Data Bases*, 25(3):425–447, 2016.

[SLGL06] Kelvin Sim, Jinyan Li, Vivekanand Gopalkrishnan, and Guimei Liu. Mining maximal quasi-bicliques to co-cluster stocks and financial ratios for value investment. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 1059–1063. IEEE, 2006.

[SMST18] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2150–2159. ACM, 2018.

[SP18] Ahmet Erdem Sarıyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 504–512, New York, NY, USA, 2018. ACM.

[SRM14] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *Proceedings of IPDPS*, pages 550–559, 2014.

[SRTU09] Serguei Saavedra, Felix Reed-Tsochas, and Brian Uzzi. A simple model of bipartite cooperation for ecological and organizational networks. *Nature*, 457(7228):463–466, 2009.

[UBMK12] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon Kleinberg. Structural diversity in social contagion. *PNAS*, 109(16):5962–5966, 2012.

[WA05] Stefan Wuchty and Eivind Almaas. Peeling the yeast protein network. *Proteomics*, 5(2):444–449, 2005.

[WCL$^+$18] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. Efficient computing of radius-bounded k-cores. In *Proceedings of ICDE*, pages 233–244, 2018.

[WDVR06] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 501–508. ACM, 2006.

[WHZ$^+$18] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, KDD '18, pages 839–848, New York, NY, USA, 2018. ACM.

[WJZZ15] Yubao Wu, Ruoming Jin, Xiaofeng Zhu, and Xiang Zhang. Finding dense and connected subgraphs in dual networks. In *ICDE*, pages 915–926, 2015.

[WQZ$^+$16] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. I/O efficient core graph decomposition at web scale. In *ICDE*, pages 133–144, 2016.

[WYL$^+$19] Xudong Wu, Long Yuan, Xuemin Lin, Shiyu Yang, and Wenjie Zhang. Towards efficient k-tripeak decomposition on large graphs. In *Proceedings of DASFAA*, pages 604–621, 2019.

[YCL14] Quan Yuan, Gao Cong, and Chin-Yew Lin. Com: A generative model for group recommendation. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 163–172, New York, NY, USA, 2014. ACM.

[YQL$^+$15] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Diversified top-k clique search. In *ICDE*, pages 387–398, 2015.

[YQL$^+$16a] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Diversified top-k clique search. *VLDB J.*, 25(2):171–196, 2016.

[YQL$^+$16b] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. I/O efficient ECC graph decomposition via graph reduction. *PVLDB*, 9(7):516–527, 2016.

[YQL$^+$17a] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Effective and efficient dynamic graph coloring. *PVLDB*, 11(3):338–351, 2017.

[YQL$^+$17b] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.*, 26(2):275–300, 2017.

[YQZ$^+$18] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. Index-based densest clique percolation community search in networks. *IEEE Trans. Knowl. Data Eng.*, 30(5):922–935, 2018.

[ZLWX14] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: A total order approach. In *Proceedings of SIGMOD*, pages 1323–1334, 2014.

[ZLZ$^+$18] Fan Zhang, Conggai Li, Ying Zhang, Lu Qin, and Wenjie Zhang. Finding critical users in social communities: The collapsed core and truss problems. *TKDE*, 2018.

[ZP12] Yang Zhang and Srinivasan Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *ICDE*, pages 1049–1060, 2012.

[ZPR$^+$14] Yun Zhang, Charles A. Phillips, Gary L. Rogers, Erich J. Baker, Elissa J. Chesler, and Michael A. Langston. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC Bioinformatics*, 15(1):110, Apr 2014.

[ZYZ$^+$18] Fan Zhang, Long Yuan, Ying Zhang, Lu Qin, Xuemin Lin, and Alexander Zhou. Discovering strong communities with user engagement and tie strength. In *DASFAA*, pages 425–441, 2018.

[ZYZQ17] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *Proceedings of ICDE*, pages 337–348, 2017.

[ZZL18] Rong Zhu, Zhaonian Zou, and Jianzhong Li. Diversified coherent core search on multi-layer graphs. In *ICDE*, pages 701–712, 2018.

[ZZQ$^+$17] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. When engagement meets similarity: Efficient (k, r)-core computation on social networks. *PVLDB*, 10(10):998–1009, 2017.

[ZZQ$^+$18] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficiently reinforcing social networks over user engagement and tie strength. In *ICDE*, pages 557–568, 2018.

[ZZZ$^+$17] Fan Zhang, Wenjie Zhang, Ying Zhang, Lu Qin, and Xuemin Lin.

OLAK: an efficient algorithm to prevent unraveling in social networks. *PVLDB*, 10(6):649–660, 2017.