Efficient (α, β)-core Computation: an Index-based Approach

Boge Liu
University of New South Wales
boge.liu@unsw.edu.au

Long Yuan*
Nanjing University of Science and Technology
longyuan@njust.edu.cn

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

Lu Qin
University of Technology, Sydney
lu.qin@uts.edu.au

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

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com

ABSTRACT

The problem of computing (α, β)-core in a bipartite graph for given α and β is a fundamental problem in bipartite graph analysis and can be used in many applications such as online group recommendation, fraudsters detection, etc. Existing solution to computing (α, β)-core needs to traverse the entire bipartite graph once. Considering the real bipartite graph can be very large and the requests to compute (α, β)-core can be issued frequently in real applications, the existing solution is too expensive to compute the (α, β)-core. In this paper, we present an efficient algorithm 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. Moreover, we 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. We also discuss efficient algorithms to maintain the index when the bipartite graph is dynamically updated and parallel implementation of the index construction algorithm. 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, respectively, compared with existing techniques.

KEYWORDS

Graph processing, optimization, indexing

ACM Reference Format:
Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core Computation: an Index-based Approach. In *WWW 2018: The 2019 Web Conference, May 13–17, 2019, San Francisco, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558.3313522

1 INTRODUCTION

Many real-world relationships across various entities can be modelled as bipartite graphs, such as customer-product networks [40], user-page networks [8], gene co-expression networks [21], collaboration networks [24], 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 [10, 11]. 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

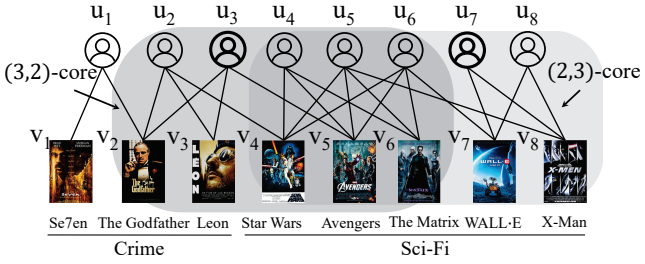

Figure 1: Part of a customer-movie network

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 [4, 9, 15, 49]. Fault-tolerant group recommendation is proposed to deal with missing values in incomplete data and has shown its effectiveness in group recommendation [18, 31, 33]. 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 [30] to compute the relevance of products to u_q. Recently, to accelerate the computation of fault-tolerant subspace clustering, [11] 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

*Corresponding author.

WWW 2019, May 13–17, 2019, San Francisco, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313522

the degree of tolerance for missing values [11], efficiently computing (α, β)-core is a critical procedure for online fault-tolerant group recommendation.

Example 1.1: Figure 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 calculate movie preference. In this case, Sci-Fi movies would be recommended to the group as both u_3 and u_7 have preference for Sci-Fi movies. □

(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 [2, 8]. 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 [23, 52] and quasi-biclique computation [25, 28].

Motivations. In the literature, an online algorithm [11] 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 [27, 41]; in the application of online group recommendation, there can be millions of groups issuing recommendation requests at the peak time [12, 27, 49]. 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 [27]). 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 [2, 8] and we don't want to miss out the suspicious people. Motivated by this, in this paper, 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 (α, β) 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 (general) graphs [7] 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 4.1), 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., 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.

Contributions. In this paper, we overcome the above challenges and make the following contributions:

(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.

(3) Efficient incremental maintenance algorithm for dynamic graphs and parallel implementation of index construction algorithm. In many applications, graphs are frequently updated. Therefore, we develop an efficient incremental algorithm to maintain BiCore-Index for dynamic graphs by reducing unnecessary recomputation in the procedure of updating BiCore-Index. Moreover, we also discuss the parallel implementation of our index construction algorithm.

(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.

Outline. Section 2 gives the problem definition and the existing solution. Section 3 introduces our proposed index, BiCore-Index, and the optimal algorithm to compute (α, β)-core for arbitrary α and β. Section 4 presents algorithms to construct BiCore-Index. Section 5 presents an efficient algorithm to maintain BiCore-Index

in dynamic graphs and discusses parallel implementation of the index construction algorithm. Section 6 evaluates our algorithms using extensive experiments. Section 7 reviews the related work and Section 8 concludes the paper.

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 $\deg(u, G)$, 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., $\mathsf{dmax}_U(G) = \max\{\deg(u, G) | u \in U(G)\}$ ($\mathsf{dmax}_V(G) = \max\{\deg(v, G) | v \in V(G)\}$). 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 $C_{\alpha,\beta}$, 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., $\forall u \in \mathcal{U}, \deg(u, \mathrm{g}) \geq \alpha \wedge \forall v \in \mathcal{V}, \deg(v, \mathrm{g}) \geq \beta$. □

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 $C_{\alpha,\beta}$ is proposed in [11]. Intuitively, it computes $C_{\alpha,\beta}$ 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 $C_{\alpha,\beta}$ 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 $Q_{\alpha,\beta}$.

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. We first introduce a naive index structure. After analyzing the problems in the naive index structure, we present our linear space index structure, BiCore-Index. Based on BiCore-Index, we propose an optimal query processing algorithm. At last, we analyze the space complexity of BiCore-Index.

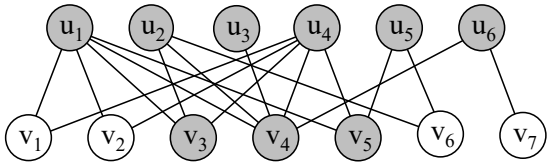

Figure 2: A bipartite graph G

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 $C_{\alpha,\beta}$ in optimal time by visiting the nodes stored in the location referred by the (α, β) value. We show the naive index in the following example.

Example 3.1: Considering the graph G in Figure 2, the naive index of G is shown in Figure 3. For ease of presentation, we only show the nodes in $U(G)$ in Figure 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. 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. 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 $C_{1,3}$ by following bold arrows and obtain $C_{1,3}.\mathcal{U} = \{u_1, \ldots, u_6\}$. □

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, 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 $C_{1,1}$, $C_{1,2}$, $C_{1,3}$, $C_{1,4}$ and $C_{1,5}$. 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 $C_{5,3}$, $C_{5,4}$ and $C_{5,5}$ 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, the (α, β)-cores in G has the following monotonic containment relationship:

Lemma 3.1: *Given a bipartite graph G, $C_{\alpha,\beta}$ is contained in $C_{\alpha',\beta'}$ if $\beta' \leq \beta$ and $\alpha' \leq \alpha$.*

Based on Lemma 3.1, 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

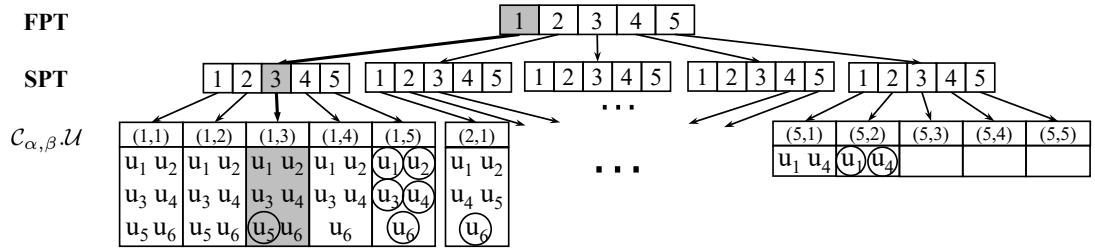

Figure 3: Naive Index

u_1 is contained in $C_{1,5}$, we know u_5 is also contained in $C_{1,1}, C_{1,2}$, $C_{1,3}$ and $C_{1,4}$. In other word, storing u_1 at $C_{1,1}, C_{1,2}$, $C_{1,3}$ and $C_{1,4}$ is redundant regarding (α, β)-core query processing and we only need to store it at $C_{1,5}$ (marked with circle in Figure 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 contain 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, after the node removal, $C_{1,1}.\mathcal{U}$ is empty. To address this problem, the naive index structure for nodes in $V(G)$.

Following the above idea, we give the formal definition of our index. Before that, to characterize 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 $C_{\alpha,\beta}$. 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 $C_{\alpha,\beta}$. 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 $\mathbb{I}^V$ is symmetrical to $\mathbb{I}^U$, 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 equals to the maximum β value among the node blocks pointed by this 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 associated with (α, β') where $\beta' \geq \beta$.

Example 3.2: Figure 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. □

Algorithm 1 QueryOPT

Input: $\mathbb{I}$ of G and $Q_{\alpha,\beta}$
Output: $C_{\alpha,\beta}$ of G

```
1:  C_{α,β} ← ∅;
2:  if 𝕀^U.FPT.size() < α or 𝕀^U[α].size() < β then
3:      return ∅;
4:  nb ← node block pointed by 𝕀^U[α][β];
5:  while the first element of the associated value of nb = α do
6:      for each u ∈ nb do
7:          C_{α,β}.𝒰 ← C_{α,β}.𝒰 ∪ u;
8:      nb ← next node block in 𝕀^U.NB;
9:  Compute C_{α,β}.𝒱 similarly
10: return C_{α,β} ;
```

3.2 Optimal Query Processing

With BiCore-Index, for a query $Q_{\alpha,\beta}$, we compute $C_{\alpha,\beta}$ by retrieving $C_{\alpha,\beta}.\mathcal{U}$ through $\mathbb{I}^U$ and $C_{\alpha,\beta}.\mathcal{V}$ 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 $C_{\alpha,\beta}.\mathcal{U}$ and computes the node block nb referred by the pointer in $\mathbb{I}^U[\alpha][\beta]$ (line 4). 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 5 and 8). All the nodes in visited nb are added into $C_{\alpha,\beta}.\mathcal{U}$ (line 6-7). The nodes in $C_{\alpha,\beta}.\mathcal{V}$ are retrieved similarly and $C_{\alpha,\beta}$ is returned at the end (line 9-10).

Example 3.3: Figure 4 illustrates the procedure of QueryOPT to compute $C_{1,3}$. Processing steps are shown in bold arrows and the visited elements are marked in grey. To compute $C_{1,3}.\mathcal{U}$, QueryOPT

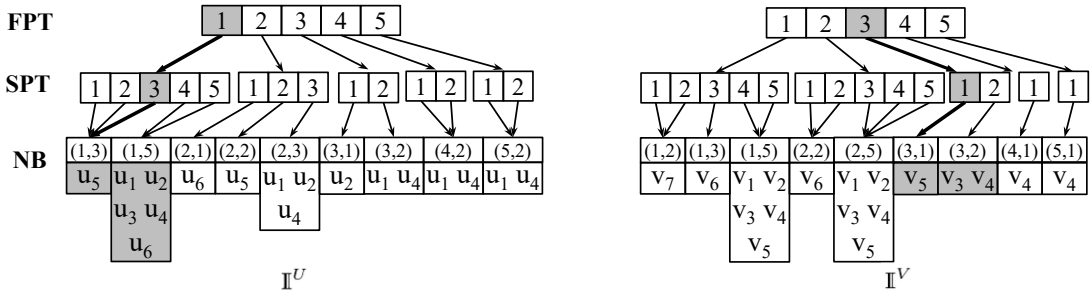

Figure 4: BiCore-Index **and procedure of** QueryOPT **for** $Q_{1,3}$

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 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, $C_{1,3}.\mathcal{U} = \{u_1, u_2, u_3, u_4, u_5, u_6\}$. 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 $C_{1,3}.\mathcal{V} = \{v_3, v_4, v_5\}$. □

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

Proof: For a given α, each $u \in U(G)$ appears at most once in the node blocks pointed by the pointers in α-th array in SPT. Therefore, no duplicate node is added in $C_{\alpha,\beta}.\mathcal{U}$ in line 7. Similarly, no duplicate node is added in $C_{\alpha,\beta}.\mathcal{V}$ in line 9. Since all the nodes visited in QueryOPT are exactly the nodes we need to retrieve, QueryOPT computes $C_{\alpha,\beta}$ in $O(|C_{\alpha,\beta}.\mathcal{U}| + |C_{\alpha,\beta}.\mathcal{V}|)$ time. □

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 3.2. Then, we prove that the space complexity of BiCore-Index is $O(m)$ in Theorem 3.2.

Lemma 3.2: *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 $\alpha > \deg(u_i, G)$, u_i only influences the length of the k-th arrays in SPT with $1 \le k \le \deg(u_i, G)$. Because the length of the α-th array increases at most one after the insertion of u_i, the size of SPT increases at most $\deg(u_i, G)$. Thus, the space of SPT in $\mathbb{I}^U$ is bounded by $O(\sum_{u \in U(G)} \deg(u, G)) = O(m)$. 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 $O(m)$. For each node $u \in U(G) \cup V(G)$, the number of node blocks containing u is $\deg(u)$. Hence, the space of NB is $O(\sum_{u \in U(G) \cup V(G)} \deg(u, G)) = O(m)$. Based on Lemma 3.2, the space of BiCore-Index can be bounded by $O(m)$. □

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 consistence 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 4.3. Therefore, we first present techniques to conduct the core decomposition.

4.1 A Basic Core Decomposition Algorithm

Inspired by the algorithm in [11], considering a node $u \in U(G)$ and a specific α, if $u \in C_{\alpha,\beta}.\mathcal{U}$ and $u \notin C_{\alpha,\beta+1}.\mathcal{U}$, we know $\beta_{\max,\alpha}(u) = \beta$. Moreover, for a specific α, $C_{\alpha,\beta+1}$ is contained in $C_{\alpha,\beta}$. 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. 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 value of α for all nodes in $U(G)$ is dmax_U, it iterates α between 1 and dmax_U and computes $\beta_{\max,\alpha}(u)$ for $u \in U(G)$ regarding the specific α by invoking compute$\beta_{\max}$ (line 1-2). Similarly, $\alpha_{\max,\beta}(v)$ for $v \in V(G)$ are computed in line 3-4.

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 7-8). Then, it processes the nodes in $U(G)$ in increasing of β (line 10). Whenever a node v with $\deg(v, G') \le \beta$ is removed (line 11), if there exists a node u with $\deg(u, G') < \alpha$ in G', we know that $u \in C_{\alpha,\beta}$ but $u \notin C_{\alpha,\beta+1}$, which means $\beta_{\max,\alpha}(u)$ regarding α is β (line 14). 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 (line 16-17).

Example 4.1: Considering the graph in Figure 2, Figure 5 shows the procedure of BasicDecom to conduct the core decomposition. Since

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 α = 1 to dmax_U do
2:   compute β_max(G, α);
3: for β = 1 to dmax_V do
4:   compute α_max(G, β);
5: procedure compute β_max(G, α)
6:   G' ← G;
7:   while ∃u ∈ U(G') : deg(u, G') < α do
8:     remove u and its incident edges from G';
9:   while G' ≠ ∅ do
10:    β ← min_{v∈V(G')} deg(v, G');
11:    while ∃v ∈ V(G') : deg(v, G') ≤ β do
12:      remove v and its incident edges from G';
13:      while ∃u ∈ U(G') : deg(u, G') < α do
14:        β_{max,α}(u) ← β;
15:        remove u and its incident edges from G';
16: procedure compute α_max(G, β)
17:   line 6-15 by interchanging u with v, U with V, α with β;
```

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 5: Decomposition procedure of Algorithm 2

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 as both dmax_U and dmax_V are 5. □

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

Proof: The removal of node v in line 12 and node u in line 8 and 15 can be done in $O(\deg(v, G'))$ and $O(\deg(u, G'))$ time with the efficient data structure proposed in [22]. Since each node is removed once, the time complex of compute$\beta_{\max}$ is bounded by $O(m)$. Similarly, the running time compute$\alpha_{\max}$ is also $O(m)$. Thus, the time complexity of BasicDecom is $O(\text{dmax} \cdot m)$. □

4.2 A Computation-sharing Core Decomposition Algorithm

Algorithm 3 processes the nodes in $U(G)$ and $V(G)$ independently and has to conduct $O(\text{dmax})$ iterations to complete the core decomposition. However, dmax can be very large in real graphs [6], which makes Algorithm 2 impractical. In this section, we reduce the number of iterations to 2δ, where δ is the maximum value

Algorithm 3 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: δ ← the maximum value such that C_{δ,δ} ≠ ∅
2: for α = 1 to δ do
3:   compute β_max(G, k);
4: for β = 1 to δ do
5:   compute α_max(G, β);
6: procedure compute β+_max(G, α)
7:   line 6-8 of Algorithm 2;
8:   while G' ≠ ∅ do
9:     β ← min_{v∈V(G')} deg(v, G');
10:    while ∃v ∈ V(G') : deg(v, G') ≤ β do
11:      remove v and its incident edges from G';
12:      for i = 1 to β do
13:        if α_{max,i}(v) < α then
14:          α_{max,i}(v) ← α;
15:      while ∃u ∈ U(G') : deg(u, G') < α do
16:        β_{max,α}(u) ← β;
17:        remove u and its incident edges from G';
18: procedure compute α+_max(G, β)
19:   line 8-19 by interchanging u with v, U with V, α with β;
```

such that $C_{\delta,\delta}$ is nonempty and is bounded by $\sqrt{m}$, by exploring computation-sharing opportunities during processing the nodes in $U(G)$ and $V(G)$.

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

Lemma 4.1: *Given a bipartite graph G, $\alpha_{\max,\beta}(v) \le \delta$, for all $\beta > \delta$ and $v \in V(G)$; $\beta_{\max,\alpha}(v) \le \delta$, 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 $\alpha_{\max,\beta}(v)$, $C_{\delta+1,\delta+1}$ must be nonempty, which contradicts the definition of δ. Thus, $\alpha_{\max,\beta}(v) \le \delta$, 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 $\alpha \le \delta$ 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 $\beta \le \delta$ and each $v \in V(G)$ but also $\beta_{\max,\alpha}(u)$ for each $\alpha > \delta$ and each $u \in U(G)$.

Algorithm. Following above idea, our computation-sharing algorithm, ComShrDecom, is shown in Algorithm 3. In Algorithm 3, 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

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 6: Decomposition procedure of Algorithm 3

$\beta_{\max,\alpha}(u)$ 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 (line 2-5).

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 11), for each i from 1 to β, if $\alpha_{\max,i}(v) < \alpha$, it updates the corresponding $\alpha_{\max,i}(v)$ as α (line 13-14). This is because when v is removed, v is in a $C_{\alpha,\beta}$, 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 4.2: Figure 6 shows the procedure of ComShrDecom to compute $\beta_{\max,\alpha}(u_1)$ and $\alpha_{\max,\beta}(v_4)$. ComShrDecom first computes $\delta = 2$, thus it needs 4 iterations to finish the decomposition. Compared with BasicDecom, ComShrDecom 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 ComShrDecom 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. □

Theorem 4.2: *Given a bipartite graph G, the time complexity of Algorithm 3 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 12-14. Since the maximum possible value of β in line 12 can be no larger than $\deg(v,G)$, the time complexity of line 12-14 is $O(\deg(v,G))$. Hence, compute$\beta^+_{\max}$ runs in $O(m)$ time. Similarly, compute$\alpha^+_{\max}$ runs in $O(m)$ time. Thus, Algorithm 3 also runs in $O(\delta \cdot m)$.

Let g denote the subgraph induced by $C_{\delta,\delta}$. 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 3, 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 need 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 3 is within a constant factor of 2 to the optimal number of iterations.

4.3 Index Construction Algorithm

After obtaining the core decomposition result, we can construct BiCore-Index based on its structure directly. For $\mathbb{I}^U$, we first constructs $\mathbb{I}^U$.NB based on $\beta_{\max,\alpha}(u)$ and sort all the node blocks with their associated (α,β) value. After that, we store the address of the first node block (α,β') such that $\beta' \geq \beta$ in $\mathbb{I}^U[\alpha][\beta]$ and the address of the α-th array in $\mathbb{I}^U[\alpha]$. $\mathbb{I}^V$is constructed similarly. For a given bipartite graph G and its core decomposition result, the index can be constructed in $O(m)$ time.

5 EXTENSIONS

Index Maintenance on Dynamic Graphs. When graphs are dynamically updated, a straightforward solution to maintain BiCore-Index is reconstructing it from scratch, which is inefficient for large graphs. In this section, we discuss the incremental algorithms for maintaining BiCore-Index on dynamic graphs. We mainly focus on edge insertion and deletion, because node insertion/deletion can be treated as a sequence of edge insertions/deletions. For brevity, we use G^+/G^- to represent the updated graph after (u,v) is inserted/removed. Since the computation of $\beta_{\max,\alpha}(u)$ and $\alpha_{\max,\beta}(v)$ is the most time-consuming part for maintaining BiCore-Index, we concentrate on computing these values here.

Edge Insertion. Suppose that an edge (u,v) is inserted, for an integer α, let $\tau_\alpha = \min\{\beta_{\max,\alpha}(u,G), \beta_{\max,\alpha}(v,G)\}$. Because both u and v have already been included in any $C_{\alpha,\beta}$ whose $\beta < \tau_\alpha$, these (α,β)-cores will not change after the insertion. Thus, we only need to consider those nodes u' whose $\beta_{\max,\alpha}(u',G) \geq \tau_\alpha$, i.e., nodes in C_{α,τ_α}. Similarly, for an integer β, let $\tau_\beta = \min\{\alpha_{\max,\beta}(u,G), \alpha_{\max,\beta}(v,G)\}$, all the nodes u' whose $\alpha_{\max,\beta}(u',G^+)$ value will change after the insertion of (u,v) are contained in $C_{\tau_\beta,\beta}$. Thus, we retrieve the induced subgraph of C_{α,τ_α} ($C_{\tau_\beta,\beta}$) through the BiCore-Index and compute $\beta_{\max,\alpha}(u',G^+)$ ($\alpha_{\max,\beta}(u',G^+)$) for those nodes by using compute$\beta^+_{\max}$ (compute$\alpha^+_{\max}$).

Edge Removal. Similarly to edge insertion case, for an integer α, all the nodes u' whose $\beta_{\max,\alpha}(u',G)$ will change after the removal of (u,v) must have $\beta_{\max,\alpha}(u',G) \leq \tau_\alpha$. This is because edge (u,v) is not included in any induced subgraph of $C_{\alpha,\beta}$ where $\beta > \tau_\alpha$. Similarly, for an integer β, we know all the nodes u' whose $\alpha_{\max,\beta}(u',G)$ will change after the removal of (u,v) must have $\alpha_{\max,\beta}(u',G) \leq \tau_\beta$. Thus, we retrieve C_{α,τ_α} ($C_{\tau_\beta,\beta}$) through the BiCore-Index and compute $\beta_{\max,\alpha}(u',G^-)$ ($\alpha_{\max,\beta}(u',G^-)$) for those nodes similarly as edge insertion case.

Batch Update. 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. After that we rearrange the order such that all the inserted edges come after removed edges. Thus, we can treat batch update as first removing a set of edges then inserting another set of edges. When a set of edges is inserted, our incremental algorithm can be easily extended to handle such situation. For an integer $\alpha(\beta)$, we set $\pi_\alpha(\pi_\beta)$ as the smallest $\beta_{\max,\alpha}(u,G)(\alpha_{\max,\beta}(u,G))$ where u is incident to at least one inserted edge. Then, we use the method in *Edge Insertion* to update BiCore-Index by replacing $\tau_\alpha(\tau_\beta)$ as $\pi_\alpha(\pi_\beta)$. Similarly, when a set of edges is removed, we set $\pi_\alpha(\pi_\beta)$ as the largest $\beta_{\max,\alpha}(u,G)(\alpha_{\max,\beta}(u,G))$ where u is incident to at least one removed edge. Then, we can use the method in *Edge Removal* by replacing $\tau_\alpha(\tau_\beta)$ as $\pi_\alpha(\pi_\beta)$.

Parallel Index Construction. Algorithm 3 can be easily extended to run in parallel. We first compute δ as line 1 in Algorithm 3. Then, we run compute$\beta^+_{\max}$ for each α from 1 to δ and compute$\alpha^+_{\max}$ for

each β from 1 to δ in parallel, i.e., each thread is in charge of a continuous subrange of possible values of α or β. To avoid race condition, we keep a copy of $\beta_{\max,\alpha}(u,G)$ and $\alpha_{\max,\beta}(v,G)$ for each thread. At last, we set $\beta_{\max,\alpha}(u,G)$ and $\alpha_{\max,\beta}(v,G)$ as the largest value among the results computed in all threads.

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 (UD) in which all edges are added with the same probability. The details of these graphs are shown in Table 1. 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:

- Baseline: the state-of-the-art existing solution proposed in [11] (introduced in Section 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 4.3).
- ComShrDecom: Our proposed index construction algorithm based on computation-sharing decomposition algorithm (Algorithm 3 + Index construction algorithm in Section 4.3).
- BiCore-Index-Ins: Our algorithm for handling edge insertion.
- BiCore-Index-Rem: Our algorithm for handling edge removal.
- BiCore-Index-Batch: Our algorithm for handling batch update.
- ParallelDecom: Our proposed algorithm for parallel index construction.

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.

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).

Exp-1: Query performance on different datasets. Figure 7 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).

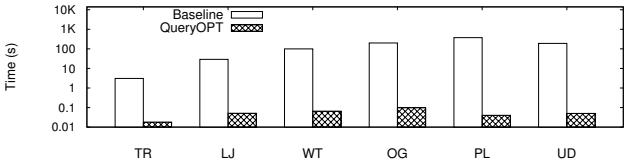

Figure 7: Query performance on different datasets

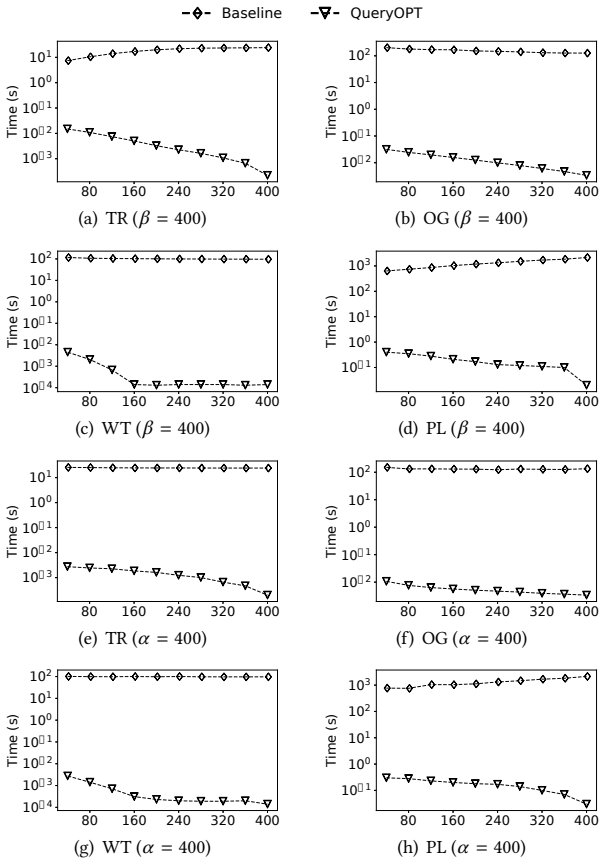

Figure 8: Query time for different α (β)

Exp-2: Varying α (β). The running time of Baseline and QueryOPT when varying α (β) is reported in Figure 8. We just show the results on four real graphs due to the similar trends. As shown in Figure 8, 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, 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 $C_{\alpha,\beta}$ decreases and the running time of QueryOPT depends on the size of $C_{\alpha,\beta}$ while that of Baseline depends on the size of input graph. The results when varying β is similar to varying α.

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

Table 1: 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
LG (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

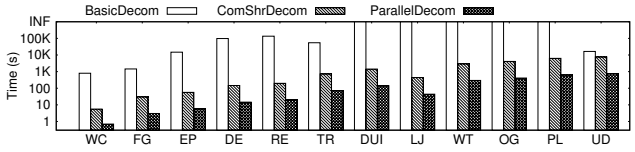

Figure 9: Index construction time for different datasets

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 1. For ease of comparison, we also report the graph size in Table 1. As shown in Table 1, 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.

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 9. 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 1 as these values directly relates to their running time. As shown in Table 1, 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 4 and are consistent with the results in Exp-4.

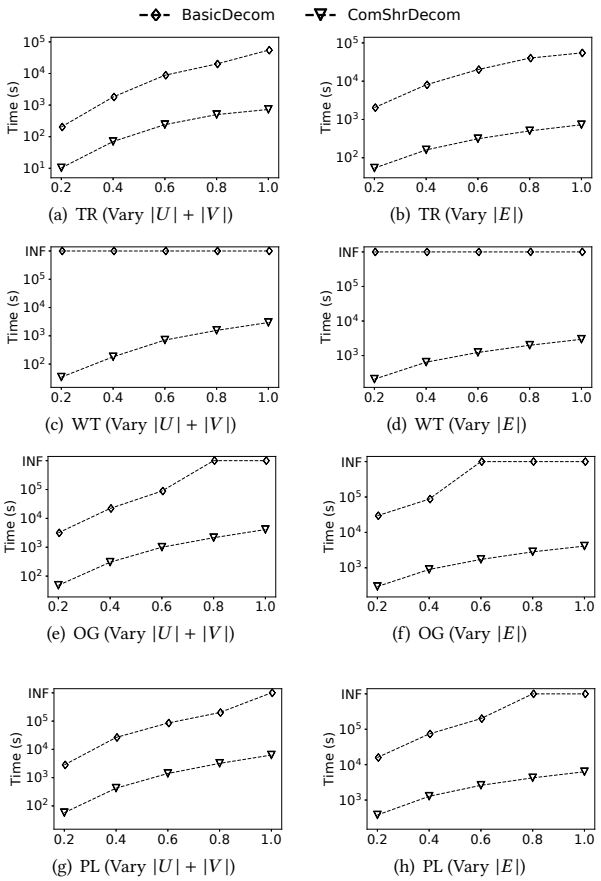

Figure 10: Scalability of index construction algorithms

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 10 since trends are similar on all other datasets. As shown in Figure 10, 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.

6.3 Dynamic Maintenance and Parallel Construction

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-7: Index maintenance on different datasets. For BiCore-Index-Ins and BiCore-Index-Rem, we randomly remove 5000 distinct existing edges from the graph and report the average processing time for each edge removal. After that, 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 and each edge is randomly chosen as insertion or removal. We report the processing time in Figure 11. 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-Rem can handle edge insertion and removal in 31s and 35s respectively while the baseline solution ComShrDecom requires 2,953s. This is because our proposed algorithms are incremental algorithms and save lots of unnecessary computation. Moreover, BiCore-Index-Batch can handle batch update on WT and OG in 292s and 857s respectively.

Exp-8: Parallel index construction. In this experiment, we implement parallel index construction algorithm ParallelDecom using C++11 thread class and test it with 12 cores in default. The running time of ParallelDecom on all datasets is reported in Figure 9. ParallelDecom achieves one order magnitude improvement over BasicDecom. For, example, in WT, the running time of ParallelDecom is 317s while ComShrDecom costs 2953s. We also report performance of ParallelDecom on TR, WT, OG, and PL with different number of cores in Figure 12. The running time of ParallelDecom is almost inversely proportional to the number of cores, which shows that ParallelDecom is efficient in practice.

7 RELATED WORK

Dense Subgraphs in Bipartite Graphs. (α, β)-core is first introduced in [1]. [10] and [11] extend the linear k-core mining algorithm to compute (α, β)-core. [19, 23] generalizes the k-clique on unipartite graph to biclique on bipartite graphs. [32] proves that finding the maximum edge biclique is NP-complete. [53] proposes an efficient algorithm to enumerate all bicliques. [26, 39] relax the definition of biclique to introduce quasi-biclique on bipartite graphs and propose heuristic algorithms to enumerate all quasi-bicliques. [37] defines a framework of bipartite subgraphs based on the butterfly motif (2,2-biclique) to model the dense regions in a hierarchical structure. [34] proposes efficient algorithms for counting the butterfly motif.

Bipartite Graph Models. [16] shows that all complex networks can be decomposed into underlying bipartite structures sharing some important statistics. [17, 29] model bipartite graphs by assigning degree distribution to each node set separately. [20] 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 [17].

Cohesive Subgraph Detection in Unipartite Graphs. Seidman first introduces k-core in [38]. [7] gives an efficient linear-time algorithm for core decomposition. Core decomposition is also studied in weighted graphs [14] and directed graphs [13]. Algorithms for core number maintenance in dynamic graphs are proposed in [35, 36, 54]. Application of k-core can be found in social networks [14, 46], graph visualization[3, 51], protein interaction network analysis [5, 42] and so on. Other cohesive subgraph models are also studied recently, such as clique [43, 44, 48], k-edge connected component [45, 47], and k-mutual-friend subgraph model [50].

8 CONCLUSION

In this paper, 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 maintain the index in dynamic graphs and construct it with parallel algorithm. The experimental results demonstrate the efficiency of our proposed algorithms.

REFERENCES

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

[2] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Elisa Bertino, and Norman Foo. 2013. Collusion Detection in Online Rating Systems. In *Web Technologies and Applications*, Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–207.

[3] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. k-core decomposition: A tool for the visualization of large scale networks. *arXiv preprint cs/0504107* (2005).

[4] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. 2009. Group Recommendation: Semantics and Efficiency. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 754–765. https://doi.org/10.14778/1687627.1687713

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

[6] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509 arXiv:http://science.sciencemag.org/content/286/5439/509.full.pdf

[7] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

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

[9] Lucas Augusto Montalvão Costa Carvalho and Hendrik Teixeira Macedo. 2013. Users' satisfaction in recommendation systems for groups: an approach based on noncooperative games. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 951–958.

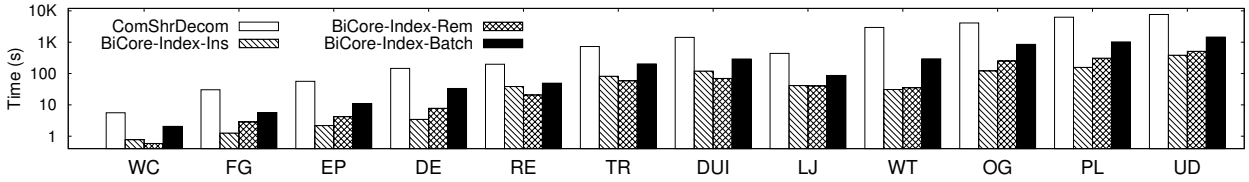

Figure 11: Time cost for index maintenance

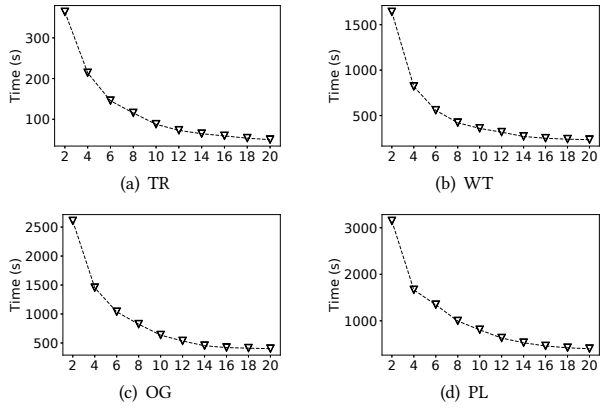

Figure 12: ParallelDecom **with varying number of cores**

[10] Monika CerinÅąek and Vladimir Batagelj. 2015. Generalized two-mode cores. *Social Networks* 42 (2015), 80 – 87. https://doi.org/10.1016/j.socnet.2015.04.001

[11] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient Fault-Tolerant Group Recommendation Using Alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, New York, NY, USA, 2047–2050. https://doi.org/10.1145/3132847.3133130

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

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

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

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

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

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

[18] S. Gunnemann, E. Muller, S. Raubach, and T. Seidl. 2011. Flexible Fault Tolerant Subspace Clustering for Data with Missing Values. In *2011 IEEE 11th International Conference on Data Mining*. 231–240. https://doi.org/10.1109/ICDM.2011.70

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

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

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

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

[23] Sune Lehmann, Martin Schwartz, and Lars Kai Hansen. 2008. Biclique communities. *Phys. Rev. E* 78 (Jul 2008), 016108. Issue 1. https://doi.org/10.1103/PhysRevE.78.016108

[24] Michael Ley. 2002. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *Proc. Int. Symposium on String Processing and Information Retrieval*. 1–10.

[25] Jinyan Li, Kelvin Sim, Guimei Liu, and Limsoon Wong. [n. d.]. *Maximal Quasi-Bicliques with Balanced Noise Tolerance: Concepts and Co-clustering Applications*. 72–83. https://doi.org/10.1137/1.9781611972788.7 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9781611972788.7

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

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

[28] Xiaowen Liu, Jinyan Li, and Lusheng Wang. 2010. Modeling Protein Interacting Groups by Quasi-Bicliques: Complexity, Algorithm, and Application. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 7, 2 (April 2010), 354–364. https://doi.org/10.1109/TCBB.2008.61

[29] Mark EJ Newman, Steven H Strogatz, and Duncan J Watts. 2001. Random graphs with arbitrary degree distributions and their applications. *Physical review E* 64, 2 (2001), 026118.

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

[31] Eirini Ntoutsi, Kostas Stefanidis, Katharina Rausch, and Hans-Peter Kriegel. 2014. "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)*. ACM, New York, NY, USA, 729–738. https://doi.org/10.1145/2661829.2662026

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

[33] Ardian Kristanto Poernomo and Vivekanand Gopalkrishnan. 2009. 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)*. ACM, New York, NY, USA, 697–706. https://doi.org/10.1145/1557019.1557097

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

[35] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

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

[37] Ahmet Erdem Sarıyüce and Ali Pinar. 2018. Peeling Bipartite Networks for Dense Subgraph Discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18)*. ACM, New York, NY, USA, 504–512. https://doi.org/10.1145/3159652.3159678

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

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

[40] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. 2006. 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*. ACM, 501–508.

[41] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation

in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. ACM, New York, NY, USA, 839–848. https://doi.org/10.1145/3219819.3219869

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

[43] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2015. Diversified top-k clique search. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 387–398. https://doi.org/10.1109/ICDE.2015.7113300

[44] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2016. Diversified top-k clique search. *VLDB J.* 25, 2 (2016), 171–196. https://doi.org/10.1007/s00778-015-0408-z

[45] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2016. I/O Efficient ECC Graph Decomposition via Graph Reduction. *PVLDB* 9, 7 (2016), 516–527. https://doi.org/10.14778/2904483.2904484

[46] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Effective and Efficient Dynamic Graph Coloring. *PVLDB* 11, 3 (2017), 338–351. https://doi.org/10.14778/3157794.3157802

[47] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.* 26, 2 (2017), 275–300. https://doi.org/10.1007/s00778-016-0451-4

[48] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2018. Index-Based Densest Clique Percolation Community Search in Networks. *IEEE Trans. Knowl. Data Eng.* 30, 5 (2018), 922–935. https://doi.org/10.1109/TKDE.2017.2783933

[49] Quan Yuan, Gao Cong, and Chin-Yew Lin. 2014. COM: A Generative Model for Group Recommendation. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 163–172. https://doi.org/10.1145/2623330.2623616

[50] Fan Zhang, Long Yuan, Ying Zhang, Lu Qin, Xuemin Lin, and Alexander Zhou. 2018. Discovering Strong Communities with User Engagement and Tie Strength. In *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part I*. 425–441. https://doi.org/10.1007/978-3-319-91452-7_28

[51] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting analyzing and visualizing triangle k-core motifs within networks. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 1049–1060.

[52] Yun Zhang, Charles A. Phillips, Gary L. Rogers, Erich J. Baker, Elissa J. Chesler, and Michael A. Langston. 2014. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC Bioinformatics* 15, 1 (15 Apr 2014), 110. https://doi.org/10.1186/1471-2105-15-110

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

[54] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 337–348. https://doi.org/10.1109/ICDE.2017.93